Abhijit Ghatak

Deep Learning with R

Springer

Deep Learning with R

Abhijit Ghatak

Deep Learning with R

Springer

Abhijit Ghatak
Kolkata, India

ISBN 978-981-13-5849-4 ISBN 978-981-13-5850-0 (eBook)
https://doi.org/10.1007/978-981-13-5850-0

Library of Congress Control Number: 2019933713

© Springer Nature Singapore Pte Ltd. 2019
This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.
The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.
The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, expressed or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

This Springer imprint is published by the registered company Springer Nature Singapore Pte Ltd.
The registered company address is: 152 Beach Road, #21-01/04 Gateway East, Singapore 189721, Singapore

I dedicate this book to the deep learning fraternity at large who are trying their best, to get systems to reason over longtime horizons.

Preface

Artificial Intelligence

The term 'Artificial Intelligence' (AI) was coined by John McCarthy in 1956, but the journey to understand if machines can truly *think* began much before that. Vannevar Bush [1] in his seminal work—*As We May Think*,[1]—proposed a system which amplifies people's own knowledge and understanding.

Alan Turing was a pioneer in bringing AI from the realm of philosophical prediction to reality. He wrote a paper on the notion of machines being able to simulate human beings and the ability to do intelligent things. He also realized in the 1950s that it would need a greater understanding of human intelligence before we could hope to build machines which would "think" like humans. His paper titled "Computing Machinery and Intelligence" in 1950 (published in a philosophical journal called *Mind*) opened the doors to the field that would be called *AI*, much before the term was actually adopted. The paper defined what would be known as the Turing test,[2] which is a model for measuring "intelligence."

Significant AI breakthroughs have been promised "in the next 10 years," for the past 60 years. One of the proponents of AI, Marvin Minsky, claimed in 1967—"Within a generation …, the problem of creating "artificial intelligence" will substantially be solved," and in 1970, he quantified his earlier prediction by stating—"In from three to eight years we will have a machine with the general intelligence of a human being."

In the 1960s and early 1970s, several other experts believed it to be right around the corner. When it did not happen, it resulted in drying up of funds and a decline in research activities, resulting in what we term as the first *AI winter*.

During the 1980s, interest in an approach to AI known as *expert systems* started gathering momentum and a significant amount of money was being spent on

[1] https://www.theatlantic.com/magazine/archive/1945/07/as-we-may-think/303881/.

[2] https://www.turing.org.uk/scrapbook/test.html.

research and development. By the beginning of the 1990s, due to the limited scope of *expert systems*, interest waned and this resulted in the second *AI winter*. Somehow, it appeared that expectations in AI always outpaced the results.

Evolution of Expert Systems to Machine Learning

An expert system (ES) is a program that is designed to solve problems in a specific domain, which can replace a human expert. By mimicking the thinking of human experts, the expert system was envisaged to analyze and make decisions.

The knowledge base of an ES contains both factual knowledge and heuristic knowledge. The ES inference engine was supposed to provide a methodology for reasoning the information present in the knowledge base. Its goal was to come up with a recommendation, and to do so, it combined the facts of a specific case (input data), with the knowledge contained in the knowledge base (rules), resulting in a particular recommendation (answers).

Though ES was suitable to solve some well-defined logical problems, it proved otherwise in solving other types of complex problems like image classification and natural language processing (NLP). As a result, ES did not live up to its expectations and gave rise to a shift from the rule-based approach to a data-driven approach. This paved the way to a new era in AI—machine learning.

Research over the past 60 years has resulted in significant advances in search algorithms, machine learning algorithms, and integrating statistical analysis to understand the world at large.

In machine learning, the system is trained rather than explicitly programmed (unlike that in ES). By exposing large quantities of known facts (input data and answers) to a learning mechanism and performing tuning sessions, we get a system that can make predictions or classifications of unseen input data. It does this by finding out the statistical structure of the input data (and the answers) and comes up with rules for automating the task.

Starting in the 1990s, machine learning has quickly become the most popular subfield of AI. This trend has also been driven by the availability of faster computing and availability of diverse data sets.

A machine learning algorithm transforms its input data into meaningful outputs by a process known as *representations*. Representations are transformations of the input data, to represent it closer to the expected output. "Learning," in the context of machine learning, is an automatic search process for better *representations* of data. Machine learning algorithms find these *representations* by searching through a predefined set of operations.

To summarize, machine learning is searching for useful *representations* of the input data within a predefined space, using the loss function (difference between the actual output and the estimated output) as a feedback to modify the parameters of the model.

Machine Learning and Deep Learning

It turns out that machine learning focuses on learning only one or two layers of representations of the input data. This proved intractable for solving human perception problems like image classification, text-to-speech translation, handwriting transcription, etc. Therefore, it gave way to a new take on learning representations, which put an emphasis on learning multiple successive layers of representations, resulting in deep learning. The word *deep* in deep learning only implies the number of layers used in a deep learning model.

In deep learning, we deal with layers. A layer is a data transformation function which carries out the transformation of the data which goes through that layer. These transformations are parametrized by a set of weights and biases, which determine the transformation behavior at that layer.

Deep learning is a specific subfield of machine learning, which makes use of tens/hundreds of successive layers of representations. The specification of what a layer does to its input is stored in the layer's parameters. Learning in deep learning can also be defined as finding a set of values for the parameters of each layer of a deep learning model, which will result in the appropriate mapping of the inputs to the associated answers (outputs).

Deep learning has been proven to be better than conventional machine learning algorithms for these "perceptual" tasks, but not yet proven to be better in other domains as well.

Applications and Research in Deep Learning

Deep learning has been gaining traction in many fields, and some of them are listed below. Although most of the work to this date are proof-of-concept (PoC), some of the results have actually provided a new physical insight.

- **Engineering**—Signal processing techniques using traditional machine learning exploit shallow architectures often containing a single layer of nonlinear feature transformation. Examples of shallow architecture models are conventional hidden Markov models (HMMs), linear or nonlinear dynamical systems, conditional random fields (CRFs), maximum entropy (MaxEnt) models, support vector machines (SVMs), kernel regression, multilayer perceptron (MLP) with a single hidden layer, etc. Signal processing using machine learning also depends a lot on handcrafted features. Deep learning can help in getting task-specific feature representations, learning how to deal with noise in the signal and also work with long-term sequential behaviors. Vision and speech signals require deep architectures for extracting complex structures, and deep learning can provide the necessary architecture. Specific signal processing areas where deep

learning is being applied are speech/audio, image/video, language processing, and information retrieval. All this can be improved with better feature extraction at every layer, more powerful discriminative optimization techniques, and more advanced architectures for modeling sequential data.

- **Neuroscience**—Cutting-edge research in human neuroscience using deep learning is already happening. The cortical activity of "imagination" is being studied to unveil the computational and system mechanisms that underpin the phenomena of human imagination. Deep learning is being used to understand certain neurophysiological phenomena, such as the firing properties of dopamine neurons in the mammalian *basal ganglia* (a group of subcortical nuclei of different origin, in the brains of vertebrates including humans, which are associated with a variety of functions like eye movements, emotion, cognition, and control of voluntary motor movements). There is a growing community who are working on the need to distill intelligence into algorithms so that they incrementally mimic the human brain.
- **Oncology**—Cancer is the second leading health-related cause of death in the world. Early detection of cancer increases the probability of survival by nearly 10 times, and deep learning has demonstrated capabilities in achieving higher diagnostic accuracy with respect to many domain experts. Cancer detection from gene expression data is challenging due to its high dimensionality and complexity. Researchers have developed DeepGene,[3] which is an advanced cancer classifier based on deep learning. It addresses the obstacles in existing *somatic point mutation-based cancer classification* (SMCC) studies, and the results outperform three widely adopted existing classifiers. Google's CNN system[4] has demonstrated the ability to identify deadline skin cancers at an accuracy rate on a par with practitioners. Shanghai University has developed a deep learning system that can accurately differentiate between benign and malignant breast tumors on ultrasound *shear wave elastography* (SWE), yielding more than 93% accuracy on the elastogram images of more than 200 patients.[5]
- **Physics**—*Conseil Europeen pour la Recherche Nucleaire* (*CERN*) at Geneva handles multiple petabytes of data per day during a single run of the Large Hadron Collider (LHC). LHC collides protons/ions in the collider, and each collision is recorded. After every collision, the trailing particles—a Higgs boson, a pair of top quarks, or some mini-black holes—are created, which leave a trailing signature. Deep learning is being used to classify and interpret these signatures.
- **Astrophysics**—Deep learning is being extensively used to classify galaxy morphologies.[6]

[3] https://bmcbioinformatics.biomedcentral.com/articles/10.1186/s12859-016-1334-9.

[4] https://www.nature.com/articles/nature21056.

[5] https://www.umbjournal.org/article/S0301-5629(17)30002-9/abstract.

[6] https://arxiv.org/abs/0908.2033.

- **Natural Language Processing**—There has been a rising number of research papers (see Fig. 1) among the research community since 2012, as is reflected in the paper titled *Recent Trends in Deep Learning Based Natural Language Processing* by Young et al.
- Near human-level proficiency has been achieved in (a) speech recognition, (b) image recognition, (c) handwriting transcription, and (d) autonomous driving. Moreover, super-human-level performance has been achieved by AlphaGo (built by Google) when it defeated the world's best player Lee Sedol at Go.

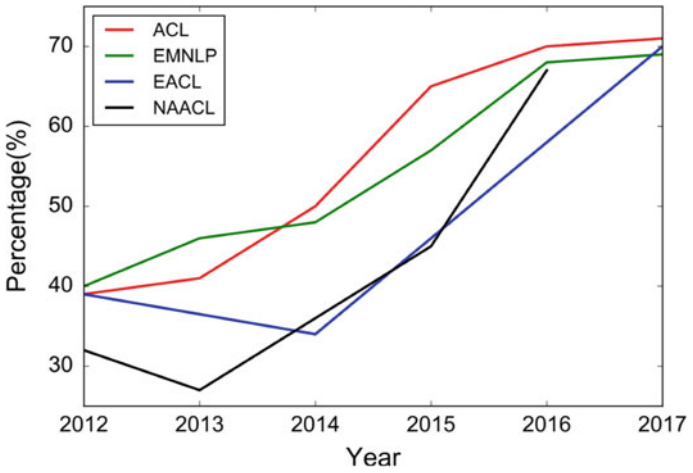

Fig. 1 Percentage of deep learning papers submitted for various conferences—Association for Computational Linguistics (ACL), Conference on Empirical Methods in Natural Language Processing (EMNLP), European Chapter of the Association for Computational Linguistics (EACL), North American Chapter of the Association for Computational Linguistics (NAACL), over the last 6 years since 2018. [2]

Intended Audience

This book has been written to address a wide spectrum of learners:

- For the beginner, this book will be useful to understand the basic concepts of machine/deep learning and the neural network architecture (Chaps. 1 and 2) before moving on to the advanced concepts of deep learning.
- For the graduate student, this book will help the reader understand the behavior of different types of neural networks by understanding the concepts, while building them up from scratch. It will also introduce the reader to relevant research papers for further exploration.

- For the data scientist who is familiar with the underlying principles of machine learning, this book will provide a practical understanding of deep learning.
- For the deep learning enthusiast, this book will explain the deep learning architecture and what goes on inside a neural network model.

An intermediate level of R programming knowledge is expected from the reader, and no previous experience of the subject is assumed.

Kolkata, India Abhijit Ghatak

Acknowledgements

Acknowledgment is an unsatisfactory word for my deepest debts.

My father bequeathed to me a love for adventure and an interest in history, literature, and mathematics.

My professors at the Faculty of Mechanical Engineering, Jadavpur University, instilled an appetite for analysis and quantitative techniques in engineering; my mentor and advisor at University of Pune, Prof. SY Bhave, motivated me to interpret the algorithm and write a program using the C language on predicting torsional vibration failures of a marine propulsion shaft using *state vectors*; and my advisors at Stevens Institute of Technology helped me to transit from a career submarine engineer in the Indian Navy to a data scientist.

My wife Sushmita lived through the slow gestation of this book. She listened and engaged with me all the way. She saw potential in this work long before I did and encouraged me to keep going.

I owe my thanks to Sunanda for painstakingly proofreading the manuscript.

I also have two old debts—`Robert Louis Stevenson` and `Arthur Conan Doyle`. In *Treasure Island*, Mr Smollet is most eager to discover the treasure and he says—"We must go on," and in *Case of Identity*, Sherlock Holmes states—"It has long been an axiom of mine that the little things are infinitely the most important." Both are profound statements in the realm of a new science, and the litterateurs had inked their thoughts claiming no distinction, when there is not a distinction between the nature of the pursuit.

I owe all of them my deepest debts.

Abhijit Ghatak

About This Book

- Deep learning is a growing area of interest to academia and industry alike. The applications of deep learning range from medical diagnostics, robotics, security and surveillance, computer vision, natural language processing, autonomous driving, etc. This has been largely possible due to a conflation of research activities around the subject and the emergence of APIs like Keras.
- This book is a sequel to **Machine Learning with R**, written by the same author, and explains deep learning from first principles—how to construct different neural network architectures and understand the hyperparameters of the neural network and the need for various optimization algorithms. The theory and the math are explained in detail before discussing the code in R. The different functions are finally merged to create a customized deep learning application. It also introduces the reader to the Keras and TensorFlow libraries in R and explains the advantage of using these libraries to get a basic model up and running.
- This book builds on the understanding of deep learning to create R-based applications on computer vision, natural language processing, and transfer learning.

This book has been written to address a wide spectrum of learners:

- For the beginner, this book will be useful to understand the basic concepts of machine/deep learning and the neural network architecture (Chaps. 1 and 2) before moving on to the advanced concepts of deep learning.
- For the graduate student, this book will help the reader to understand the behavior of different types of neural networks by understanding the concepts, while building them up from scratch. It will also introduce the reader to relevant research papers for further exploration.
- For the data scientist who is familiar with the underlying principles of machine learning, this book will provide a practical understanding of deep learning.
- For the deep learning enthusiast, this book will explain the deep learning architecture and what goes on inside a neural network model.

This book requires an intermediate level of skill in R and no previous experience of deep learning.

Contents

About the Author

Abhijit Ghatak is a Data Engineer and holds graduate degrees in Engineering and Data Science from India and USA. He started his career as a submarine engineer officer in the Indian Navy where he worked on multiple data-intensive projects involving submarine operations and submarine construction. He has thereafter worked in academia, IT consulting and as research scientist in the area of Internet of Things (IoT) and pattern recognition for the European Union. He has authored many publications in the areas of engineering, IoT and machine learning. He presently advises start-up companies on deep learning, pattern recognition and data analytics. His areas of research include IoT, stream analytics and design of deep learning systems. He can be reached at *abeghatak@gmail.com*.

Chapter 1
Introduction to Machine Learning

I believe that at the end of the century the use of words and general educated opinion will have altered so much that one will be able to speak of machines thinking without expecting to be contradicted.

Alan Turing, 1950

Abstract This chapter will introduce some of the building blocks of machine learning. Specifically, it will touch upon

- Difference between machine learning, statistics and deep learning.
- A discussion on bias and variance and how they are related to underfitting and overfitting.
- Different ways to address underfitting and overfitting, including regularization.
- The need for optimization and the gradient descent method.
- Model hyperparameters and different hyperparameters search methods.
- Quantifying and measuring loss functions of a model.

1.1 Machine Learning

Machine learning is a sub-domain of artificial intelligence (AI), which makes a system automatically discover (learn) the statistical structure of the data and convert those representations (patterns) to get closer to the expected output. The process of **learning** is improved by a measure of the feedback, which compares the computed output to the expected output. Unlike *expert systems*, the machine learning system is not explicitly programmed; it automatically searches for patterns within the *hypothesis space* and, uses the feedback signal to correct those patterns.

© Springer Nature Singapore Pte Ltd. 2019

A. Ghatak, *Deep Learning with R*,
https://doi.org/10.1007/978-981-13-5850-0_1

To enable a machine learning algorithm to work effectively on real-world data, we need to feed the machine a full range of features and possibilities to train on.

A typical machine learning workflow includes the following:

- Training the model on a training data set, tuning the model on a development set and testing the model on an unseen test data set.
- Trying out the above on different yet, appropriate algorithms using proper performance metrics.
- Selecting the most appropriate model.
- Testing the model on real-world data. If the results are not upto the speed, repeat the above by revaluating the data and/or model, possibly with different evaluation metrics.

We define data sets which we use in machine learning as follows:

- Training set: is the data on which we learn the algorithm.
- Development set: is the data on which we tune the hyperparameters of the model.
- Test set: is the data we use to evaluate the performance of our algorithm.
- Real-world set: is the data on which our selected model will be deployed.

Having data sets from different distributions can have different outcomes on the evaluation metrics of some or all of the data sets. The evaluation metrics may also differ if the model does not fit the respective data sets. We will explore these aspects during model evaluation at a later section.

1.1.1 Difference Between Machine Learning and Statistics

In machine learning, we feed labeled data in batches into the machine learning model, and the model incrementally improves its structural parameters by examining the loss, i.e., the difference between the actual and predicted values. This loss is used as a feedback to an optimization algorithm to iteratively adjust the structural parameters. Training a conventional machine learning model, therefore, consists of feeding input data to the model to train the model to learn the "best" structural parameters of the model. While machine learning focusses on predicting future data and evaluation of the model, statistics is focussed on the inference and explanation cum understanding of the phenomenon [10].

While a machine learning model is an algorithm that can learn from data without being explicitly programmed, statistical modeling is a mathematical representation of the relationship between different variables. Machine learning is a sub-domain of AI whereas, statistics is a sub-domain of mathematics.

1.1.2 Difference Between Machine Learning and Deep Learning

Traditional machine learning techniques find it hard to analyze data with complex spatial or sequence dependencies, and those that require analyzing data which need a large amount of feature engineering like problems related to computer vision and speech recognition. Perception or disambiguation is the awareness, understanding, and interpretation of information through the senses. In reference [11], deep learning is proven to be better than conventional machine learning algorithms for these "perceptual" tasks, but not yet proven to be better in other domains as well.

In deep learning, we use a similar procedure as in machine learning, by transforming the input data by a linear combination of the weights and bias through each layer, by a process known as *forward propagation*, which computes the predicted values. A loss function compares the actual and predicted values and computes a distance score between these values, thereby capturing how well the network has done on a batch of input data. This score is then used as a feedback to adjust the weights incrementally toward a direction that will lower the loss score for the current input data through an optimization algorithm. This update is done using a *backpropagation* algorithm, using the chain rule to iteratively compute gradients for every layer.

A training loop consists of a single forward propagation, calculation of the loss score and backpropagation through the layers using an optimizer to incrementally change the weights. Typically, we would need many iterations over many examples of input data to yield weight values that would minimize the loss to optimal values.

A deep learning algorithm can be thought of as a large-scale parametric model, because it has many layers and scales up to a large amount of input data. Below is the summarization of a grayscale image classification deep learning model having 1.2 million parameters. A model to classify color images can have close to 100 million parameters.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 100)               1228900
_________________________________________________________________
dense_2 (Dense)              (None, 50)                5050
_________________________________________________________________
dense_3 (Dense)              (None, 15)                765
_________________________________________________________________
dense_4 (Dense)              (None, 1)                 16
=================================================================
Total params: 1,234,731
Trainable params: 1,234,731
Non-trainable params: 0
_________________________________________________________________
```

1.2 Bias and Variance

The three major sources of error in machine/deep learning models are irreducible error, bias, and variance.

(a) **Irreducible Error**—is the error which cannot be reduced and its occurrence is due to the inherent randomness present in the data.
(b) **Bias Error**—creeps in deep learning when we introduce a simpler model for a data set which is far more "complex". To avoid bias error, we would need to increase the capacity (consider a more complex model) to match up with the complexity present in the data.
(c) **Variance**—on the contrary is present in deep learning models if we consider an overly complex algorithm for a relatively less "complex" task or data set. To avoid variance in models, we would need to increase the size of the data set.

1.3 Bias–Variance Trade-off in Machine Learning

In machine learning, we can define an appropriate trade-off point between bias and variance by discovering the appropriate model complexity with respect to the data; at which point, an increase in bias results in reduction of variance and vice-versa, as shown by the `darkgray` circle in Fig. 1.1. Any model complexity short of this trade-off point will result in an underfitted model and those beyond the trade-off point, will result in an overfitted model.

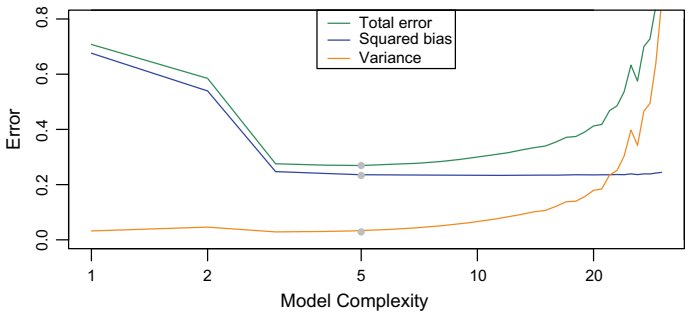

Fig. 1.1 The squared bias keeps decreasing and the variance keeps increasing as the model complexity goes up. The total error starts decreasing till it reaches a point where the model complexity is optimal, and thereafter it starts increasing. The optimal balance between the squared bias and the variance is represented by the dark gray circle

Bayes error, is the lowest possible prediction error that can be achieved and is the same as irreducible error. If we have an exact knowledge of the data distribution, then for a random process, there would still exist some errors. In statistics, the optimal error rate is also known as the Bayes error rate. In machine learning, the Bayes error can be considered as the irreducible error.
In deep learning, since we deal with problems related to human perception and therefore, we consider **Human-Level Error**, as the lowest possible error.

1.4 Addressing Bias and Variance in the Model

One of the important things to understand before we know that our model suffers from bias, variance, or a combination of both bias and variance, is to understand the *optimal error rate*. We have seen earlier that the optimal error is the Bayes error rate. The Bayes error is the best theoretical function for mapping x to y, but it is often difficult to calculate this function.

In deep learning, since we are mostly dealing with problems related to human perceptions like vision, speech, language, etc., an alternative to the Bayes error could be the human-level performance because humans are quite good at tasks related to vision, speech, language, etc. So we can ask this question—how well does our neural network model compare to human-level performance?

Based on the above, we can state the following:

(a) If there exists a large gap between human-level error and the training error, then the model is too simple in relation to human-level performance and it is a bias problem.
(b) If there is a small gap between the training error and human-level error and, a large difference between training error and validation error, it implies that our model has got high variance.
(c) If the training error is less than the human-level error, we cannot say for sure if our model suffers from bias or variance or a combination of both. There are a couple of domains where our model may surpass human-level performance and they include product recommendations, predicting transit time (Google maps), etc.

The actions we can take to address bias and variance are the following:

(a) Bias

(i) Train a more complex model.
(ii) Train for a longer duration.
(iii) Use better optimization algorithms—Momentum, RMSProp, Adam, etc.
(iv) Use different neural network architectures.
(v) Carry out better hyperparameter search.

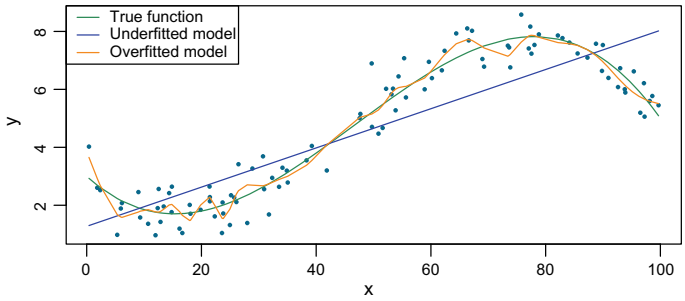

Fig. 1.2 Underfitting and Overfitting: An underfitted model is unable to capture the sinusoidal pattern of the data which is represented by the straight-line linear regression model; an overfitted model has memorized the training data and the noise and is therefore not able to generalize to the data

(b) Variance

(i) Use more data to train.
(ii) Use regularization methods, ℓ_2, dropout, etc.
(iii) Use data augmentation
(iv) Use different neural network architectures.
(v) Carry out better hyperparameter search.

1.5 Underfitting and Overfitting

Underfitting occurs when there is high bias present in the model and therefore cannot capture the trend in the data. Overfitting occurs when the model is highly complex resulting in the model capturing the noise present in the data, rather than capturing the trend in the data.

This is illustrated in Fig. 1.2, where true model is a fourth polynomial model represented by the green curve. The overfitted model oscillates wildly and the underfitted model can barely capture the trend in the data set.

1.6 Loss Function

The objective function of every algorithm is to reduce the loss $\mathcal{L}(w)$, which can be represented as

$$\mathcal{L}(w) = \textit{Measure of fit} + \textit{Measure of model complexity}$$

In machine learning, loss is measured by the sum of

- the **fit** of the model.
- the **complexity** of the model.

The measure of the model's fit is determined by

- the *MSE* (Mean Squared Error) for regression,
- *CE* (Classification Error) for classification.

Higher the model complexity, higher is the propensity for the model to capture the noise in the data by ignoring the signal. A measure of the model complexity is determined by

- the sum of the absolute values of the structural parameters of the model (ℓ_1 *regularization*),
- the sum of the squared values of the structural parameters of the model (ℓ_2 *regularization*).

1.7 Regularization

Regularization is the process used to reduce the complexity of the model.

In deep learning, we will use the ℓ_2 regularization technique

$$\ell_2 = w_0^2 + w_1^2 + \cdots + w_n^2 = \sum_{i=0}^{n} w_i^2 = \|w\|_2^2$$

Our objective is to select the model's structural parameters w_i, such that we minimize the loss function $\mathcal{L}(w)$, by using a weight decay regularization parameter λ on the ℓ_2-norm of the parameters such that, it penalizes the model for larger values of the parameters.

$$\mathcal{L}(w) = \textit{Error metric} + \lambda \|w\|_2^2$$

When λ is zero, there is no regularization; values greater than zero force the weights to take a smaller value. When $\lambda = \infty$, the weights become zero.

Since we are trying to minimize the loss function with respect to the weights, we need an optimizing algorithm, to arrive at the optimal value of the weights. One of the optimizing algorithms which we use in machine learning is the gradient descent. Other popular optimization algorithms used in deep learning is discussed in Sect. 5.

Regularization, significantly reduces the variance of a deep learning model, without any substantial increase in its bias.

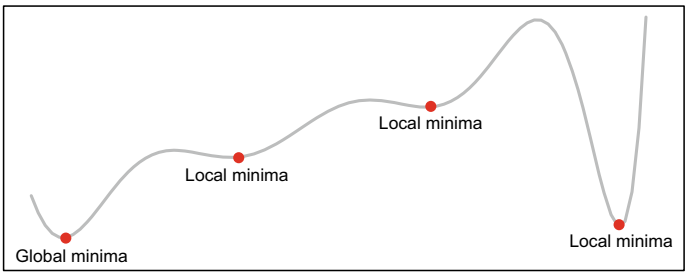

Fig. 1.3 Gradient descent involves reaching the global minima, sometimes traversing through many local minima

1.8 Gradient Descent

Gradient descent is an optimization algorithm used to find the values of weights and biases of the model with an objective to minimize the loss function.

Analogically, this can be visualized as "descending" to the lowest point of a valley (the lowest error value), known as the global minima, as shown in Fig. 1.3. In the process, the descent also has a possibility of getting stuck at a local minima and there are ways to work around and come out of the local minima by selecting the correct learning-rate hyperparameter.

The cost of the model is the sum of all the losses associated with each training example. The gradient of each parameter is then calculated by a process known as **batch gradient descent**. This is the most basic form of gradient descent because we compute the cost as the sum of the gradients of the entire batch of training examples.

In Fig. 1.4, we consider the loss function with respect to one weight (in reality we need to consider the loss function with respect to all weights and biases of the model). To move down the slope, we need to tweak the parameters of the model by calculating the gradient ($\frac{\partial \mathcal{J}}{\partial W}$) of the loss function, which will always point toward the nearest local minima.

Having found the magnitude and direction of the gradient, we now need to nudge the weight by a **hyperparameter** known as the learning rate and then update the value of the new weight (toward the direction of the minima). Mathematically, for one iteration for an observation j and learning rate α, the weight update at time step $(t+1)$ can be written as follows:

$$\text{Repeat till convergence}$$
$$w_j^{(t+1)} = w_j^{(t)} - \alpha \frac{\partial \mathcal{J}}{\partial w} \tag{1.8.1}$$

Figure 1.5 is a contour plot showing the steps of a gradient descent optimization for a `sigmoid` activation neural network having an input with two features. In a real model, the inputs may have many features and the loss function may have multiple local minima. The gradients are calculated for all the parameters while iterating over

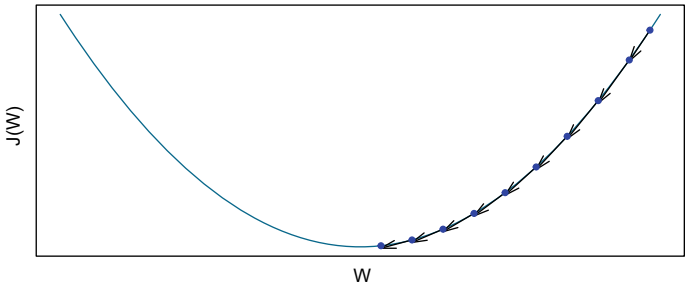

Fig. 1.4 Gradient descent: Rolling down to the minima by updating the weights by the gradient of the loss function

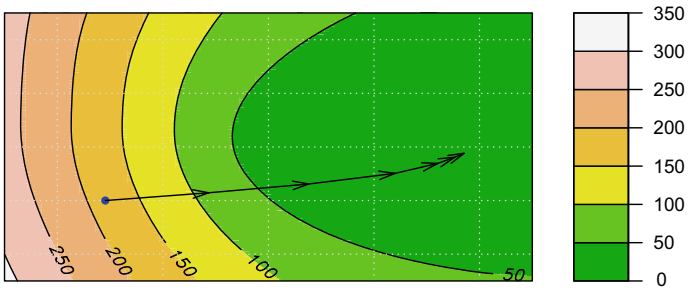

Fig. 1.5 A contour plot showing the cost contours of a sigmoid activation neural network and the cost minimization steps using the gradient descent optimization function

all the training examples. This makes things way harder to visualize, as the plot will have many multiple dimensions.

For large size data sets, batch gradient descent optimization algorithm can take a long time to compute because it considers all the examples in the data to complete one iteration. In such cases, we can consider a subset of the training examples and split our training data into mini-batches. In the **mini-batch gradient descent** method, the parameters are updated based on the current mini-batch and we continue iterating over all the mini-batches till we have seen the entire data set. The process of looking at all the mini-batches is referred to as an **epoch**.

When the mini-batch size is set to one we perform an update on a single training example, and this is a special case of the mini-batch gradient descent known as **stochastic gradient descent**. It is called "stochastic" because it randomly chooses a single example to perform an update till all the examples in the training data set have been seen.

Ideally, in any optimization algorithm, our objective is to find the global minimum of the function, which would represent the best possible parameter values. Depending on where we start in the parameter space, it is likely that we may encounter local minima and saddlepoints along the way. Saddlepoints are a special case of local minima where the derivatives in orthogonal directions are both zero. While dealing

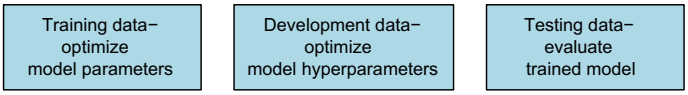

Fig. 1.6 The training process involves optimizing the cost to arrive at the optimal structural parameters using the training data, optimization of the hyperparameters using the development data, and evaluating the final model using the testing data

with high-dimensional spaces, the chances of discovering the global minima is quite low (Fig. 1.3).

There are many approaches to overcome this problem—use a moving average gradient (Momentum), adaptively scaling the learning-rate in each dimension according to the exponentially weighted average of the gradient (RMSProp) and by using a combination of the moving average gradient and adaptive learning rates (Adam). These methods have been discussed in Sects. 5.3.6. Figure 1.6 is a schematic of how the model deals with different sections of the available data.

1.9 Hyperparameter Tuning

During the model training process, our training algorithm handles three categories of data:

- The input training data—is used to configure our model to make accurate predictions from the unseen data.
- Model parameters—are the weights and biases (structural parameters) that our algorithm learns with the input data. These parameters keep changing during model training.
- Hyperparameters—are the variables that govern the training process itself. Before setting up a neural network, we need to decide the number of hidden layers, number of nodes per layer, learning rate, etc. These are configuration variables and are usually constant during a training process.

> Structural parameters like the weights and biases are learned from the input dataset, whereas the optimal hyperparameters are obtained by leveraging multiple available datasets.

A hyperparameter is a value required by our model which we really have very little idea about. These values can be learned mostly by trial and error.

While the model (structural) parameters are optimized during the training process by passing the data through a cycle of the training operation, comparing the resulting prediction with the actual value to evaluate the accuracy, and adjusting the parameters till we find the best parameter values; the hyperparameters are "tuned" by running multiple trials during a single training cycle, and the optimal hyperparameters are chosen based on the metrics (accuracy, etc), while keeping the model parameters

unchanged. In both the cases, we are modifying the composition of our model to find the best combination of the model parameters and hyperparameters.

1.9.1 Searching for Hyperparameters

In standard supervised learning problems, we try to find the best hypothesis in a given space for a given learning algorithm. In the context of hyperparameter optimization, we are interested in minimizing the validation error of a model parameterized by a vector of weights and biases, with respect to a vector of hyperparameters. In hyperparameter optimization, we try to find a configuration so that the optimized learning algorithm will produce a model that generalizes well to new data. To facilitate this, we can opt for different search strategies:

- A *grid search* is sometimes employed for hyperparameter tuning. With grid search, we build a model for each possible combination of all of the hyperparameter values in the grid, evaluate each model and select the hyperparameters, which give the best results.
- With *random search*, instead of providing a discrete set of values to explore, we consider a statistical distribution of each hyperparameter, and the hyperparameter values are randomly sampled from the distribution.
- A *greedy search* will pick whatever is the most likely first parameter. This may not be a good idea always.
- An *exact search* algorithm as its name signifies, searches for the exact value. Algorithms belonging to this search methodology are `BFS` *(Breadth First Search)* and `DFS` *(Depth First Search)*.
- In deep learning for Recurrent Neural Networks, we apply what is known as the *Beam Search*.

We also have the Bayesian optimization technique belonging to a class of `(SMBO)` *Sequential Model-Based Optimization* [8] algorithms, wherein we use the results of our search using a particular method to improve the search for the next method. The hyperparameter metric is the objective function during hyperparameter tuning. Hyperparameter tuning finds the optimal value of this metric (a numeric value), specified by the user. The user needs to specify whether this metric needs to be maximized or minimized.

> For most data sets, only a few of the hyperparameters really matter but different hyperparameters are important for different data sets. This makes the grid search a poor choice for configuring algorithms for new data sets. [6]

1.10 Maximum Likelihood Estimation

Maximum likelihood estimation (*MLE*) is a method used to determine parameter values of the model. The parameter values are found such that they maximize the likelihood that the process described by the model produces the data that were actually observed. We would, therefore, pick the parameter values which maximizes the likelihood of our data. This is known as the maximum likelihood estimate. Mathematically, we define it as

$$\arg\max_{\theta} P(y \mid x; \theta) \tag{1.10.1}$$

To arrive at the likelihood over all observations (assuming they are identical and independent of one another, i.e., i.i.d.), we take the product of the probabilities as

$$\arg\max_{\theta} \prod_{i=1}^{n} P(y^{(i)} \mid x^{(i)}; \theta) \tag{1.10.2}$$

We assume that each data point is generated independently of the others, because if the events generating the data are independent, then the total probability of observing all our data is the product of observing each data point individually (the product of the marginal probabilities).

We know that the probability density of observing a single data point generated from a Gaussian distribution is given by

$$P(x; \mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-(\frac{(x-\mu)^2}{2\sigma^2})} \tag{1.10.3}$$

To obtain the parameters μ and σ^2 of a Gaussian distribution, we need to solve the following maximization problem:

$$\max_{\mu,\sigma^2} log(\mu, \sigma^2;\ x_1, \ldots, x_n)$$

The first-order conditions to obtain a maximum are

$$\frac{\partial}{\partial\mu} log(\mu, \sigma^2;\ x_1, \ldots, x_n) = 0$$

and,

$$\frac{\partial}{\partial\sigma^2} log(\mu, \sigma^2;\ x_1, \ldots, x_n) = 0$$

The partial derivative of the log-likelihood with respect to the mean is

$$\begin{aligned}\frac{\partial}{\partial\mu}(logP(x;\mu,\sigma)) &= \frac{\partial}{\partial\mu}\, log(\mu,\sigma^2;\ x_1,\ldots,x_n)\\ &= \frac{\partial}{\partial\mu} log\left\{\frac{1}{\sqrt{2\pi\sigma^2}}e^{-(\frac{(x-\mu)^2}{2\sigma^2})}\right\}\\ &= \frac{\partial}{\partial\mu}\left\{-\frac{n}{2}ln(2\pi)-\frac{n}{2}ln(\sigma^2)-\frac{1}{2\sigma^2}\sum_{j=1}^{n}(x_j-\mu)^2\right\}\\ &= \frac{1}{\sigma^2}\sum_{j=1}^{n}(x_j-\mu)\\ &= \frac{1}{\sigma^2}\left\{\sum_{j=1}^{n}x_j-n\mu\right\}\end{aligned} \tag{1.10.4}$$

Solving for μ in Eq. 1.10.4

$$\mu = \frac{1}{n}\sum_{j=1}^{n}x_j \tag{1.10.5}$$

Similarly, the partial derivative of the log-likelihood with respect to the variance is

$$\begin{aligned}\frac{\partial}{\partial\sigma^2}(logP(x;\mu,\sigma)) &= \frac{\partial}{\partial\sigma^2}\, log(\mu,\sigma^2;\ x_1,\ldots,x_n)\\ &= \frac{\partial}{\partial\sigma^2} log\left\{\frac{1}{\sqrt{2\pi\sigma^2}}e^{-(\frac{(x-\mu)^2}{2\sigma^2}}\right\}\\ &= \frac{\partial}{\partial\sigma^2}\left\{-\frac{n}{2}log(2\pi)-\frac{n}{2}log(\sigma^2)-\frac{1}{2\sigma^2}\sum_{j=1}^{n}(x_j-\mu)^2\right\}\\ &= -\frac{n}{2\sigma^2}-\left(\frac{1}{2}\sum_{j=1}^{n}(x_j-\mu)^2\right)\left(-\frac{1}{(\sigma^2)^2}\right)\\ &= \frac{1}{2\sigma^2}\left(\frac{1}{\sigma^2}\sum_{j=1}^{n}(x_j-\mu)^2-n\right)\end{aligned} \tag{1.10.6}$$

Assuming $\sigma^2 \neq 0$, we can solve for σ^2

$$\sigma^2 = \frac{1}{n}\sum_{j=1}^{n}(x_j-\hat{\mu})^2 \tag{1.10.7}$$

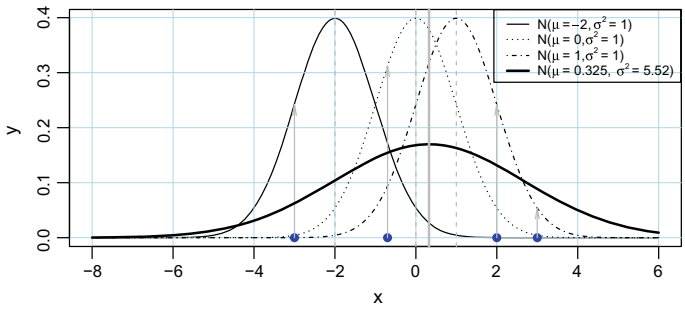

Fig. 1.7 The four data points and the possible Gaussian distributions from which they were drawn. The distributions are normally distributed with $(\mu = -2, \sigma^2 = 1)$, $(\mu = 0, \sigma^2 = 1)$, and $(\mu = 1, \sigma^2 = 1)$. The maximum likelihood estimation predicts that the data points belong to the Gaussian distribution defined with $(\mu = 0.325, \sigma^2 = 5.52)$ by maximizing the probability of observing the data

Let us consider four data points on the x-axis. We would want to know which curve was most likely responsible, for our observed data points. We will use *MLE* to find the values of μ and σ^2 to discover the gaussian distribution, that best fits our data (Fig. 1.7).

```
points <- c(-3, -0.7, 2, 3)
(mu = sum(points)/length(points)) # Refer Eq. 1.10.5
```

```
[1] 0.325
```

```
(var = (1/length(points))*sum((points-mu)^2)) # Refer Eq. 1.10.7
```

```
[1] 5.516875
```

1.11 Quantifying Loss

1.11.1 The Cross-Entropy Loss

When we develop a machine learning model for probabilistic classification, we try to map the model inputs to probabilistic predictions by an iterative method of training the model (by adjusting the model's parameters), so that our predictions are close to the true probabilities.

To arrive at our computed predictions, which are close to the true predictions, we need to reduce the loss (*Mean Squared Error/Classification Error*). In this section, we will try to define this error in terms of the Cross-Entropy(*CE*) Loss. But before that, we will revisit *MLE*, define *Entropy* of a model, its follow up to *Cross-Entropy* and, its relation to the *Kulback–Liebler* Divergence.

In Sect. 1.10, we have seen that the maximum likelihood is that which maximizes the product of the probabilities of a data distribution, which most likely gave rise to our data. Since logarithms reduce potential underflow, due to very small likelihoods, they convert a product into a summation and finally, the natural logarithmic function being a monotonic transformation (if the value on the x-axis increases, the value on the y-axis also increases); it is but natural to apply the *log* function to our likelihood Eq. (1.10.2), to obtain the log-likelihood

$$\begin{aligned} log\ P(y \mid x; \theta) &= log \prod_{i=1}^{n} P(y^{(i)} \mid x^{(i)}; \theta) \\ &= \sum_{i}^{n} log\ P(y^{(i)} \mid x^{(i)}; \theta) \end{aligned} \tag{1.11.1}$$

1.11.2 Negative Log-Likelihood

The maximazition of the negative log-likelihood of the estimated data distribution reduces the error of our machine learning model.

In **Linear Regression**, we maximize the log-likelihood (Eq. 1.11.1) of the Gaussian distribution. We may recall that $\theta^T x = \mu$

$$\begin{aligned} log[P(y \mid x; \theta)] &= \sum_{i}^{n} log\ P(y^{(i)} \mid x^{(i)}; \theta) \\ &= \sum_{i}^{n} log \frac{1}{\sigma\sqrt{2\pi}} exp^{-\frac{(y^{(i)} - \theta^T x^{(i)})^2}{2\sigma^2}} \\ &= \sum_{i}^{n} log \frac{1}{\sigma\sqrt{2\pi}} + \sum_{i}^{n} log \left(exp^{-\frac{(y^{(i)} - \theta^T x^{(i)})^2}{2\sigma^2}} \right) \\ &= n\ log \frac{1}{\sigma\sqrt{2\pi}} - \frac{1}{2\sigma^2} \sum_{i=1}^{n} (y^{(i)} - \theta^T x^{(i)})^2 \\ &= k_1 - k_2 \sum_{i=1}^{n} (y^{(i)} - \theta^T x^{(i)})^2 \end{aligned} \tag{1.11.2}$$

We can state that minimizing the negative log-likelihood is equivalent to maximizing the likelihood estimation since

$$\arg \max_x (x) = \arg \min_x (-x)$$

Maximizing Eq. 1.11.2 implies that we need to minimize the mean- squared error between the observed y and the predicted $\hat{y}$; therefore, minimizing the negative log-likelihood of our data with respect to θ is also the same as minimizing the mean squared error.

In **Logistic Regression**, we define $\phi = \frac{1}{1+exp^{-\theta^T x}}$, and the negative log-likelihood can be written as

$$\begin{aligned} -log\ P(y \mid x; \theta) &= -log \prod_{i=1}^{n} (\phi^{(i)})^{y^{(i)}} (1-\phi^{(i)})^{(1-y^{(i)})} \\ &= -\sum_{i=1}^{n} log((\phi^{(i)})^{y^{(i)}} (1-\phi^{(i)})^{(1-y^{(i)})}) \\ &= -\sum_{i=1}^{n} y^{(i)} log(\phi^{(i)}) + (1-y^{(i)}) log(1-\phi^{(i)}) \end{aligned} \tag{1.11.3}$$

In Eq. 1.11.3, minimizing the negative log-likelihood of the data with respect to θ is the same as minimizing the binary log loss (binary cross-entropy, discussed in the following section) between the observed y values and the predicted probabilities thereof.

In a **Multinoulli Distribution**, the negative log-likelihood can be written as

$$\begin{aligned} -log\ P(y \mid x; \theta) &= -log \prod_{i=1}^{n} \prod_{k=1}^{K} \pi_k^{y_k} \\ &= \sum_{i=1}^{n} \sum_{k=1}^{K} y_k log(\pi_k) \end{aligned} \tag{1.11.4}$$

In Eq. 1.11.4 minimizing the negative log-likelihood of the data with respect to θ is the same as minimizing the multi-class log loss (categorical cross-entropy), between the observed y values and the predicted probabilities thereof.

1.11.3 Entropy

Entropy in heat engineering and classical thermodynamics, is a measure of "disorder" based on the second law of thermodynamics, which states that a system's entropy never decreases spontaneously. From the concept of thermodynamics, it is the (log of) number of microstates or microscopic configurations. Intuitively, if the particles inside a system have many possible positions to move around, then the system has high entropy, and if they stay rigid, then the system has low entropy.

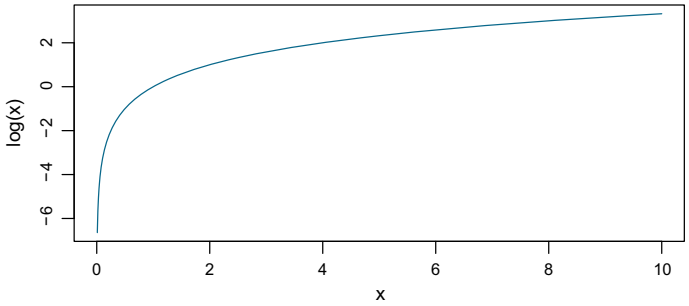

Fig. 1.8 The natural logarithm is a monotonically increasing function

In **Information theory**, entropy is a measure of uncertainty involved in making a prediction. Intuitively, we can describe entropy as how "surprised" would we be of the outcome, after we have made our initial prediction. If we consider an unfair coin that actually turns up a head 99% of the time it is tossed, we would be indeed very surprised if a particular toss turns up a tail. If we can average out the amount of surprise, the mean value of surprise is a measure for how uncertain we are. This measure of uncertainty is called entropy. Entropy therefore defines randomness; it is like describing how unpredictable something is.

If we consider a set of possible events with known probabilities of occurrence being $y_1; y_2; \ldots y_n$, as per [9] entropy is a means to find a measure of how much 'choice' is involved in the selection of the event or of how uncertain we are of the outcome. This measure, $H(y_1; y_2; \ldots y_n)$ would then be defined by the following properties

- H should be continuous in the y_i.
- If all the y_i are equal, $y_i = \frac{1}{n}$, then H should be a monotonic increasing function of n. The natural logarithm is a monotonically increasing function, implying that if the value on the x-axis increases, the corresponding value on the y-axis also increases (see Fig. 1.8).
- If a choice be broken down into two successive choices, the original H should be the weighted sum of the individual values of H.

The only H satisfying the three above assumptions is of the form

$$H = \sum_{i=1}^{n} y_i log(y_i) \tag{1.11.5}$$

Interpretation of Entropy

If we consider two events with probabilities p and 1-p the entropy H can be written as

$$H = -[p\ log(p) + (1-p)\ log(1-p)] \tag{1.11.6}$$

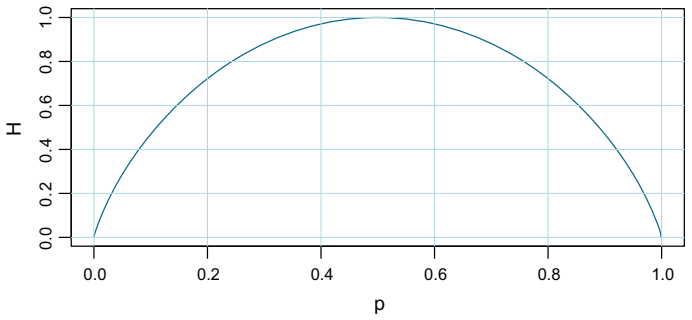

Fig. 1.9 Plot of Entropy with probabilities p and $1 - p$

Entropy can be easily interpreted from Fig. 1.9 as

- When $p = 1$, $H = 0$ implies that we are certain of the outcome and if we are not certain of the outcome, H is positive. Intuitively when $p = 0$, an event never occurs, it cannot contribute to the entropy, as it is never expected to occur and therefore $H = 0$.
- H is maximum when all the p_i are equal, i.e., $\frac{1}{n}$. This is the most uncertain situation.

Entropy is the weighted average of the log probability of the possible events, which measures the uncertainty present in their probability distribution. The higher the entropy, the less certain are we about the value.

1.11.4 Cross-Entropy

Cross-entropy (CE) loss, or log-loss, measures the performance of a classification model whose output has a probability value between 0 and 1. As the predicted probability diverges from the actual value the CE loss consequently increases. Therefore say, predicting a probability of 0.021 when the true observed value is 1 would result in a high CE loss. A perfect model, therefore, should have a CE loss of 0.

How close is the predicted probability distribution to the true distribution, i.e., to find the cross-entropy loss, we use

$$H(y, \hat{y}) = \sum_i y_i log \frac{1}{\hat{y}_i} = -\sum_i y_i log(\hat{y}_i) \tag{1.11.7}$$

where, $\sum_i y_i$ is the true probability distribution and $\sum_i \hat{y}_i$ is the computed probability distribution

Let us go through an example where we have three training items with the following computed outputs and target outputs

Table 1.1 Target values and computed probabilities of a neural network

	Class A	Class B	Class C
Target	0.000	1.000	0.000
Computed probability	0.128	0.719	0.153

For the data in Table 1.1, we can calculate the cross-entropy, using Eq. 1.11.7 as equal to 0.475, and it gives us a measure of how far away is our prediction from the true distribution.

```
H = -(0.0 * log2(0.128) + 1.0 * log2(0.719) + 0.0 * log2(0.153))
H
```

```
[1] 0.4759363
```

In **binary classification**, where the number of output class labels are *two*, CE is calculated as

$$CE_{Loss} = H(y, \hat{y}) = -(y\ log(\hat{y}) + (1-y) log(1-\hat{y})) \quad \textit{—for binary classification} \tag{1.11.8}$$

In **multi-class classification**, where the number of output class labels are more than two, i.e., *n*-labels, CE is calculated as

$$CE_{Loss} = H(y, \hat{y}) = -\sum_{1}^{n} y_i\ log(\hat{y}_i) \quad \textit{—for multi-class classification} \tag{1.11.9}$$

1.11.5 Kullback–Leibler Divergence

Kullback–Leibler[1] Divergence (KL Divergence) from $\hat{y}$ to y is the difference between CE (Eq. 1.11.7) and entropy (Eq. 1.11.5). It quantifies the additional uncertainty in y introduced by using $\hat{y}$ to approximate y

$$\begin{aligned} KL(y \mid\mid \hat{y}) &= \sum_i y_i\ log\frac{1}{\hat{y}_i} - \sum_i y_i\ log\frac{1}{y_i} \\ &= H(y, \hat{y}) - H(y) \\ &= \sum_i y_i log\frac{y_i}{\hat{y}_i} \end{aligned} \tag{1.11.10}$$

[1] Named after Solomon Kullback and Richard Leibler in 1951.

In information theory, the *KL* Divergence measures the number of bits required on average to encode symbols from y according to $\hat{y}$. The *KL* Divergence is never negative, and it is zero only when y and $\hat{y}$ are the same. Minimizing CE is the same as minimizing the *KL* Divergence from $\hat{y}$ to y.

1.11.6 Summarizing the Measurement of Loss

Let us consider the empirical true distribution to be p and, the predicted distribution (the model we are trying to optimize) to be q.

From the above discussions, we can state that **KL divergence** allows us to measure the difference between two probability distributions.

- The **entropy**, $H(p)$ of a distribution p, gives us an estimate of the uncertainty present in the distribution or, how certain can we be of the outcome.
- The **Cross-Entropy** $H(p, q)$ between two distributions p and q, quantifies the difference between the two probability distributions; i.e., how close is the predicted distribution to the true distribution. In machine learning classification problems, the Cross-Entropy loss, i.e., log-loss, measures the Cross-Entropy between the empirical distribution of the labels (given the inputs) and the distribution predicted by the model. In binary classification, the cross-entropy is proportional to the **negative log- likelihood**, and therefore minimizing the negative log-likelihood is equivalent to maximizing the likelihood.
- The difference, i.e., $KL(p \mid\mid q)$, measures the average number of extra bits per message, whereas $H(p, q)$ measures the average number of total bits per message.
- If the empirical distribution p is fixed it would be equivalent to say that we are minimizing the *KL* divergence between the empirical distribution and the predicted distribution. As we can see in the expression above, the two are related by the additive term $H(p)$, i,e, the entropy of the empirical distribution. Because p is fixed, $H(p)$ does not change with the parameters of the model, and can be disregarded in the loss function. This may not be true where p may also vary.

1.12 Conclusion

We have touched upon the basic facets, which define a machine learning algorithm. **Machine learning** belongs to the domain of AI and it endeavors to develop *models* (statistical programs) from exposure to training data. The process of training a machine learning algorithm results in a model, and is therefore called a learning algorithm.

Deep learning is another subset of AI, where *models* represent geometric transformations over many different layers.

In both machine learning and deep learning, the real knowledge are the structural parameters, i.e., the weights and biases of the model. The common ground, therefore, is to discover the best set of parameters, which will define the best model.

Chapter 2
Introduction to Neural Networks

Just as electricity transformed almost everything 100 years ago, today I actually have a hard time thinking of an industry that I don't think AI (Artificial Intelligence) will transform in the next several years.

Andrew Ng

Abstract In this chapter, we will discuss the basic architecture of neural networks including activation functions, *forward propagation*, and *backpropagation*. We will also create a simple neural network model from scratch using the `sigmoid` activation function. In particular, this chapter will discuss:

- Neural network architecture.
- Activation functions used in neural networks.
- Forward propagation.
- Loss function of neural networks.
- Backpropagation.

2.1 Introduction

Structured data refers to any data that resides in a fixed field within a record or a file. **Unstructured data** is information, in many different forms, i.e., *e*-mail messages, text documents, videos, photographs, audio files, presentations, webpages, etc. Whereas humans are good at interpreting unstructured data, neural networks can deal with both structured and unstructured data.

The most basic neural network consists of an input layer, an hidden layer and an output layer as shown in Fig. 2.1. The inputs to the neural network consist of two independent variables $X = [x_1, x_2]$. The output will be determined by the nodes in the hidden layer, which compute the *activation function* of the *weighted input*. The output of the activation function from the hidden layer is transmitted to the output layer, which calculates the predicted output.

© Springer Nature Singapore Pte Ltd. 2019

A. Ghatak, *Deep Learning with R*,
https://doi.org/10.1007/978-981-13-5850-0_2

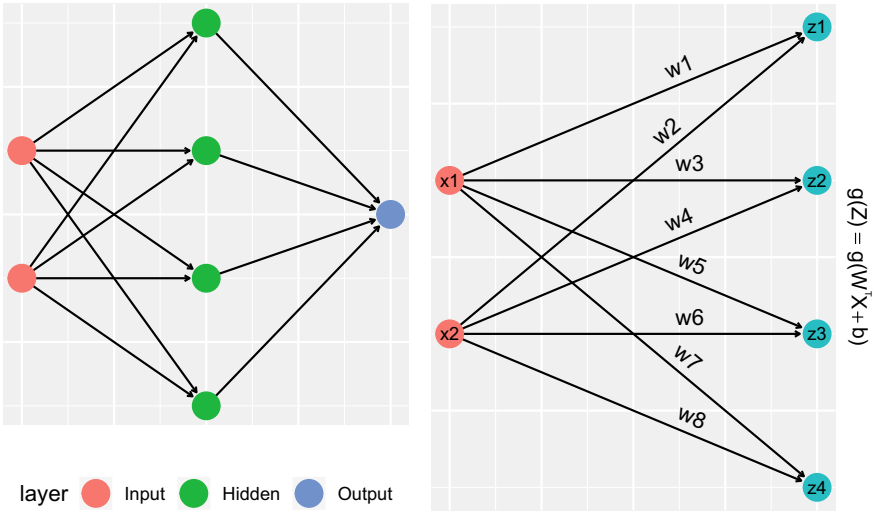

Fig. 2.1 Left: A simple neural network representation with an input layer, an hidden layer and an output layer. Right: The hidden layer calculates the summation of the input and the parametrized weights and the bias and calculates its activation function

The input layer of a neural network are the features of the input data. The features of the input data are defined by the number of nodes in the input layer. The output layer contains the predicted values. In between the input and output layers, there are hidden layers, through which the input data is propagated forward. The hidden layers also have nodes and the edges between the nodes are the weights (as shown in the right hand plot of Fig. 2.1).

Based on the predictions in the output node, the cost is computed with respect to the actual labels/values.

At each iteration, we adjust the weights to minimize the cost by a process known as backward propagation to calculate the gradients. This is achieved by calculating the partial derivatives of all the computations with respect to the output layer or the previous hidden layer. First, the partial derivatives of the weights of the hidden layer with respect to the output layer are calculated. The sequence of calculation of the partial derivatives then proceeds towards the input layer. The gradients obtained are then used to update the weights and we start the process (with forward propagation) all over again. With each pass, which is also called an **epoch**, we get closer to the optimal parameter weights.

2.2 Types of Neural Network Architectures

2.2.1 Feedforward Neural Networks (FFNNs)

These are the most commonly used types of neural network, where the first layer is the input layer and the last layer is the output layer and the in-between layers are the hidden layers. These networks compute a series of transformations between their input and their output resulting in a new representation of the input at each successive layer. This is achieved by a nonlinear transformation of the activities in the layer below.

These are therefore multilayered classifiers having many layers of weights separated by nonlinearities (sigmoid, tanh, rectified linear units (ReLU), scaled exponential linear units (selu), softmax). They are also known as Multilayered Perceptrons. FFNNs can be used for classification and unsupervised feature learning as autoencoders.

2.2.2 Convolutional Neural Networks (ConvNets)

Almost any state of the art vision based deep learning result in the world today has been achieved using Convolutional Neural Networks.[1] ConvNets are composed of convolutional layers which act as hierarchical feature extractors. They are mostly used for image classification, text classification, object detection, and image segmentation.

2.2.3 Recurrent Neural Networks (RNNs)

This is a much more interesting kind architecture in which information can flow round in cycles. RNNs model sequences by applying the same set of weights recursively on the aggregator state at a time t and, input at a time t (if the sequence has inputs at times $0 \ldots t \ldots T$, and has a hidden state at each time t which is the output from t-1 step of the RNN).

They are deep networks having one hidden layer per time slice. They use the same weights at every time slice and receive an input at every time slice as shown in Fig. 2.2. These networks can store past information for a long time and are much more difficult to train. However, a lot of progress has been made in training recurrent neural networks, and they can now do some fairly impressive things.

The counterparts of RNN are Long Short-Term Memory Networks or LSTMs (capable of learning long-term dependencies) and Gated Recurrent Networks or

[1] CNNs are attributed to its inventor, Yann LeCun.

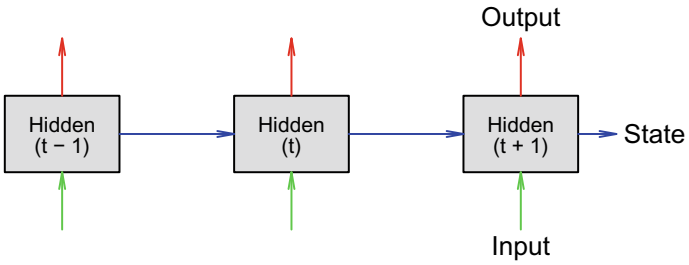

Fig. 2.2 A simple RNN architecture. They have distributed hidden states which allow them to efficiently store past information, indicated by the time step 't'

GRUs (they aim to solve the vanishing gradient problem; vanishing gradients is discussed in Sect. 5.4). Both are state-of-the-art networks in most sequence modeling tasks. RNNs may be used for any sequence modeling task especially text classification, machine translation, language modeling, etc.

Let us have a look at the different components of a simple feedforward neural network.

2.3 Forward Propagation

A neural network's input could be anything. They could, for example, be the grayscale intensity (between 0 and 1) of a 20 pixel by 20 pixel image that represents a bunch of handwritten digits. In this case, we would have 400 input units (features).

Figure 2.3 depicts a typical neural network flow diagram with an input layer, two hidden layers, and an output layer.

Figure 2.4 is a representation of a hidden layer having as input (in this case) the input layer.

As stated in Sect. 1.1.2, forward propagation computes the predicted values by successively transforming the data through the layers upto the output layer, where the output of each layer is the input for the next successive layer.

A loss function compares the predicted values from the output layer to the actual values and computes a distance score between these values, thereby capturing how well the network has done on a batch of input data.

2.3.1 Notations

In neural networks, the notation representation which we will follow is as follows:

1. For inputs $x \in \mathbb{R}^{n_x}$ (n features):

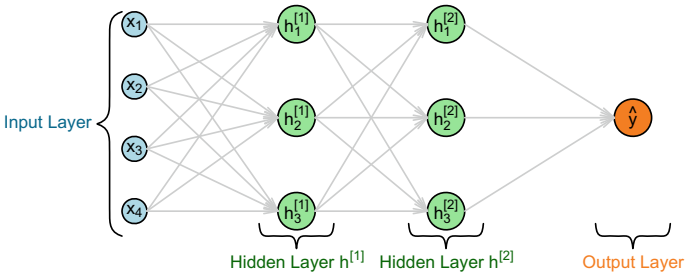

Fig. 2.3 A representation of a neural network with four input features, two hidden layers with three nodes each, and an output layer

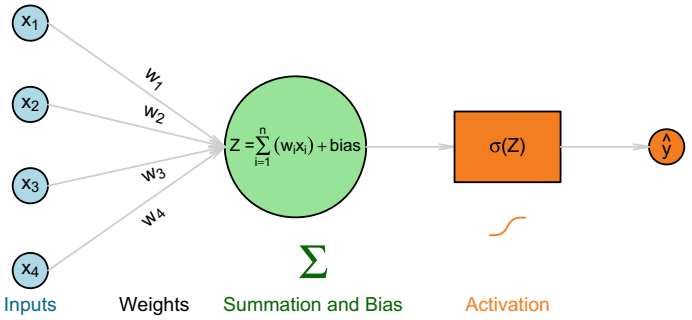

Fig. 2.4 The hidden layer carries out a geometric transformation function of the data which goes through that layer

- superscript (i): is a reference to the ith observation; i.e., $x^{(1)}$ is the input value of the first observation.
- subscript n: is a reference to the nth feature number; i.e., $x_2^{(1)}$ is the first observation of the second feature.

2. For network layers:

- superscript $[l]$: refers to layer l; i.e., $z^{[1]}$ is the linear function at layer 1.
- subscript n: is a reference to the node number; i.e., $z_1^{[1]}$ is the linear function consisting of the weighted inputs plus the bias, at the first node in layer 1.

2.3.2 Input Matrix

Let us consider a set of m linear equation with n features:

$$y^{(i)} = w_0 + w_1 x_1^{(i)} + w_2 x_2^{(i)} + \cdots + w_n x_n^{(i)} + \epsilon_i^{(i)} \text{ for } i = 1, \ldots, m \qquad (2.3.1)$$

We can represent Eq. 2.3.1 in matrix notation as

$$\begin{aligned}\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix} &= \begin{bmatrix} 1 & x_1^{(1)} & x_2^{(1)} & \cdots & x_n^{(1)} \\ 1 & x_1^{(2)} & x_2^{(2)} & \cdots & x_n^{(2)} \\ \vdots & & & & \\ 1 & x_1^{(m)} & x_2^{(m)} & \cdots & x_n^{(m)} \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_m \end{bmatrix} + \begin{bmatrix} \epsilon_0 \\ \epsilon_1 \\ \vdots \\ \epsilon_m \end{bmatrix} \\ &= \begin{bmatrix} w_0 & w_1 & w_2 & \cdots & w_m \end{bmatrix} \begin{bmatrix} 1 & 1 & \cdots & 1 \\ x_1^{(1)} & x_1^{(2)} & \cdots & x_1^{(m)} \\ x_2^{(1)} & x_2^{(2)} & \cdots & x_2^{(m)} \\ \vdots & \vdots & \cdots & \vdots \\ x_n^{(1)} & x_n^{(2)} & \cdots & x_n^{(m)} \end{bmatrix} + \begin{bmatrix} \epsilon_0 \\ \epsilon_1 \\ \vdots \\ \epsilon_m \end{bmatrix}\end{aligned} \tag{2.3.2}$$

If we ignore the error terms ϵ, we write Eq. 2.3.2 as

$$\boldsymbol{Y} = \boldsymbol{W}^T \boldsymbol{X} \tag{2.3.3}$$

The dimensions of the input matrix in Eq. 2.3.3 are *(num features, num observations)*. In Fig. 2.3, there are four features in the input matrix **X**. Let us assume that there are only two observations, in which case the dimensions of the input matrix will be (4, 2). In matrix form, we can represent our input as:

$$\boldsymbol{X} = \begin{bmatrix} \boldsymbol{X}_1 & \boldsymbol{X}_2 \end{bmatrix} = \begin{bmatrix} x_1^{(1)} & x_1^{(2)} \\ x_2^{(1)} & x_2^{(2)} \\ x_3^{(1)} & x_3^{(2)} \\ x_4^{(1)} & x_4^{(2)} \end{bmatrix} \tag{2.3.4}$$

2.3.3 Bias Matrix

The dimensions of the bias matrix $b^{[1]}$ are *(number of nodes in $l^{[1]}$,1)*. For our input matrix **X**, the dimensions of the bias matrix are therefore (3, 1), as there are 3 nodes in hidden layer 1. In matrix form, we can represent the bias matrix as:

$$b^{[1]} = \begin{bmatrix} r \\ s \\ t \end{bmatrix} \tag{2.3.5}$$

The dimensions of the bias matrix $b^{[2]}$ are *(number of nodes in $l^{[2]}$, 1)* and can be written as:

$$b^{[2]} = \begin{bmatrix} u \\ v \\ w \end{bmatrix} \tag{2.3.6}$$

2.3.4 Weight Matrix of Layer-1

For a data set with m observations and n input features if we need w_n weights to perform a dot product for a single observation; with $n^{[\ell]}$ nodes in the ℓth layer, we need $n \times w_n^{[\ell]}$ number of weights. Referring to Fig. 2.3, there are four input features and three nodes in the first hidden layer. Therefore, we have 12 weights for each observation.

For the first hidden layer, the dot product for an input data set $\boldsymbol{X} = \boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_m$ (m observations) having a corresponding set of weights $\boldsymbol{W}^{[1]}$ is

$$Z^{[1]} = \sum_{i=1}^{m} \boldsymbol{w}_i^{[1]T} \boldsymbol{x}_i \; + \; bias^{[1]}$$

In our example, we have considered two observations each having four features. Therefore, we need four sets of weights for each observation, at each node.

For two observations, having four features each and considering three nodes, we require 24 weights at hidden layer 1.

If we denote the weights by its (feature number, node number), w_{11} will represent the weight of the first feature at node number 1.

The weights matrix can be represented as

$$\boldsymbol{W}^{[1]} = \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \\ w_{41} & w_{42} & w_{43} \end{bmatrix}$$

and the dot product represented as

$$\boldsymbol{W}^{[1]T} \cdot \boldsymbol{X} = \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \\ w_{41} & w_{42} & w_{43} \end{bmatrix} \begin{bmatrix} x_1^{(1)} & x_1^{(2)} \\ x_2^{(1)} & x_2^{(2)} \\ x_3^{(1)} & x_3^{(2)} \\ x_4^{(1)} & x_4^{(2)} \end{bmatrix} \tag{2.3.7}$$

If there are $n^{[1]}$ nodes in the first hidden layer, the dimensions of the first hidden layer weight matrix will be ($features$, $n^{[1]}$). In our example case, the first hidden layer has 3 nodes and there are 4 input features (they can also be thought of as the input layer nodes), and hence our weight matrix $\boldsymbol{W}^{[1]}$ has the dimension (4, 3).

To generalize, the weight matrix for subsequent hidden layers ℓ having $n^{[\ell]}$ nodes will have the dimension ($n^{[\ell-1]}$, $n^{[\ell]}$).

2.3.5 Activation Function at Layer-1

After adding the bias term, Eq. 2.3.7 above can be generalized as

$$\boldsymbol{Z}^{[\ell]} = \boldsymbol{W}^{[\ell]T}\boldsymbol{A}^{[\ell-1]} + b^{[\ell]} \tag{2.3.8}$$

The activation at layer-1 can be written as

$$\boldsymbol{Z}^{[1]} = \boldsymbol{W}^{[1]T}\boldsymbol{X} + b^{[1]} \tag{2.3.9}$$

where, $\boldsymbol{X}$ is the input to the first hidden layer and, $Z^{[\ell-1]}$ is the input to subsequent hidden layers.

$\boldsymbol{Z}^{[1]}$ has the dimension ($n^{[1]}$, *num observations*). In our example case, the dimension of $\boldsymbol{Z}^{[1]}$ is (3, 2).

Equation 2.3.9 can be expanded to the matrix form as

$$\begin{aligned}\boldsymbol{Z}^{[1]} &= \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \\ w_{41} & w_{42} & w_{43} \end{bmatrix} \begin{bmatrix} x_1^{(1)} & x_1^{(2)} \\ x_2^{(1)} & x_2^{(2)} \\ x_3^{(1)} & x_3^{(2)} \\ x_4^{(1)} & x_4^{(2)} \end{bmatrix} + \begin{bmatrix} r \\ s \\ t \end{bmatrix} \\ &= \begin{bmatrix} z_{1,1}^{[1]} & z_{1,2}^{[1]} \\ z_{2,1}^{[1]} & z_{2,2}^{[1]} \\ z_{3,1}^{[1]} & z_{3,2}^{[1]} \end{bmatrix}\end{aligned} \tag{2.3.10}$$

The activation function $\boldsymbol{A}^{[1]}$ has the same dimension as $\boldsymbol{Z}^{[1]}$ and is written as

$$\boldsymbol{A}^{[1]} = \begin{bmatrix} g(z_{1,1}^{[1]}) & g(z_{1,2}^{[1]}) \\ g(z_{2,1}^{[1]}) & g(z_{2,2}^{[1]}) \\ g(z_{3,1}^{[1]}) & g(z_{3,2}^{[1]}) \end{bmatrix} \tag{2.3.11}$$

where g is an activation function (i.e., `sigmoid`, etc.)

For our example case, the first row of $A^{[1]}$ belongs to *node*-1, the second row belongs to *node*-2, and the third row belongs to *node*-3 of layer-1, respectively.

The activations in the first hidden layer nodes is depicted in Fig. 2.5.

2.3.6 Weights Matrix of Layer-2

The output from the first hidden layer ($\boldsymbol{A}^{[1]}$) are the inputs to the second hidden layer.

For our example case, these are the activations from the three nodes in the first hidden layer. The second hidden layer also has three nodes. We therefore need 9 weights for the second hidden layer.

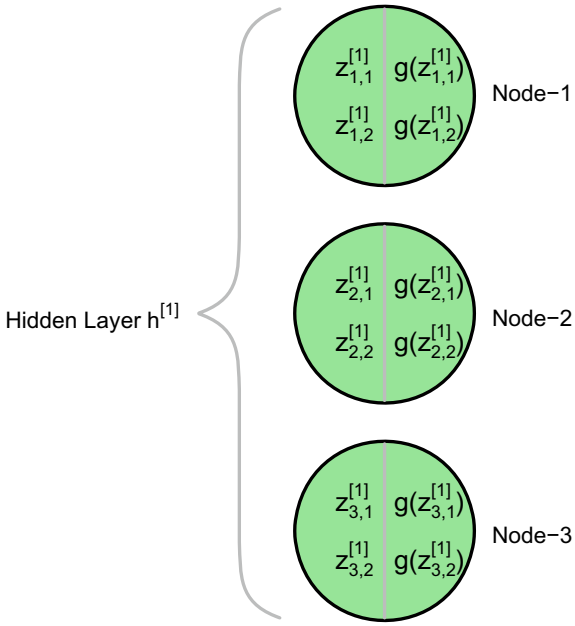

Fig. 2.5 Input and activations represented in the nodes of hidden layer 1

You may also recall that the dimensions of the weight matrix for subsequent hidden layers ℓ (other than the first hidden layer), having $n^{[\ell]}$ nodes are $(n^{[\ell-1]}, n^{[\ell]})$. In our example case, the dimensions of the weight matrix will be (3, 3) and we can represent the weights matrix as follows:

$$\boldsymbol{W}^{[2]} = \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \end{bmatrix} \tag{2.3.12}$$

The dot product for the second hidden layer can be represented as

$$\begin{aligned} \boldsymbol{Z}^{[2]} &= \boldsymbol{W}^{[2]T}\boldsymbol{A}^{[1]} + b^{[2]} \\ &= \begin{bmatrix} w_{11} & w_{21} & w_{31} \\ w_{12} & w_{22} & w_{32} \\ w_{13} & w_{23} & w_{33} \end{bmatrix} \begin{bmatrix} g(z_{1,1}^{[1]}) & g(z_{1,2}^{[1]}) \\ g(z_{2,1}^{[1]}) & g(z_{2,2}^{[1]}) \\ g(z_{3,1}^{[1]}) & g(z_{3,2}^{[1]}) \end{bmatrix} + bias^{[2]} \end{aligned} \tag{2.3.13}$$

The dimensions of the $\boldsymbol{Z}^{[2]}$ matrix will be $(n^{[l]}, num\ observations)$. In our example case, $n^{[2]} = 3$ and the number of observations remains the same, i.e., 2. Therefore, the dimensions of our $\boldsymbol{Z}^{[2]}$ matrix is (3, 2), which is evident from Eq. 2.3.13.

2.3.7 Activation Function at Layer-2

From the above discussion, we can write

$$\boldsymbol{A}^{[2]} = \begin{bmatrix} g(z_{1,1}^{[2]}) & g(z_{1,2}^{[2]}) \\ g(z_{2,1}^{[2]}) & g(z_{2,2}^{[2]}) \\ g(z_{3,1}^{[2]}) & g(z_{3,2}^{[2]}) \end{bmatrix} \tag{2.3.14}$$

Each row of $A^{[2]}$ corresponds to a node of the hidden layer. For our example case, the first row of $A^{[2]}$ belongs to *node*-1, the second row to *node*-2, and the third row to *node*-3 of the second hidden layer.

The activation function $\boldsymbol{A}^{[2]}$ has the same dimension as $\boldsymbol{Z}^{[2]}$ and is written as

$$\boldsymbol{A}^{[2]} = \begin{bmatrix} g(z_{1,1}^{[2]}) & g(z_{1,2}^{[2]}) \\ g(z_{2,1}^{[2]}) & g(z_{2,2}^{[2]}) \\ g(z_{3,1}^{[2]}) & g(z_{3,2}^{[2]}) \end{bmatrix} \tag{2.3.15}$$

Please note that the activation functions at hidden layer 2 could be different from that at hidden layer 1, i.e., they may not be the same.

The layer-2 nodes is depicted in Fig. 2.6.

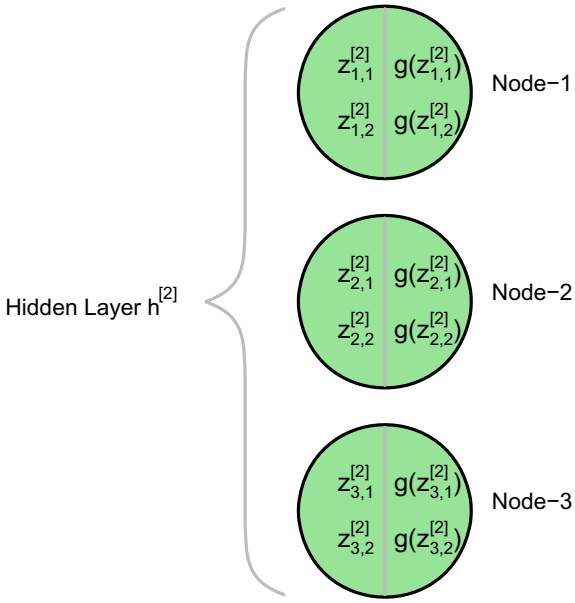

Fig. 2.6 Input and activations represented in the nodes of hidden layer 2

2.3.8 *Output Layer*

Suppose, we are considering a binary classification example (say, using the `sigmoid` activation in the output layer). Then, the input to the output layer will have a dimension of $\boldsymbol{A}^{[2]}$, i.e., (3, 2). Each column of this matrix represents an observation. The `sigmoid` activation function maps the output from $\boldsymbol{A}^{[2]}$ to values between 0 and 1.

The label for each observation is selected based on the probability threshold (in most cases this could be 0.5, though this value may be altered, especially where we are interested to reduce the False Positives or the False Negatives). The first column may represent a "Yes" and the second column a "No" or vice versa.

If any of the values in either of the columns exceeds the defined probability threshold, that observation (represented by the respective column) is either a "Yes" prediction or otherwise.

Let us assume that $\boldsymbol{A}^{[2]}$ outputs the following matrix:

$$\begin{bmatrix} 0.7 & 0.3 \\ 0.4 & 0.6 \\ 0.1 & 0.5 \end{bmatrix}$$

and the `sigmoid` function at the output layer gives the following matrix:

$$\begin{bmatrix} \mathbf{0.66} & 0.57 \\ 0.59 & \mathbf{0.64} \\ 0.52 & 0.62 \end{bmatrix}$$

Considering that the first column represents a 1 ("Yes") and the second column represents a 0 ("No"), then our network outputs a 0.66% probability that the first observation is 1 and 0.64% probability that the second observation is 0, (assuming a probability value > 0.5 is 1 and, ≤ 0.5 is 0).

It may also be noted that the `sigmoid` output does not represent a probability distribution, i.e., the sum of the probabilities is not unity.

This is illustrated by the following `define_sigmoid` function below:

```
define_sigmoid <- function(Z){
  output = 1/(1 + exp(-Z))

  return(output)
}

output_matrix <- define_sigmoid(matrix(c(0.7, 0.4, 0.1, 0.3, 0.6, 0.5), nrow = 3))
output_matrix
```

```
          [,1]      [,2]
[1,] 0.6681878 0.5744425
[2,] 0.5986877 0.6456563
[3,] 0.5249792 0.6224593
```

```
sum(output_matrix)
```

```
[1] 3.634413
```

Table 2.1 Summary of dimensions of the components in a neural network

	Dim W	Dim b	Activation	Dim Activation
Layer 1	$(num\ features, n^{[1]})$	$(n^{[1]}, 1)$	$Z^{[1]} = W^{[1]}X + b^{[1]}$	$(n^{[1]}, num\ obs)$
Layer ℓ	$(n^{[\ell-1]}, n^{[\ell]})$	$(n^{[\ell]}, 1)$	$Z^{[\ell]} = W^{[\ell]}A^{[\ell-1]} + b^{[\ell]}$	$(n^{[\ell]}, num\ obs)$
Layer L	$(n^{[L-1]}, n^{[L]})$	$(n^{[L]}, 1)$	$Z^{[L]} = W^{[L]}A^{[L-1]} + b^{[L]}$	$(n^{[L]}, num\ obs)$

Please note that the output in this case is not a probability distribution (i.e., `sigmoid` activation function does not sum to 1, unlike the `softmax` activation function, which we will discuss in detail in a later section).

2.3.9 Summary of Forward Propagation

The forward propagation equations for a three-layer network for a single observation can be represented as

$$\begin{aligned} z^{[1](i)} &= w^{[1]}x^{(i)} + b^{[1](i)} \\ a^{[1](i)} &= g^{[1]}(z^{[1](i)}) \\ z^{[2](i)} &= w^{[2]}a^{[1](i)} + b^{[2](i)} \\ a^{[2](i)} &= g^{[2]}(z^{[2](i)}) \\ z^{[3](i)} &= w^{[3]}a^{[2](i)} + b^{[3](i)} \\ a^{[3](i)} &= g^{[3]}(z^{[3](i)}) \end{aligned} \tag{2.3.16}$$

The vectorized form for Eq. 2.3.16 is

$$\begin{aligned} &\text{for } \ell \text{in} 1, 2, 3, \ldots, \text{L} \\ Z^{[\ell]} &= W^{[\ell]}A^{[\ell]} + b^{[\ell]} \\ A^{[\ell]} &= g^{[\ell]}(Z^{[\ell]}) \end{aligned} \tag{2.3.17}$$

The dimensions at different layers for the weight matrix, bias, and activation matrix are summarized in Table 2.1.

2.4 Activation Functions

An activation function is a mathematical function which transforms the input data from the previous layer into a meaningful representation, which is closer to the expected output. Figure 2.7 depicts sigmoid activations in the two hidden layers and as well in the output layer.

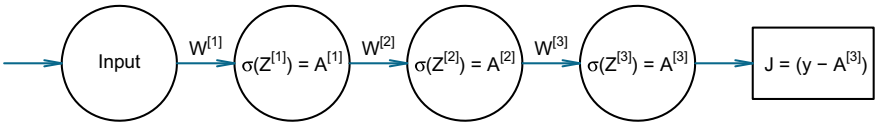

Fig. 2.7 A two hidden layer neural network with sigmoid activation. The output is the activation $A^{[3]}$ and J is the cost computed for a single epoch of the computation

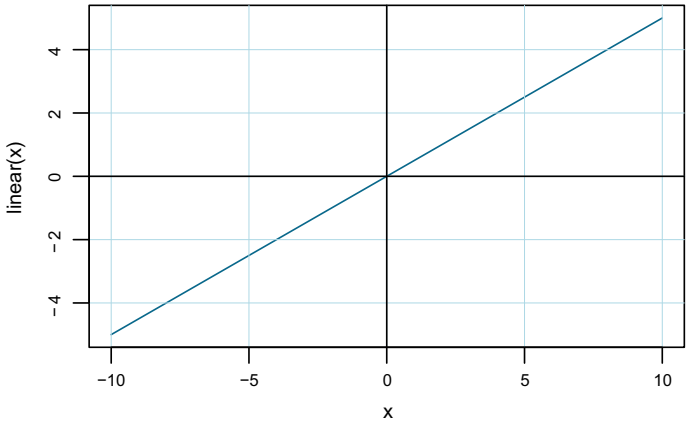

Fig. 2.8 A linear function

Activation Functions can be

- Linear Activation Function
- Nonlinear Activation Functions

Linear Activation Function

The linear activation function is a straight line (linear) and the output of these functions will range from $-\infty$ to $+\infty$. Linear functions do not help with the complexity of data and the parameters present in neural networks. With linear activation functions, we can stack up as many hidden layers in the neural network, and the final output will still be a linear combination of the input data (Fig. 2.8).

Nonlinear Activation Function

Most real-world problems are nonlinear in nature and therefore we resort to a nonlinear mapping (function) called activation functions in deep learning. A neural network must be able to take any input from $-\infty$ to $+\infty$, and map it to an output that ranges between [0, 1] or [−1, 1], etc. Nonlinear activation functions are used in the hidden layers of neural networks and are only segregated on the basis of their range or the degree of nonlinearity. The output layers use either the `sigmoid` or `softmax` activation, depending on the classification task, i.e., binary classification or multi-label classification.

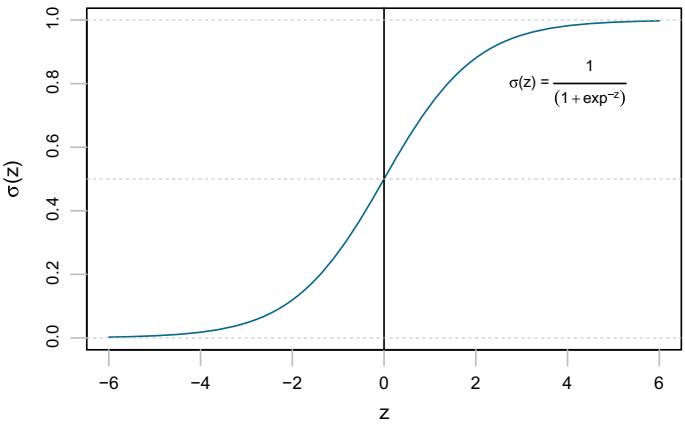

Fig. 2.9 The sigmoid function

2.4.1 Sigmoid

The `sigmoid` function is widely used in machine learning for binary classification in logistic regression and neural network implementations for binary classification. The `sigmoid` activation of the linear function Z is represented as

$$\sigma(Z) = \frac{1}{1 + exp^{-Z}} \tag{2.4.1}$$

As we have seen earlier, the `sigmoid` function maps the linear input to a nonlinear output.

From Fig. 2.9 above, it can seen that the `sigmoid` function takes in values between $-\infty$ and $+\infty$ and outputs a value between 0 and 1; the nature of the output values is a useful representation of the predicted probability of an binary output. The `sigmoid` function is a monotonically increasing function. However, the `sigmoid` function has the propensity to get *"stuck"*, i.e., the output values would be very near 0 when the input values are strongly negative and vice versa.

Since the parameter gradients are calculated on the activations, this can result in model parameters getting updated less regularly than desired, and therefore getting *"stuck"* in their current state, during training.

> An undesirable property of the `sigmoid` activation is that it saturates at either tail with a value of 0 or 1. When this happens, the gradient at these regions is almost near to zero. During backpropagation (discussed later), the local gradient will be multiplied to the gradient of `sigmoid` activation function and, if the local gradient is very small, it will effectively make the gradient "vanish" and, no signal will flow through the neuron to its weights.

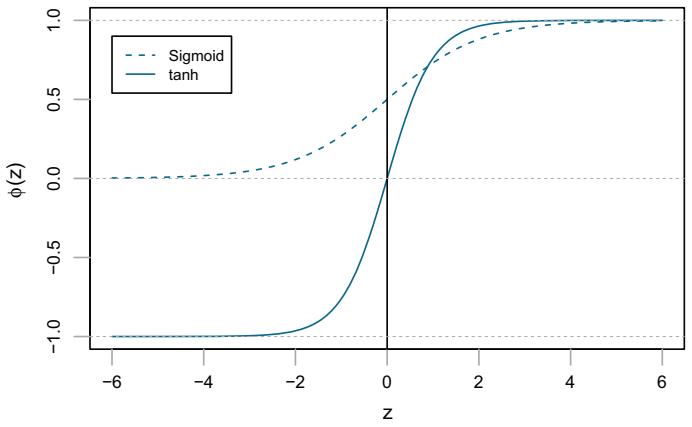

Fig. 2.10 tanh versus Sigmoid Function

2.4.2 Hyperbolic Tangent

Like the `sigmoid` function, the `tanh` function is also *sigmoidal* ("S"-shaped), but outputs values between -1 and $+1$, as shown in Fig. 2.10. Large negative inputs to the `tanh` function will give negative outputs. Further, only zero-valued inputs are mapped to near-zero outputs. These properties does make the network less likely to get *"stuck"*, unlike the `sigmoid` activation.

The `tanh` activation of Z can be represented by

$$g(Z) = \frac{exp^{Z} - exp^{-Z}}{exp^{Z} + exp^{-Z}} \tag{2.4.2}$$

The gradient of `tanh` is stronger than `sigmoid` (i.e., the derivatives are steeper). Like `sigmoid`, `tanh` activation also has the vanishing gradient problem.

2.4.3 Rectified Linear Unit

The rectified linear unit, `relu` is the most used activation function for the hidden layers. The `relu` function is half rectified, as shown in Fig. 2.11. A rectified linear unit has an output of 0 if the input is less than 0, and if the input is greater than 0, the output is the same as the input. The range of the `relu` activation function is therefore between 0 and $+\infty$ and can be represented by

$$g(Z) = max(Z, 0) \tag{2.4.3}$$

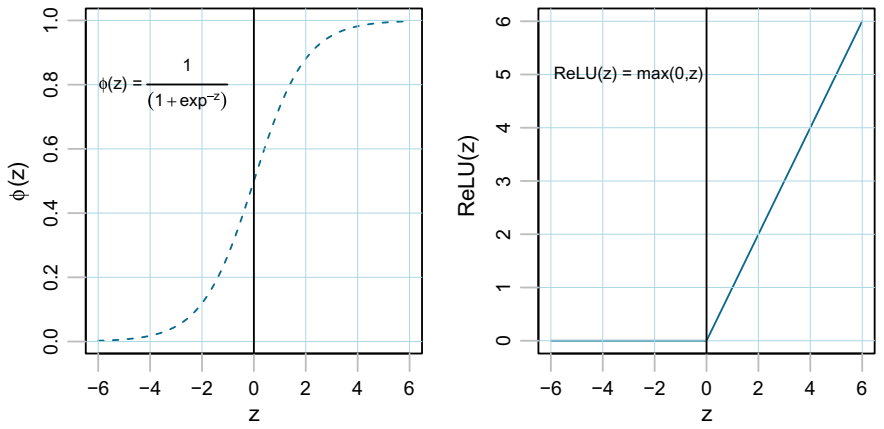

Fig. 2.11 Sigmoid Function versus ReLU

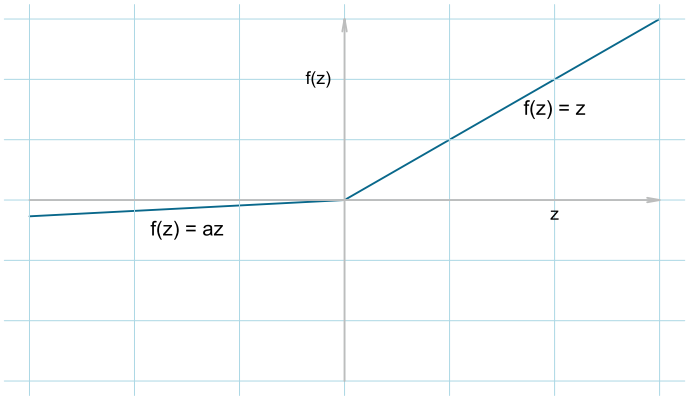

Fig. 2.12 leakyRelu function

`relu` often results in much faster training for large networks [12]. Most frameworks like `TensorFlow` and `TFLearn` make it simple to use `relu` on the hidden layers. However the issue is, all the negative values become zero, which in turn affects the resulting graph by not mapping the negative values appropriately; further, it also decreases the ability of the model to fit or train from the data properly. The `leakyrelu` tries to rectify this drawback.

2.4.4 Leaky Rectified Linear Unit

The leaky rectified linear unit, `lrelu` is represented in Fig. 2.12.

The range of the `lrelu` is $-\infty$ to $+\infty$. The `lrelu` can be written as

Table 2.2 Weight distribution for different labels as an example case

	a	b	c
x_1	0.6	0.4	0.2
x_2	0.2	0.7	0.4
x_3	0.5	0.2	0.1
x_4	0.4	0.1	0.7

$$g(Z) = \begin{cases} Z & \text{if } Z > 0 \\ aZ & \text{otherwise} \end{cases} \tag{2.4.4}$$

The "leak' increases the range of the `relu` function. In Fig. 2.12, a is normally assigned a value of 0.01. When a is not 0.01, this function is called **Randomized relu**.

2.4.5 Softmax

`Sigmoid` function can only handle two classes for classification problems. For multi-class classification, `softmax` is the preferred function. The `softmax` function calculates the probabilities of each target class over all possible target classes. The output range of the `softmax` activation function is between 0 and 1 and the sum of all the probabilities is equal to 1.

It computes the exponential of the given input value and the sum of exponential values of all the input values. The ratio of the exponential of the input value and the sum of exponential values is the output of the `softmax` function.

Let us derive the `softmax` function from ground up.

Suppose we have an input vector $x = [x_1, x_2, x_3, x_4]$ having 3 labels a, b, c. Our objective is to predict the probability of producing each output conditional on the input, i.e., $P(a \mid \boldsymbol{x})$, $P(b \mid \boldsymbol{x})$, $P(c \mid \boldsymbol{x})$, respectively. The weights for our model are as depicted in Table 2.2.

The unnormalized probabilities can be calculated by the summation of weights for each label. Let us assume they are

$$z_a = \sum_{i=1}^{3} w_{i,a} x_i = 0.71$$

$$z_b = \sum_{i=1}^{3} w_{i,b} x_i = 0.7$$

$$z_c = \sum_{i=1}^{3} w_{i,c} x_i = 0.44$$

We convert the unnormalized probabilities to normalized probabilities

$$P(a \mid x) = \frac{0.71}{(0.71 + 0.7 + 0.44)} = 0.383$$

$$P(b \mid x) = \frac{0.7}{(0.71 + 0.7 + 0.44)} = 0.378$$

$$P(c \mid x) = \frac{0.44}{(0.71 + 0.7 + 0.44)} = 0.237$$

For a valid probability distribution, all numbers must sum to 1:

$$0.383 + 0.378 + 0.237 \equiv 1.0$$

We are aware that a probability lies within the range [0, 1].

Now let us consider a case where due to some negative values in the weights, we get a negative value for the unnormalized probability, i.e., $z_a = -0.71$. In this situation, we cannot have a valid probability distribution because, $P(a \mid x) = -0.383$ and this is outside the range [0, 1].

To ensure that all unnormalized probabilities are positive, we must transform the input to a strictly positive real number and we choose the Euler's number *exp* to carry out the transformation

$$z_a = exp^{-0.71} = 0.4916$$

$$z_b = exp^{0.7} = 2.0137$$

$$z_c = exp^{0.44} = 1.5527$$

Recalculating for the normalized probabilities, we get

$$P(a \mid x) = \frac{0.4916}{(0.4916 + 2.0137 + 1.5527)} = 0.1211$$

$$P(b \mid x) = \frac{2.0137}{(0.4916 + 2.0137 + 1.5527)} = 0.4962$$

$$P(c \mid x) = \frac{1.5527}{(0.4916 + 2.0137 + 1.5527)} = 0.3826$$

And the sum of the probabilities is

$$0.1211 + 0.4962 + 0.3826 \equiv 1.0$$

To generalize our function, we can write

$$P(y \mid x) = \frac{exp^{z_j}}{\sum_{k=1}^{n} exp^{z_k}} \quad \forall j \in 1, \ldots n \tag{2.4.5}$$

Intuitively, the `softmax` function is a "soft" version of the `maximum` function. Instead of selecting one maximal element, it breaks the vector into parts of a whole with the maximal input element getting a proportionally larger value, and the other elements getting a lesser proportion of the value. **The property of outputting a probability distribution makes the `softmax` function suitable for probabilistic interpretation in classification tasks**.

Let us consider z as a vector of inputs to the output layer. The output layer units are the number of nodes in the output layer and therefore, the length of the z vector is the number of units in the output layer (if we have ten output units, there are ten z elements).

For an n-dimensional vector $\boldsymbol{Z} = [z_1, z_2, \ldots z_n]$, the `softmax` function produces another n-dimensional vector with values in the range [0, 1].

$$\boldsymbol{Z} = \begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_n \end{bmatrix} and, \quad p(\mathbf{Z}) \rightarrow \begin{bmatrix} p_1 \\ p_2 \\ \vdots \\ p_n \end{bmatrix}$$

$$p_j = \frac{exp^{z_j}}{\sum_{k=1}^{n} exp^{z_k}} \forall j \in 1, 2, \ldots, n \tag{2.4.6}$$

Let us examine the softmax transformation of the 3-element vector (5, 3, 9). The `softmax` activation transforms the vector by the relative size of the vector, and they add up to 1, as is shown in the following code. In the code, we create a random array of dimension = (2, 4), calculate the `softmax` activation of the vector, and sum the outputs of the activation

```
X = array(rnorm(8), dim = c(2,4))

softmax <- function(X){
  exps <- exp(X)

return (exps / rowSums(exps))
}

X
```

```
           [,1]       [,2]       [,3]      [,4]
[1,] -0.1182523 0.9972326 -0.1697167 0.9321687
[2,] -0.1896131 2.2139245 -1.8058676 0.8105641
```

```
output <- softmax(X)

rowSums(output)
```

```
[1] 1 1
```

The probabilities do add up to unity.

Softmax is predominantly used in the output layer for multi-class classification- The `softmax` activation function is used to represent a probability distribution over multiple possible values of a variable, mostly in the output layer. Using this activation function in the hidden layer may cause the hidden nodes to be linearly dependent, whereas we would like to keep them as much linearly independent. In some cases, using this activation function in the hidden layer may also decrease the accuracy and the speed of learning.

However, if we wish that the network needs to choose between one of the possible values for some internal variable, we may use the `softmax` activation function in one of the hidden layers. This is more of an exception than the rule.

2.5 Derivatives of Activation Functions

The derivative of a function is the rate of change of one quantity over another. What this implies is that we can measure the rate of change of the output error (loss) with respect to the network weights. If we know how the error changes with the weights, we can change those weights in a direction that decreases the error.

The partial derivative of a function is the rate of change of one quantity over another, irrespective of another quantity if more than two factors are in the function. Partial derivatives come into play because we train neural networks with gradient descent, where we deal with multiple variables.

2.5.1 Derivative of Sigmoid

The `sigmoid` function outputs a probability score for a specific input and we normally use it as the final layer in neural networks for binary classification. During backpropagation (discussed in Sect. 2.8), we need to calculate the derivative (gradient), so that we can pass it back to the previous layer. The first derivative of the `sigmoid` function will be positive if the input is greater than or equal to zero or negative, if the number is less than or equal to zero, as is shown in Fig. 2.13.

The gradient of the `sigmoid` function is derived below

$$\sigma(Z) = \frac{1}{1 + exp^{-Z}}$$

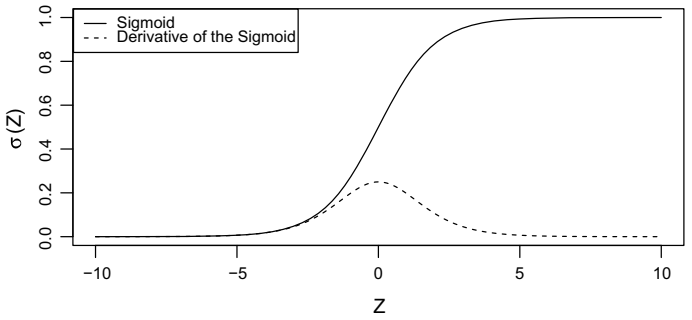

Fig. 2.13 Plot of the Sigmoid activation function and its gradient

$$\begin{aligned}
\frac{d}{dZ}\sigma(Z) &= \frac{d}{dZ}\frac{1}{1+exp^{-Z}} \\
&= \frac{d}{dZ}(1+exp^{-Z})^{-1} \\
&= -(1+exp^{-Z})^{-2}(-exp^{-Z}) \\
&= \frac{exp^{-Z}}{(1+exp^{-Z})^2} \\
&= \frac{1}{1+exp^{-Z}}\left(\frac{1+exp^{-Z}}{1+exp^{-Z}} - \frac{1}{1+exp^{-Z}}\right) \\
&= \frac{1}{(1+exp^{-Z})}\left(1 - \frac{1}{1+exp^{-Z}}\right) \\
&= \sigma(Z)(1-\sigma(Z))
\end{aligned} \tag{2.5.1}$$

2.5.2 Derivative of tanh

The hyperbolic tangent function is represented as

$$g(Z) = tanh(Z) = \frac{sinh(Z)}{cosh(Z)} = \frac{exp^{Z} - exp^{-Z}}{exp^{Z} + exp^{-Z}}$$

The derivative of the hyperbolic tangent function is derived as below

$$\begin{aligned}\frac{d}{dZ}tanh(Z) &= \frac{d}{dZ}\frac{sinh(Z)}{cosh(Z)} \\ &= \frac{\frac{d}{dZ}sinh(Z) \times cosh(Z) - \frac{d}{dZ}cosh(Z) \times sinh(Z)}{cosh^2(Z)} \\ &= \frac{cosh^2(Z) - sinh^2(Z)}{cosh^2(Z)} \\ &= 1 - \frac{sinh^2(Z)}{cosh^2(Z)} \\ &= 1 - tanh^2(Z)\end{aligned} \tag{2.5.2}$$

2.5.3 Derivative of Rectified Linear Unit

We are aware that `relu` function is represented as

$$relu(Z) = g(Z) = max(Z, 0)$$

The derivative of `relu` is

$$\frac{d}{dZ}relu(Z) = \begin{cases} 1 & \text{if } Z > 0 \\ 0 & \text{otherwise} \end{cases} \tag{2.5.3}$$

2.5.4 Derivative of Leaky Rectified Linear Unit

$$lrelu(Z) = g(Z) = \begin{cases} Z & \text{if } Z > 0 \\ aZ & \text{otherwise} \end{cases}$$

Similarly the derivative of `lrelu` is

$$\frac{d}{dZ}lrelu(Z) = \begin{cases} 1 & \text{if } Z > 0 \\ a & \text{otherwise} \end{cases} \tag{2.5.4}$$

2.5.5 Derivative of Softmax

We have discussed why the `softmax` function is often used in the output layer in neural networks. Since it is used in the output layer for multi-class classification,

we need to calculate the derivative so as to pass it back to the previous layer during backpropagation.

From Eq. 2.4.6, `softmax` function is

$$p_i = \frac{exp^{z_i}}{\sum_{k=1}^{n} exp^{z_k}} \quad \forall\, i \in 1, \ldots n$$

Since `softmax` takes multiple inputs in the form of a vector and produces multiple outputs in the form of an output vector we need to specify, which output component of `softmax` we are seeking to find the derivative of.

The `softmax` equation can be interpreted as $p_i = P(y = i \mid z)$,
where the output class is represented as $y \in 1, \ldots n$ and z is a n-dimensional vector.

The partial derivative of the ith output p_i with respect to the jth input z_j can be represented as $\frac{\partial p_i}{\partial z_j}$.

The derivative matrix (which is a Jacobian matrix),[2] can be represented as

$$\frac{\partial p}{\partial z} = \begin{bmatrix} \frac{\partial p_1}{\partial z_1} & \frac{\partial p_1}{\partial z_2} & \cdots & \frac{\partial p_1}{\partial z_n} \\ \frac{\partial p_2}{\partial z_1} & \frac{\partial p_2}{\partial z_2} & \cdots & \frac{\partial p_2}{\partial z_n} \\ \vdots & \vdots & \vdots & \vdots \\ \frac{\partial p_n}{\partial z_1} & \frac{\partial p_n}{\partial z_2} & \cdots & \frac{\partial p_n}{\partial z_n} \end{bmatrix} \tag{2.5.5}$$

For an arbitrary i and j, the derivative $\frac{\partial p_i}{\partial z_j}$ is

$$\frac{\partial p_i}{\partial z_j} = \frac{\partial \frac{exp^{z_i}}{\sum_{k=1}^{n} exp^{z_k}}}{\partial z_j} \tag{2.5.6}$$

We know from the partial rule in calculus, that if $f(x) = \frac{g(x)}{h(x)}$-

$$f'(x) = \frac{g'(x)h(x) - h'(x)g(x)}{(h(x))^2}$$

In our case, $g(x) = exp^{z_i}$ and $h(x) = \sum_{k=1}^{n} exp^{z_k}$

$$\frac{\partial h(x)}{\partial (exp^{z_j})} = \frac{\partial \sum_{k=1}^{n} exp^{z_k}}{\partial (exp^{z_j})} = exp^{z_j} \quad \forall\, j$$

and,

$$\frac{\partial g(x)}{\partial (exp^{z_j})} = \frac{\partial (exp^{z_i})}{\partial (exp^{z_j})} = exp^{z_j} \quad \text{-only when } i = j$$

We therefore have two situations to calculate the derivative

[2] In vector calculus, a Jacobian matrix is computed from the first-order partial derivatives of a vector function. When the output is a square matrix, it is named as a Jacobian.

- when $i = j$

$$\begin{aligned}\frac{\partial \frac{exp^{z_i}}{\sum_{k=1}^{n} exp^{z_k}}}{\partial z_j} &= \frac{exp^{z_i}\sum_{k=1}^{n} exp^{z_k} - exp^{z_j}exp^{z_i}}{\left(\sum_{k=1}^{n} exp^{z_k}\right)^2} \\ &= \frac{exp^{z_i}}{\sum_{k=1}^{n} exp^{z_k}} \frac{\left(\sum_{k=1}^{n} exp^{z_k}\right) - exp^{z_j}}{\sum_{k=1}^{n} exp^{z_k}} \\ &= \frac{exp^{z_j}}{\sum_{k=1}^{n} exp^{z_k}} \frac{\left(\sum_{k=1}^{n} exp^{z_k}\right) - exp^{z_j}}{\sum_{k=1}^{n} exp^{z_k}} \\ &= p_i(1 - p_j)\end{aligned} \tag{2.5.7}$$

- when $i \neq j$

$$\begin{aligned}\frac{\partial \frac{exp^{z_i}}{\sum_{k=1}^{n} exp^{z_k}}}{\partial z_j} &= \frac{0 - exp^{z_j}exp^{z_i}}{\left(\sum_{k=1}^{n} exp^{z_k}\right)^2} \\ &= -\frac{exp^{z_j}}{\sum_{k=1}^{n} exp^{z_k}} \frac{exp^{z_i}}{\sum_{k=1}^{n} exp^{z_k}} \\ &= -p_j p_i\end{aligned} \tag{2.5.8}$$

To summarize above, the derivative of the softmax function is

$$\frac{\partial p_i}{\partial z_j} = \begin{cases} p_i(1 - p_j) & \text{if } i = j \\ -p_i p_j & \text{if } i \neq j \end{cases} \tag{2.5.9}$$

It is left to the reader to think what happens when $i = j$.

2.6 Cross-Entropy Loss

As we have seen in Sect. 1.11.4, Cross-Entropy (*CE*) measures the divergence between two probability distributions. If the *CE* is large it means that the difference between the two distributions is large and, if the *CE* is small, it implies that the two distributions are similar to each other. *CE* therefore gives us an estimate of the distance between what the model believes the output distribution should be, and what the original distribution really is.

Loss function is usually a function defined for a observation (data point) and, it measures the penalty (difference between a single label and the predicted label).

Cost function is usually more general. It is more generally a sum of the loss functions over the training data set and the penalty for model complexity, i.e., regularization.

However, they are sometimes used interchangeably.

CE is used as a loss function in NNs (neural networks), which have `sigmoid`/`softmax` activation in the output layer and is defined as

$$H(Y, p) = -\sum_i y_i log(p_i)$$

Suppose we have a neural network model, (known as the hypothesis), which predicts n classes $1, 2, \ldots, n$ having probability of occurrences as $\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_n$ and having k_i instances for each of the n classes, i.e., $k_1 + k_2 + \cdots + k_n = m$, where m is the total number of observations.

As per our hypothesis (i.e. model), the likelihood of this happening can be represented as

$$P(data \mid model) = \hat{y}_1^{k_1} \hat{y}_2^{k_2} .. \hat{y}_n^{k_n}$$

Using the negative log-likelihood,

$$-logP(data \mid model) = -\sum_i k_i log(\hat{y}_i)$$

Denoting the empirical probabilities as $y_i = k_i/m$, we can write the generalized *CE* loss function as

$$-\frac{1}{m} logP(data \mid model) = -\frac{1}{m} \sum_i k_i log(\hat{y}_i) = -\sum_i y_i log(\hat{y}_i) \tag{2.6.1}$$

As we have discussed in the section on entropy, the log-likelihood of a data set, given a model, can be interpreted as "how surprised we would be if we get an outcome after we made our initial prediction". From the view point of information theory, it is the number of bits we would expect to spend to encode this information if our encoding scheme is based on our hypothesis. Intuitively, given true $y^{(i)}$, we would like to optimize our model to get the predicted $\hat{y}^{(i)}$ as close as possible to $y^{(i)}$.

Generally speaking, **it is not a good idea** to use `relu`, `lrelu` or `tanh` activations on the output layer, in combination with a *CE* loss. This is because *CE* is a **cost function, which attempts to compute the difference between two probability distribution functions**.

The output of a neural network needs to satisfy the following criteria to represent a probability distribution function such that each of the categories in the distribution can be represented using a probability value and, summing to one

- The probability value of each output is between zero and one.
- The sum of all probability values equals one.

Generally, we use `softmax` activation instead of `sigmoid` with *CE* loss because **`softmax` activation distributes the probability throughout each output node**.

But, for binary classification, `sigmoid` activation is the same as `softmax`.

Most often, the output layer of a neural network is activated using a `softmax` or `sigmoid` function, which forces the output of the network to satisfy the above-mentioned criteria.

Using a `relu`, `lrelu` or `tanh` activation function in the output, with a *CE* loss is therefore unreliable as these activation functions do not generate values that can be interpreted as probabilities, whereas *CE* requires its inputs to be interpreted as probabilities.

We therefore derive the *CE* loss function for the `sigmoid` and `softmax` activations functions only.

The prediction at the final layer activation can be written as $\hat{y}^{(i)} = A^{[L](i)}$. For binary classification using `sigmoid` function, we can assume the value of $\hat{y}^{(i)}$ to be either 0 or 1.

For a network with one output predicting two classes, i.e., an output of 1 for positive class membership and 0 for negative class membership, the (i) in $\hat{y}^{(i)}$ may have only one value.

Therefore, in Eq. 2.6.1, for the `sigmoid`, $\hat{y}^{(i)}$ and $y^{(i)}$ can be interpreted as the corresponding binary distributions $(\hat{y}^{(i)}, 1 - \hat{y}^{(i)})$ and $(y^{(i)}, 1 - y^{(i)})$.

The *CE* loss function for `sigmoid` is represented as

$$\begin{aligned} \mathcal{J}_{sigmoid}(w) &= -\sum_i y_i log(\hat{y}_i) \\ &= -\frac{1}{m}\sum_{i=1}^{m}(y^{(i)} \log\left(\hat{y}^{(i)}\right) + (1 - y^{(i)}) \log\left(1 - \hat{y}^{(i)}\right)) \end{aligned} \tag{2.6.2}$$

and the *CE* loss for the complete data set with m observations and K classes for `softmax` is

$$\mathcal{J}_{softmax} = -\sum_i^m \sum_k^K y^{(k)} log(\hat{y}^{(k)}) \quad [\text{for } m \text{ observations and } K \text{ classes}] \tag{2.6.3}$$

> While using the `sigmoid` function, there is a danger if $\hat{y} = 0$. In that case, we will get *NaN* values.
>
> But for a randomly initialized `softmax` layer, it is extremely unlikely to output an exact zero for any class, but it may still be possible.
>
> A recommended solution while using the `sigmoid` function, could be to avoid calculating $log(\hat{y})$ when $\hat{y} = 0$ (while calculating the cost); or coerce the $\hat{y}$

value to a nonzero number before backpropagation, (by using something like $log(max(\hat{y}, 1e\text{-}15))$).

Please also to refer to Sect. 4.2 on dealing with NaNs.

2.7 Derivative of the Cost Function

For reasons mentioned in section above, we will endeavor to find the gradient of the `sigmoid` and the `softmax` activation functions only (as they are mostly used in the output layer).

2.7.1 Derivative of Cross-Entropy Loss with Sigmoid

The derivative of the *CE* loss function for the `sigmoid` activation function is relatively simple (refer Eq. 1.11.6)

$$\begin{aligned} \mathcal{J}_{sigmoid}(w) &= -\frac{1}{m}(Y \log(A) + (1-Y)\log(1-A)) \\ \frac{\partial \mathcal{J}_{sigmoid}(w)}{\partial w} &= -\frac{Y}{A} + \frac{(1-Y)}{1-A} \end{aligned} \tag{2.7.1}$$

2.7.2 Derivative of Cross-Entropy Loss with Softmax

In Sect. 2.4.5, we had defined the `softmax` function in `R`. Let us now define the *CE* loss of `softmax` in `R`. *(This is only a coded representation of the `softmax` cross-entropy loss; albeit with no output.)*

```
cross_entropy_softmax <- function(X, y){
  m = length(y)
  p = softmax(X)
  log_likelihood = -log(p[m, y])
  loss = sum(log_likelihood) / m

  return(loss)
}
```

The derivative of the *CE* loss of the `softmax` function requires a little more math and is derived as under *(o represents the output)*

$$
\begin{aligned}
\mathcal{J} &= -\sum_i y_k log(\hat{y}_k) \\
\frac{\partial \mathcal{J}}{\partial o_i} &= -\sum_k y_k \frac{\partial log(\hat{y}_k)}{\partial o_i} \\
&= -\sum_k y_k \frac{\partial log(\hat{y}_k)}{\partial \hat{y}_k} \frac{\partial \hat{y}_k}{\partial o_i} \\
&= -\sum_k y_k \frac{1}{\hat{y}_k} \frac{\partial \hat{y}_k}{\partial o_i} \\
&= -y_i(1-\hat{y}_i) - \sum_{k \neq i} y_k \frac{1}{\hat{y}_k}(-\hat{y}_k \hat{y}_i) \\
&= -y_i(1-\hat{y}_i) + \sum_{k \neq i} y_k \hat{y}_i \\
&= -y_i + y_i \hat{y}_i + \sum_{k \neq i} y_k \hat{y}_i \\
&= \hat{y}_i(y_i + \sum_{k \neq i} y_k) - y_i
\end{aligned}
$$

y is a one-hot encoded vector of the labels, where the rows sum to one, therefore

$$
\left(\sum_k y_k = 1\right) \quad and \quad \left(y_i + \sum_{k \neq i} y_k = 1\right)
$$

$$
\frac{\partial \mathcal{J}}{\partial o_i} = \hat{y}_i - y_i \tag{2.7.2}
$$

2.8 Back Propagation

Backpropagation was first applied to the task for optimizing neural networks by gradient descent in a paper titled *Learning Internal Representations by Error Propagation*, 1985 by [3]. Later, work was done in the late 80's and 90's by [56], who first applied it to convolutional neural networks. The success of neural networks is largely attributed to them for their groundbreaking efforts, including Geoffrey E. Hinton, who was one of the first researchers who demonstrated the use of the generalized backpropagation algorithm for training multilayer neural networks.

Backpropagation is usually seems to be the most complex and most mathematical part in deep learning, but it is relatively simple if we look at it piecewise.

We have seen earlier that a neural network propagates forward through the layers by a set of connections known as weights and to the final hidden layer and on to the output layer to make a prediction. Like in any machine learning algorithm as well in deep learning, the important part is to estimate the correct weights. The metric to

evaluate the weights is the accuracy of prediction or the loss function of the network. We try to arrive at the minimum cost and the only way we can change the cost is by changing the weights, as the loss function is parameterized by the weights of each layer.

How do we optimize our loss function?

Minimizing any function involves finding the derivative of the function and setting it to zero, i.e., $\frac{\partial \mathcal{J}}{\partial w} = 0$. However, in the case of neural networks (unlike most conventional machine learning models), we are faced with some difficulties to find the local minima of the loss function, which are

- We are dealing with weights from different layers, therefore multiple equations need to be optimized.
- Each layer may have different number of nodes, therefore number of weights for each layer will be different.
- There exists multiple local minima and we have to settle at a global minima.

To perform backpropagation, we need to first establish an objective (cost) function to measure performance. For regression problems, this is often the mean squared error (*MSE*) and for classification problems it is the binary and multi-categorical *CE*. Neural networks can have multiple loss functions but we will focus on using one.

On each forward pass the network will measure its performance based on the chosen cost function. The network will then work backwards through the layers, compute the gradient of the cost with respect to the network weights, adjust the weights a little in the opposite direction of the gradient, and repeat till the loss function is minimized.

Let us revisit the network represented in Fig. 2.7.

Calculation of Cost of Sigmoid Activation in Output Layer

The *CE* cost function using `sigmoid` activation for the network, represented in Fig. 2.7, is mathematically represented as

$$\mathcal{J} = -(Y \cdot log A^{[3]} + (1 - Y) \cdot log(1 - A^{[3]}))$$

We had earlier derived the derivative of the above as-

$$\frac{\partial \mathcal{J}}{\partial A^{[3]}} = -\frac{Y}{A^{[3]}} + \frac{(1 - Y)}{(1 - A^{[3]})} \tag{2.8.1}$$

The backpropagation equations for the network layers are

$$\begin{aligned} dZ3 &= \frac{\partial \mathcal{J}}{\partial Z^{[3]}} = \frac{\partial \mathcal{J}}{\partial A^{[3]}} \frac{\partial A^{[3]}}{\partial Z^{[3]}} \\ &= \left(-\frac{Y}{A^{[3]}} + \frac{(1 - Y)}{(1 - A^{[3]})} \right) A^{[3]}(1 - A^{[3]}) \\ &= A^{[3]} - Y \end{aligned} \tag{2.8.2}$$

$$\begin{aligned} dW3 &= \frac{\partial \mathcal{J}}{\partial W^{[3]}} = \frac{\partial \mathcal{J}}{\partial Z^{[3]}} \frac{\partial Z^{[3]}}{\partial W^{[3]}} \\ &= \frac{\partial \mathcal{J}}{\partial Z^{[3]}} A^{[2]} \\ &= (A^{[3]} - Y) A^{[2]} \\ &= dZ3\, A^{[2]} \end{aligned} \tag{2.8.3}$$

$$\begin{aligned} db3 &= \frac{\partial \mathcal{J}}{\partial b^{[3]}} = \frac{\partial \mathcal{J}}{\partial Z^{[3]}} \frac{\partial Z^{[3]}}{\partial b^{[3]}} \\ &= (A^{[3]} - Y) * 1 \\ &= dZ3 \end{aligned} \tag{2.8.4}$$

$$\begin{aligned} dA2 &= \frac{\partial \mathcal{J}}{\partial A^{[2]}} = \frac{\partial \mathcal{J}}{\partial Z^{[3]}} \frac{\partial Z^{[3]}}{\partial A^{[2]}} \\ &= (A^{[3]} - Y)\ W^{[3]} \\ &= [dZ3[][W^{[3]}] \end{aligned} \tag{2.8.5}$$

$$\begin{aligned} dZ2 &= \frac{\partial \mathcal{J}}{\partial Z^{[2]}} = \frac{\partial \mathcal{J}}{\partial A^{[3]}} \frac{\partial A^{[3]}}{\partial Z^{[3]}} \frac{\partial Z^{[3]}}{\partial A^{[2]}} \frac{\partial A^{[2]}}{\partial Z^{[2]}} \\ &= [A^{[3]} - Y][W^{[3]}] \frac{\partial \sigma(Z^{[2]})}{\partial Z^{[2]}} \\ &= [dA2][A^{[2]}(1 - A^{[2]})] \end{aligned} \tag{2.8.6}$$

$$\begin{aligned} dW2 &= \frac{\partial \mathcal{J}}{\partial W^{[2]}} = \frac{\partial \mathcal{J}}{\partial Z^{[2]}} \frac{\partial Z^{[2]}}{\partial W^{[2]}} \\ &= [(A^{[3]} - Y) W^{[3]} A^{[2]}(1 - A^{[2]})] A^{[1]} \\ &= dZ2\, A^{[1]} \end{aligned} \tag{2.8.7}$$

$$\begin{aligned} \frac{\partial \mathcal{J}}{\partial b^{[2]}} &= \frac{\partial \mathcal{J}}{\partial Z^{[2]}} \frac{\partial Z^{[2]}}{\partial b^{[2]}} \\ &= (A^{[3]} - Y) W^{[3]} A^{[2]}(1 - A^{[2]}) * 1 \\ &= dZ2 \end{aligned} \tag{2.8.8}$$

$$\begin{aligned} dA1 &= \frac{\partial \mathcal{J}}{\partial A^{[1]}} = \frac{\partial \mathcal{J}}{\partial Z^{[2]}} \frac{\partial Z^{[2]}}{\partial A^{[1]}} \\ &= [(A^{[3]} - Y) W^{[3]} A^{[2]} (1 - A^{[2]})][W^{[2]}] \\ &= dZ2\ W^{[2]} \end{aligned} \tag{2.8.9}$$

$$\begin{aligned} dZ1 &= \frac{\partial \mathcal{J}}{\partial Z^{[1]}} = \frac{\partial \mathcal{J}}{\partial A^{[2]}} \frac{\partial A^{[2]}}{\partial Z^{[2]}} \frac{\partial Z^{[2]}}{\partial A^{[1]}} \frac{\partial A^{[1]}}{\partial Z^{[1]}} \\ &= [(A^{[3]} - Y) W^{[3]}][A^{[2]} (1 - A^{[2]})][W^{[2]}] \frac{\partial \sigma(Z^{[1]})}{\partial Z^{[1]}} \\ &= dA1[A^{[1]}(1 - A^{[1]})] \end{aligned} \tag{2.8.10}$$

$$\begin{aligned} d\,W1 &= \frac{\partial \mathcal{J}}{\partial W^{[1]}} = \frac{\partial \mathcal{J}}{\partial Z^{[1]}} \frac{\partial Z^{[1]}}{\partial W^{[1]}} \\ &= [(A^{[3]} - Y) W^{[3]} A^{[2]} (1 - A^{[2]}) W^{[2]} A^{[1]} (1 - A^{[1]})] X \\ &= dZ1\ X \end{aligned} \tag{2.8.11}$$

$$\begin{aligned} db1 &= \frac{\partial \mathcal{J}}{\partial b^{[1]}} = \frac{\partial \mathcal{J}}{\partial Z^{[1]}} \frac{\partial Z^{[1]}}{\partial b^{[1]}} \\ &= dZ1 \end{aligned} \tag{2.8.12}$$

2.8.1 Summary of Backward Propagation

The backpropagation equations for a single observation can be generalized as

$$\begin{aligned} dz^{[\ell]} &= da^{[\ell]}\ g'^{[\ell]}(z^{[\ell]}) \\ d\,w^{[\ell]} &= dz^{[\ell]}\ a^{[\ell-1]} \\ db^{[\ell]} &= dz^{[\ell]} \\ da^{[\ell-1]} &= w^{[\ell+1]T} dZ^{[\ell]} \\ dz^{[\ell]} &= w^{[\ell+1]T}\ dz^{[\ell+1]}\ g'^{[\ell]}(z^{[l]}) \end{aligned} \tag{2.8.13}$$

The vectorized form to include all observations is

$$\begin{aligned} dZ^{[\ell]} &= dA^{[\ell]}\, \sigma'^{[\ell]}(Z^{[\ell]}) \\ dW^{[\ell]} &= \frac{1}{m} dZ^{[\ell]}\, A^{[\ell-1]T} \\ db^{[\ell]} &= \frac{1}{m} rowSums(dZ^{[\ell]}) \\ dA^{[\ell-1]} &= W^{[\ell]}\, dZ^{[\ell]} \end{aligned} \tag{2.8.14}$$

2.9 Writing a Simple Neural Network Application

We are now ready to write a composite code to:

- define the architecture of a neural network,
- initialize its parameters,
- calculate the the predictions using forward propagation,
- calculate the loss,
- correct the loss using a backpropagation algorithm (we will use the simple gradient descent method),
- iterate the same procedure, till we arrive at a negligible cost.

We will construct a single hidden layer with a single node and a `sigmoid` activation neural network.

The `sigmoid` activation function can be written as below

```
sigmoid <- function(x){
  1/(1+exp(-x))
}
```

The following function creates a vector of zeros by initializing our weight vector and bias scalars to zero.

```
initialize_with_zeros <- function(dim){
w = matrix(0, nrow = dim, ncol = 1)
b = 0

return(list(w, b))
}
```

After initializing our parameters above, we will carry out the `forward propagation` of the network to learn our parameters. The `propagate()` function below calculates the cost and its gradient. This is the negative log-likelihood cost of the logistic regression, defined in Eq. 1.11.3. We then find the gradient of the cost as follows:

$$\mathbf{A} = \sigma(w^T X + b) = (a^{(1)}, a^{(2)}, \ldots, a^{(m)})$$

$$\mathcal{J} = -\frac{1}{m}\sum_{i=1}^{m} y^{(i)} log(a^{(i)} + (1 - y^{(i)}) log(1 - a^{(i)}))$$

$$\frac{\partial \mathcal{J}}{\partial w} = \frac{\partial \mathcal{J}}{\partial z}\frac{\partial z}{\partial w} = (a^{(i)} - y^{(i)}) * x^{(i)}$$

$$\frac{\partial \mathcal{J}}{\partial w} = \frac{1}{m} X \sum_{i=1}^{m} (a^{(i)} - y^{(i)})^T$$

$$\frac{\partial \mathcal{J}}{\partial b} = \frac{1}{m}\sum_{i=1}^{m} (a^{(i)} - y^{(i)})$$

dw is the gradient of the cost with respect to w and therefore shares the same dimension as w. Similarly, db is the gradient of the cost with respect to b and has the same dimension as b.

The `propagate` function returns the gradients dw, db and the cost in the form of a list.

```
propagate <- function(w, b, X, Y){

m = ncol(X)

# Forward Propagation
A = sigmoid((t(w) %*% X) + b)
cost = (-1 / m) * sum(Y * log(A) + (1 - Y) * log(1 - A))

# Backward Propagation
dw = (1 / m) * (X %*% t(A - Y))
db = (1 / m) * rowSums(A - Y)

grads <- list(dw, db)

return(list(grads, cost))
}
```

In the following function `optimize`, we run the gradient descent algorithm to update the structural parameters of the neural network, w and b.

We run the gradient descent algorithm for a finite number of times defined by the hyperparameter, *num_iter*, and *learning_rate*, which is the learning step for the gradient descent.

The function prints the cost, J for every 500th iteration. The cost for every iteration is stored in a list named *costs*.

This function returns the updated weights and bias w and b in a list called *params* and the gradients dw and db in a list called *grads*.

The dimensions of the input vector $\boldsymbol{X}$ are (number of features, number of observations), i.e., $(12287, 200)$ and, that of the $\boldsymbol{Y}$ vector is (1, number of observations), i.e., $(1, 200)$, respectively.

```
optimize <- function(w, b, X, Y, num_iter, learning_rate, print_cost = FALSE) {

    cost <- list()
```

```
    for (i in 1:num_iter) {
        # Cost and gradient calculation
        grads = propagate(w, b, X, Y)[[1]]  # grads is a list
        cost[i] = propagate(w, b, X, Y)[[2]]

        # Retrieve the derivatives
        dw = matrix(grads[[1]])
        db = grads[[2]]

        # Update the parameters
        w = w - learning_rate * dw
        b = b - learning_rate * db

        # Record the cost
        if (i%%100 == 0) {
            costs <- cost
        }

        # Print the cost every 500th iteration
        if ((print_cost == T) & (i%%500 == 0)) {
            cat(sprintf("Cost after iteration %d: %06f\n", i,
                cost[[i]]))
        }

        params <- list(w, b)
        grads <- list(dw, db)
    }

    return(list(params, grads, costs))
}
```

Having computed the optimized structural parameters w and b, we are now in a position to predict the output as either 0 or 1.

m is the number of observations and the probability threshold is defined as 0.5.

```
pred <- function(w, b, X) {

    m = ncol(X)
    Y_prediction <- matrix(0, nrow = 1, ncol = m)

    # Activation vector A to predict the probability of a dog/cat
    A = sigmoid((t(w) %*% X) + b)

    for (i in 1:ncol(A)) {
        if (A[1, i] > 0.5) {
            Y_prediction[1, i] = 1
        } else Y_prediction[1, i] = 0
    }

    return(Y_prediction)
}
```

In the above code segments, we have completed the following:

- Coded the `sigmoid` activation function
- Initialized the structural parameters w and b of our neural network
- Coded the forward propagation function to predict the labels and the cost, (the difference between the true and predicted labels)
- Predicted the labels with our latest structural parameters w and b

- Reused the parameters to backpropagate, to calculate the gradient of the structural parameters w and b

We are now poised to agglomerate the above functions in our neural network application to iterate the above procedure.

As a reminder, the input layer consists of 12287 nodes (features), with each node having 200 observations.

```
simple_model <- function(X_train,
                         Y_train,
                         X_test,
                         Y_test,
                         num_iter,
                         learning_rate,
                         print_cost = FALSE){

# initialize parameters with zeros
w = initialize_with_zeros(nrow(X_train))[[1]]
b = initialize_with_zeros(nrow(X_train))[[2]]

# Gradient descent
optFn_output <- optimize(w,
                         b,
                         X_train,
                         Y_train,
                         num_iter,
                         learning_rate,
                         print_cost)

parameters <- optFn_output[[1]]
grads <- optFn_output[[2]]
costs <- optFn_output[[3]]

# Retrieve parameters w and b
w = as.matrix(parameters[[1]])
b = parameters[[2]]

# Predict test/train set examples
pred_train = pred(w, b, X_train)
pred_test = pred(w, b, X_test)

# Print train/test Errors
cat(sprintf("train accuracy: %#.2f \n", mean(pred_train == Y_train) * 100))

cat(sprintf("test accuracy: %#.2f \n", mean(pred_test == Y_test) * 100))


res = list("costs"= costs,
          "pred_train" = pred_train,
          "pred_test"= pred_test,
          "w" = w,
          "b" = b,
          "learning_rate" = learning_rate,
          "num_iter" = num_iter)

return(res)
}
```

For our training and testing data sets, we will use the "Dogs vs. Cats" data set.[3]

[3]downloaded from https://www.kaggle.com/c/dogs-vs-cats/data on April 02, 2018, 07:40 IST.

```
library(EBImage)
file_path_train <- "~/data/train"
file_path_test <- "~/data/test"

library(pbapply)
height = 64
width = 64
channels = 3

extract_feature <- function(dir_path, width, height) {
    img_size <- width * height

    images <- list.files(dir_path)
    label <- ifelse(grepl("dog", images) == T, 1, 0)
    print(paste("Processing", length(images), "images"))

    feature_list <- pblapply(images, function(imgname) {

        img <- readImage(file.path(dir_path, imgname))
        img_resized <- EBImage::resize(img, w = width, h = height)
        img_matrix <- matrix(reticulate::array_reshape(img_resized, (width *
            height * channels)), nrow = width * height * channels)
        img_vector <- as.vector(t(img_matrix))

        return(img_vector)
    })

    feature_matrix <- do.call(rbind, feature_list)

    return(list(t(feature_matrix), label))
}
```

```
# Reshape the images

data_train <-extract_feature(file_path_train,width = 64,height = 64)
```

```
[1] "Processing 200 images"
```

```
trainx <-data_train[[1]]
trainy <-data_train[[2]]
dim(trainx)
```

```
[1] 12288   200
```

```
data_test <-extract_feature(file_path_test,width = 64,height = 64)
```

```
[1] "Processing 50 images"
```

```
testx <-data_test[[1]]
testy<- data_test[[2]]
dim(testx)
```

```
[1] 12288    50
```

Let us have a look at the actual image and the resized image (Figs. 2.14 and 2.15).

```
par(mfrow = c(1, 2))

images <- list.files(file_path_train)
img <- readImage(file.path(file_path_train, images[101]))
```

Fig. 2.14 An image from the train set is shown in the left along with the resized image having dimensions $64 \times 64 \times 3$ on the right

```
EBImage::display(img, method = "raster")
EBImage::display(matrix(as.matrix(trainx[, 101]),
                        c(64, 64, 3),
                        byrow = TRUE),
                 method = 'raster')
```

```
par(mfrow = c(1, 2))

images <- list.files(file_path_test)
img <- readImage(file.path(file_path_test, images[18]))

EBImage::display(img, method = "raster")
EBImage::display(matrix(as.matrix(testx[, 18]),
                        c(64, 64, 3),
                        byrow = TRUE),
                 method = 'raster')
```

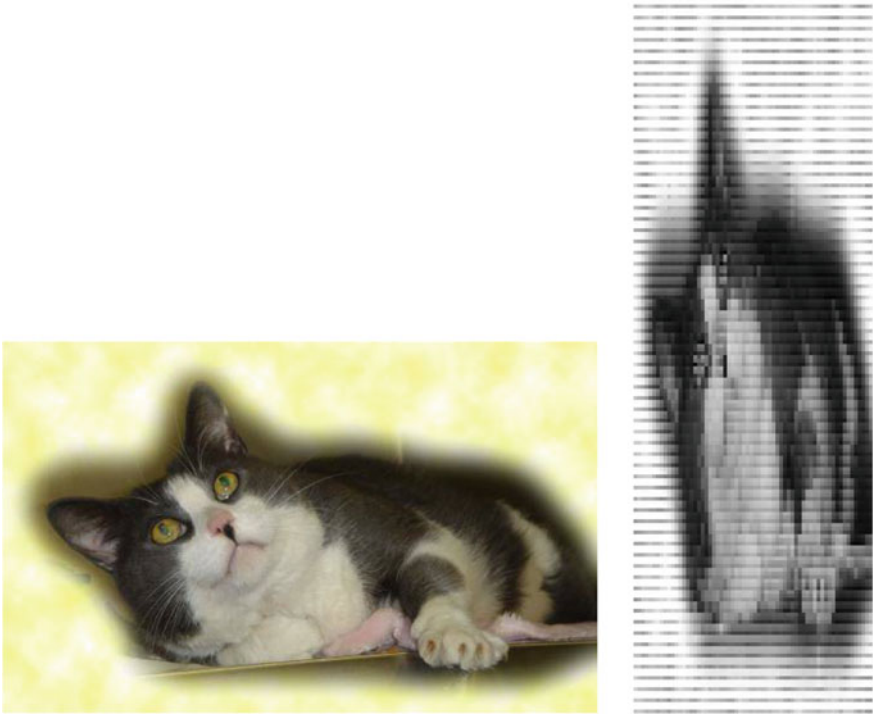

Fig. 2.15 An image from the test set is shown in the left along with the resized image having dimensions $64 \times 64 \times 3$ in the right

> Data normalization is important in neural networks. When we apply the `sigmoid` function to our hypothesis, the output probabilities can take values which are approximately all 0's or all 1's rendering the cost function a *NaN* value. This in most cases is likely due to the widely different ranges of each feature rendering the weighted sum of $x \times w$ to assume very large negative or positive values. When we apply the `sigmoid` function to these values, we run the risk of ending up with values very close to 0 or 1, rendering NaN values to the gradient.
>
> One way to combat this problem is to normalize the data with zero mean and unit variance, before performing training using gradient descent.

```
trainx <- scale(trainx)
testx <- scale(testx)
```

Time to run our simple neural network model (Fig. 2.16 and 2.17).

```
model = simple_model(trainx,
                     trainy,
                     testx,
```

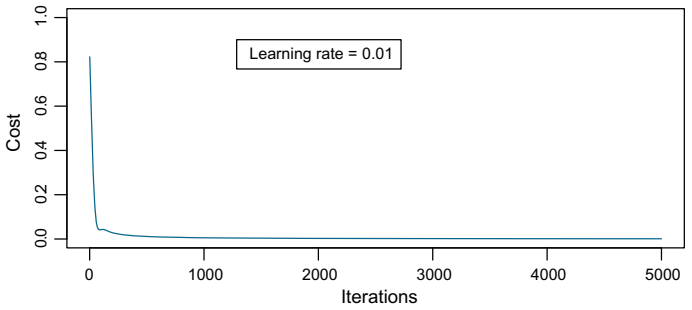

Fig. 2.16 Cost of our sigmoid activation neural network with a learning rate of 0.001

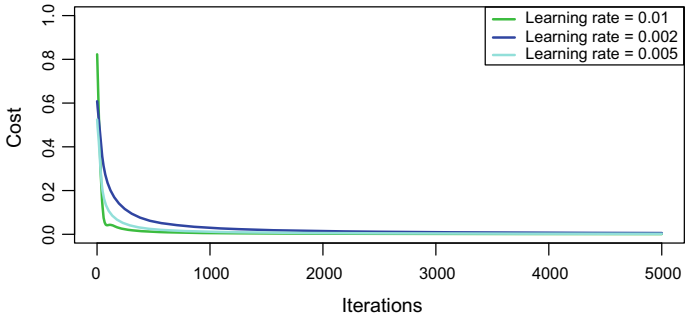

Fig. 2.17 Plot of Cost versus iteration with different learning rates

```
                    testy,
                    num_iter = 5000,
                    learning_rate = 0.01,
                    print_cost = TRUE)
```

```
Cost after iteration 500: 0.011374
Cost after iteration 1000: 0.005671
Cost after iteration 1500: 0.003772
Cost after iteration 2000: 0.002825
Cost after iteration 2500: 0.002257
Cost after iteration 3000: 0.001880
Cost after iteration 3500: 0.001610
Cost after iteration 4000: 0.001408
Cost after iteration 4500: 0.001251
Cost after iteration 5000: 0.001126
train accuracy: 100.00
test accuracy: 62.00
```

```
# Plot Cost vs Iteration
x = c(1:5000)
y = model$costs
smoothingSpline = smooth.spline(x, y, spar = 0.35)

plot(NULL, type = "n",
     xlab = "Iterations", ylab = "Cost",
     xlim = c(1, 5000), ylim = c(0, 1),
```

```
     xaxt = "n", yaxt = "n",
     cex.lab = 0.7)
lines(smoothingSpline, col = 'deepskyblue4')
axis(side = 1,  col = "black", cex.axis = 0.7)
axis(side = 2,  col = "black", cex.axis = 0.7)

legend(1550, 0.9, inset = 0.001, c('Learning rate = 0.01'), cex = 0.6)
```

Let us play around with the learning rate hyperparameter and plot the cost with different learning rates:

```
learning_rates = c(0.01, 0.002, 0.005)
models = list()
smoothingSpline <- list()

plot(NULL, type = "n",
     xlab = "Iterations", ylab = "Cost",
     xlim = c(1, 5000), ylim = c(0, 1),
     xaxt = "n", yaxt = "n",
     cex.lab = 0.7)

for(i in 1:length(learning_rates)){
  cat(sprintf("Learning rate: %#.3f \n", learning_rates[i]))
  models[[i]] = simple_model(trainx,
                             trainy,
                             testx,
                             testy,
                             num_iter = 5000,
                             learning_rate = learning_rates[i],
                             print_cost = F)
  cat('\n-------------------------------------------------------\n')

  x = c(1:5000)
  y = unlist(models[[i]]$costs)
  smoothingSpline = smooth.spline(x, y, spar = 0.35)

  lines(smoothingSpline, col = i + 2, lwd = 2)
}
```

```
Learning rate: 0.010
train accuracy: 100.00
test accuracy: 62.00

-------------------------------------------------------
Learning rate: 0.002
train accuracy: 100.00
test accuracy: 60.00

-------------------------------------------------------
Learning rate: 0.005
train accuracy: 100.00
test accuracy: 62.00

-------------------------------------------------------
```

```
axis(side = 1,  col = "black", cex.axis = 0.7)
axis(side = 2,  col = "black", cex.axis = 0.7)

legend("topright", inset = 0.001,
       c('Learning rate = 0.01',
         'Learning rate = 0.002',
```

```
        'Learning rate = 0.005'),
      lwd = c(2, 2, 2),
      lty = c(1, 1, 1),
      col = c('green3', 'blue', 'cyan'),
      cex = 0.6)
```

Although our model is highly overfitted, we have constructed a simple neural network model up from scratch.

In the following chapters, we will explore how we can improve the predictive power by constructing a deep neural network with different activations.

2.10 Conclusion

We now have a fair idea about the basic neural network architecture, the activation functions used in neural networks and their limitations, and the forward propagation and backward propagation procedures.

We have also constructed a simple neural network using `sigmoid` activation and explored the outcomes using different learning rates.

In the next chapter, we will delve further into deeper neural networks.

Chapter 3
Deep Neural Networks-I

It is not complicated, it is just a lot of it.

Feynman

Abstract In this section we will learn the foundations of deep learning and how deep learning actually works. In particular, we will discuss

- How to build, train and apply a fully connected deep neural network.
- Understand the key parameters in a neural network's architecture.
- Introduction to `keras`

We will also construct a deep learning algorithm from scratch.

3.1 Writing a Deep Neural Network (DNN) Algorithm

We will be using the following packages:

```
library(dplyr)
library(ggplot2)
library(reticulate)
library(keras)
```

Let us first write down the different activation functions.

The following four functions compute the `sigmoid`, `relu`, `tanh`, and `softmax` activations of the input vector.

```
sigmoid <- function(Z) {
    A <- 1/(1 + exp(-Z))
    cache <- Z

    return(list(A = A, Z = Z))
}
```

© Springer Nature Singapore Pte Ltd. 2019

A. Ghatak, *Deep Learning with R*,
https://doi.org/10.1007/978-981-13-5850-0_3

```
relu <- function(Z) {
    A <- pmax(Z, 0)
    cache <- Z

    return(list(A = A, Z = Z))
}
```

```
tanh <- function(Z) {
    A <- sinh(Z)/cosh(Z)
    cache <- Z

    return(list(A = A, Z = Z))
}
```

```
softmax <- function(Z) {
    # get unnormalized probabilities
    exp_scores = exp(t(Z))
    # get the normalized probabilities
    A = exp_scores/rowSums(exp_scores)

    return(list(A = A, Z = Z))
}
```

The following four functions compute the gradient (derivative) of the activations as discussed in Sect. 2.5.

$$\frac{d}{dZ}\sigma(Z) = \sigma(Z)(1 - \sigma(Z))$$

```
derivative_sigmoid <- function(dA, cache) {
    Z <- cache
    s <- 1/(1 + exp(-Z))
    dZ <- dA * s * (1 - s)

    return(dZ)
}
```

$$\frac{d}{dZ}relu(Z) = \begin{cases} 1 & \text{if } Z > 0 \\ 0 & \text{otherwise} \end{cases}$$

```
derivative_relu <- function(dA, cache) {
    Z <- cache
    dZ <- dA
    a <- (Z > 0)  # Find which values of Z are greater than zero
    dZ <- dZ * a  # when Z <= 0, dZ is set to zero

    return(dZ)
}
```

$$\frac{d}{dZ}tanh(Z) = 1 - tanh^2(Z)$$

```
derivative_tanh <- function(dA, cache) {
    Z = cache
    a = sinh(Z)/cosh(Z)
    dZ = dA * (1 - a^2)

    return(dZ)
}
```

```
derivative_softmax <- function(dA, cache, X, Y, num_classes) {
    y.mat <- matrix(Y, ncol = 1)
    y <- matrix(0, nrow = length(Y), ncol = num_classes)

    for (i in 0:(num_classes - 1)) {
        y[y.mat[, 1] == i, i + 1] <- 1
    }

    Z <- cache
    exp_scores = exp(t(Z))
    probs = exp_scores/rowSums(exp_scores)
    dZ = probs - y

    return(dZ)
}
```

In the following function, we initialize the structural parameters of our network. To do so, we need to define the number of layers and the number of nodes in each layer by a parameter `layer_dims` defined by a list. The length of this list is the sum of the number of the input layer, the hidden layers and the output layer. The first entry in the list is the number of input nodes (i.e., features of the input vector); the last entry is the number of nodes in the output layer (which is normally one) and the in-between entries are the number of nodes in the respective hidden layers.

The function returns the initialized values set of the parameters in the form of a vector.

The dimensions of the parameters are: $dim(W^{[\ell]}) =$ (num nodes at layer ℓ, num nodes at layer $\ell - 1$) and $dim(b^{[\ell]}) =$ (num nodes at layer ℓ, 1).

We have defined four different types of initializations in the function, however we will only use random initialization, i.e., choose the parameters from a random gaussian distribution.

We will discuss the other initialization techniques and their effect on convergence in the next chapter.

```
initialize_params <- function(layers_dims, initialization){
  set.seed(2)
  layerParams <- list()
  for(layer in 2:length(layers_dims)){

    if(initialization == 'zero'){
      n = 0 * rnorm(layers_dims[layer] * layers_dims[layer - 1])

      layerParams[[paste('W', layer - 1, sep = "")]] =
        matrix(n,
               nrow = layers_dims[layer],
               ncol = layers_dims[layer - 1])
      layerParams[[paste('b', layer - 1, sep = "")]] =
```

```
        matrix(rep(0, layers_dims[layer]),
               nrow = layers_dims[layer],
               ncol = 1)
    }
    else if(initialization == 'random'){
      n = rnorm(layers_dims[layer] * layers_dims[layer - 1],
                mean = 0,
                sd = 1) * 0.01

      layerParams[[paste('W', layer - 1, sep = "")]] =
        matrix(n,
               nrow = layers_dims[layer],
               ncol = layers_dims[layer - 1])
      layerParams[[paste('b', layer - 1, sep = "")]] =
        matrix(rep(0, layers_dims[layer]),
               nrow = layers_dims[layer],
               ncol = 1)
    }
    else if(initialization == 'He'){
      n = rnorm(layers_dims[layer] * layers_dims[layer - 1], mean = 0, sd = 1) *
        sqrt(2/layers_dims[layer - 1])

      layerParams[[paste('W',layer - 1, sep = "")]] =
        matrix(n,
               nrow = layers_dims[layer],
               ncol = layers_dims[layer - 1])
      layerParams[[paste('b',layer - 1,sep = "")]] =
        matrix(rep(0, layers_dims[layer]),
               nrow = layers_dims[layer],
               ncol = 1)
    }
    else if(initialization == 'Xavier'){
      n = rnorm(layers_dims[layer] * layers_dims[layer - 1], mean = 0, sd = 1) *
        sqrt(1 / layers_dims[layer - 1])

      layerParams[[paste('W',layer - 1, sep = "")]] =
        matrix(n,
               nrow = layers_dims[layer],
               ncol = layers_dims[layer - 1])
      layerParams[[paste('b',layer - 1, sep = "")]] =
        matrix(rep(0, layers_dims[layer]),
               nrow = layers_dims[layer],
               ncol = 1)
    }
  }

return(layerParams)
}
```

We are now ready to compute our activations in the hidden layers. The activation of the previous layer is defined by the parameter `A_prev`. For the first hidden layer, `A_prev` is the input X values.

For the in-between hidden layers it is either `relu` or the `tanh` activations, depending on our design of the network.

The output layer will have `sigmoid` activation for binary classification and `softmax` activation for multi-class classification.

The linear function Z is the weighted combination of the inputs, defined as $Z^{[l]} = W^{[l]}A^{[l]} + b^{[l]}$, where $A^{[1]} = X$.

The linear function Z is activated by the activation function as

- $tanh(Z) = tanh(WA + b)$.
- $relu(Z) = pmax(WA + b, 0)$.

This function returns the activation at layer $(A_{prev} + 1)$ and a cache containing the values (X, W, b, Z)

```
f_prop_helper <- function(A_prev, W, b, hidden_layer_act){

  Z <-sweep(W %*% A_prev, 1, b, '+')

  forward_cache <- list("A_prev" = A_prev, "W" = W, "b" = b)

  if(hidden_layer_act == "sigmoid"){
    act_values = sigmoid(Z)
  }
  else if (hidden_layer_act == "relu"){
    act_values = relu(Z)
  }
  else if(hidden_layer_act == 'tanh'){
    act_values = tanh(Z)
  }
  else if(hidden_layer_act == 'softmax'){
    act_values = softmax(Z)
  }
  cache <- list("forward_cache" = forward_cache,
                "activation_cache" = act_values[['Z']])

return(list("A" = act_values[['A']], "cache" = cache))
}
```

Having computed the activation at each layer, we are now in a position to compute the forward propagation for all the layers. The length of the parameters defined by `L` is the number of layers (except the input layer).

The inputs to the function are X (input features), W, b (weights and biases). The function returns the activation AL (output layer) and a cache containing the values of (X, W, b, Z) in a list.

```
forward_prop <- function(X, parameters, hidden_layer_act, output_layer_act) {

    caches <- list()
    A <- X
    L <- length(parameters)/2

    # Loop through from layer 1 to upto layer L-1
    for (l in 1:(L - 1)) {
        A_prev <- A
        W <- parameters[[paste("W", l, sep = "")]]
        b <- parameters[[paste("b", l, sep = "")]]
        actForward <- f_prop_helper(A_prev, W, b, hidden_layer_act[[l]])
        A <- actForward[["A"]]
        caches[[l]] <- actForward
    }
```

```
    W <- parameters[[paste("W", L, sep = "")]]
    b <- parameters[[paste("b", L, sep = "")]]

    actForward = f_prop_helper(A, W, b, output_layer_act)
    AL <- actForward[["A"]]
    caches[[L]] <- actForward

    return(list(AL = AL, caches = caches))
}
```

Forward propagation is akin to a prediction function. When the data set has been passed through the layers in the network, we get an output that can be compared to the actual label.

The purpose of the cost function is to determine how far the output is from the target value.

We will only consider the `sigmoid` and `softmax` activation for the output layer, for reasons stated in Sect. 2.6.

```
compute_cost <- function(AL, X, Y, num_classes, output_layer_act){

  if(output_layer_act == "sigmoid"){
    m = length(Y)
    cross_entropy_cost = -(sum(Y * log(AL) + (1 - Y) * log(1 - AL))) / m
  }
  else if(output_layer_act == "softmax"){
    m = ncol(X)
    y.mat <- matrix(Y, ncol = 1)
    y <- matrix(0, nrow = m, ncol = num_classes)
    for (i in 0:(num_classes - 1)) {
      y[y.mat[, 1] == i, i+1] <- 1
    }
    correct_logprobs <- -log(AL)
    cross_entropy_cost <- sum(correct_logprobs * y) / m
  }

return(cross_entropy_cost)
}
```

The notation commonly used in deep learning coding to represent the gradients is

- $dW1 = \frac{\partial \mathcal{J}}{\partial W_1}$.
- $db1 = \frac{\partial \mathcal{J}}{\partial b_1}$.
- $dW2 = \frac{\partial \mathcal{J}}{\partial W_2}$.
- $db2 = \frac{\partial \mathcal{J}}{\partial b_2}$.

We now need to backpropagate through the layers, i.e., calculate the gradients of the weights dW, the bias db and the linear function dZ of each layer with respect to the loss. Backpropagation is used to calculate the error contribution of each weight with respect to cost. The idea is to backward engineer the derivative or slope of every computation and update the weights so that the cost will decrease at each iteration.

We first calculate $dZ^{[L]}$, which will give us $dW^{[L]}$ and $db^{[L]}$.

Having calculated $dZ^{[\ell]} = \frac{\partial \mathcal{J}}{\partial Z^{[\ell]}}$, we would want to find $dW^{[\ell]}$, $db^{[\ell]}$, $dA^{[\ell-1]}$.

The general equation for calculating the gradients are the following:

- $dW^{[\ell]} = \frac{\partial \mathcal{J}}{\partial W^{[\ell]}} = \frac{1}{m} dZ^{[\ell]} A^{[\ell-1]T}$
- $db^{[\ell]} = \frac{\partial \mathcal{J}}{\partial b^{[\ell]}} = \frac{1}{m} \sum_{i=1}^{m} dZ^{[\ell](i)}$
- $dA^{[\ell-1]} = \frac{\partial \mathcal{J}}{\partial A^{[\ell-1]}} = W^{[\ell]T} dZ^{[\ell]}$

```
back_prop_helper  <- function(dA, cache, Y, hidden_layer_act, num_classes){

  forward_cache <-cache[['forward_cache']]
  # Get Z
  activation_cache <- cache[['activation_cache']]
  A_prev <- forward_cache[['A_prev']]
  m = dim(A_prev)[2]

  if(hidden_layer_act == "relu"){
    dZ <- derivative_relu(dA, activation_cache)
  }
  else if(hidden_layer_act == "sigmoid"){
    dZ <- derivative_sigmoid(dA, activation_cache)
  }
  else if(hidden_layer_act == "tanh"){
    dZ <- derivative_tanh(dA, activation_cache)
  }
  else if(hidden_layer_act == "softmax"){
    dZ <- derivative_softmax(dAL, activation_cache, X, Y, num_classes)
  }

  W <- forward_cache[['W']]
  b <- forward_cache[['b']]
  m = dim(A_prev)[2]

  if(hidden_layer_act == 'softmax'){
    dW = 1 / m * t(dZ) %*% t(A_prev) #+ (lambd / m) * W
    db = 1 / m * colSums(dZ)
    dA_prev = t(W) %*% t(dZ)
  }
  else{
  dW = 1 / m * dZ %*% t(A_prev)
  db = 1 / m * rowSums(dZ)
  dA_prev = t(W) %*% dZ
  }

return(list("dA_prev" = dA_prev, "dW" = dW, "db" = db))
}
```

In the function `forward_prop`, we had stored (X, W, b, Z) in the cache. We will use these values in `back_prop`, to calculate the gradients. We will iterate through all the hidden layers backward, starting from the final layer L. At each layer, we will use the cached values for that layer to backpropagate through that layer.

We know that the output at layer L is $A^{[L]} = \sigma(Z^{[L]})$. We will have to calculate the post activation gradient $dAL = \frac{\partial \mathcal{J}}{\partial A^{[L]}}$. For `sigmoid` activation at the output layer, the derivative $dAL = \frac{\partial \mathcal{J}}{\partial A^{[L]}}$ is $-(\frac{Y}{AL} - \frac{(1-Y)}{(1-AL)})$.

```
back_prop <- function(AL,
                      Y,
                      caches,
                      hidden_layer_act,
                      output_layer_act,
                      num_classes){

  gradients = list()
  L = length(caches)
  m = dim(AL)[2]

  if(output_layer_act == "sigmoid"){
    dAL = -((Y/AL) - (1 - Y)/(1 - AL))
  }
  else if(output_layer_act == 'softmax') {
    y.mat <- matrix(Y, ncol = 1)
    y <- matrix(0, nrow = length(Y), ncol = num_classes)
    for (i in 0:(num_classes - 1)) {
      y[y.mat[, 1] == i, i + 1] <- 1
    }
    dAL = (AL - y)
  }

  current_cache = caches[[L]]$cache
  loop_back_vals <-  back_prop_helper(dAL, current_cache,
                                      Y,
                                      hidden_layer_act = output_layer_act,
                                      num_classes)

  gradients[[paste("dA", L, sep = "")]] <- loop_back_vals[['dA_prev']]
  gradients[[paste("dW", L, sep = "")]] <- loop_back_vals[['dW']]
  gradients[[paste("db", L, sep = "")]] <- loop_back_vals[['db']]

  for(l in (L - 1):1){
    current_cache = caches[[l]]$cache
    loop_back_vals = back_prop_helper(gradients[[paste('dA', l + 1, sep = "")]],
                                      current_cache,
                                      Y,
                                      hidden_layer_act[[l]],
                                      num_classes)

    gradients[[paste("dA", l, sep = "")]] <- loop_back_vals[['dA_prev']]
    gradients[[paste("dW", l, sep = "")]] <- loop_back_vals[['dW']]
    gradients[[paste("db", l, sep = "")]] <- loop_back_vals[['db']]
  }

  return(gradients)
}
```

Update each of the structural parameters by subtracting the product of the parameter gradient and the learning rate from the parameter for each propagation.

```
update_params <- function(parameters, gradients, learning_rate) {
    L = length(parameters)/2

    for (l in 1:L) {
        parameters[[paste("W", l, sep = "")]] = parameters[[paste("W",
            l, sep = "")]] - learning_rate * gradients[[paste("dW",
            l, sep = "")]]
```

```
        parameters[[paste("b", l, sep = "")]] = parameters[[paste("b",
            l, sep = "")]] - learning_rate * gradients[[paste("db",
            l, sep = "")]]
    }

    return(parameters)
}
```

The `predict_model` function makes use of the `forward_prop` function to compute the final layer activation. The probability threshold is kept at 0.5 (the threshold however, could be different than 0.5).

```
predict_model <- function(parameters, X, hidden_layer_act, output_layer_act){

  pred <- numeric()
  scores <- forward_prop(X,
                         parameters,
                         hidden_layer_act,
                         output_layer_act)[['AL']]

  if(output_layer_act == 'softmax') {
    pred <- apply(scores, 1, which.max)
  }
  else{
  for(i in 1:ncol(scores)){
    if(scores[i] > 0.5) pred[i] = 1
    else pred[i] = 0
    }
  }

return (pred)
}
```

We are now ready to combine the above functions to write a n layer deep neural network algorithm.

```
n_layer_model <- function(X,
                          Y,
                          X_test,
                          Y_test,
                          layers_dims,
                          hidden_layer_act,
                          output_layer_act,
                          learning_rate,
                          num_iter,
                          initialization,
                          print_cost = F){

  set.seed(1)
  costs <- NULL
  parameters <- initialize_params(layers_dims, initialization)
  num_classes <- length(unique(Y))
  start_time <- Sys.time()

  for( i in 0:num_iter){
      AL = forward_prop(X,
                        parameters,
                        hidden_layer_act,
```

```
                        output_layer_act)[['AL']]

    caches = forward_prop(X,
                          parameters,
                          hidden_layer_act,
                          output_layer_act)[['caches']]

    cost <- compute_cost(AL,
                         X,
                         Y,
                         num_classes,
                         output_layer_act)

    gradients = back_prop(AL,
                          Y,
                          caches,
                          hidden_layer_act,
                          output_layer_act,
                          num_classes)

    parameters = update_params(parameters,
                               gradients,
                               learning_rate)

    costs <- c(costs, cost)

    if(print_cost == T & i %% 1000 == 0){
      cat(sprintf("Cost after iteration %d = %05f\n", i, cost))
    }
}

    if(output_layer_act != 'softmax'){
      pred_train <- predict_model(parameters,
                                  X,
                                  hidden_layer_act,
                                  output_layer_act)
      Tr_acc <- mean(pred_train == Y) * 100
      pred_test <- predict_model(parameters,
                                 X_test,
                                 hidden_layer_act,
                                 output_layer_act)
      Ts_acc <- mean(pred_test == Y_test) * 100
      cat(sprintf("Cost after iteration  %d, = %05f;
                  Train Acc: %#.3f, Test Acc: %#.3f, \n",
                  i, cost, Tr_acc, Ts_acc))
      }
      else if(output_layer_act == 'softmax'){
        pred_train <- predict_model(parameters,
                                    X,
                                    hidden_layer_act,
                                    output_layer_act)
        Tr_acc <- mean((pred_train - 1) == Y)
        pred_test <- predict_model(parameters,
                                   X_test,
                                   hidden_layer_act,
                                   output_layer_act)
        Ts_acc <- mean((pred_test - 1) == Y_test)
        cat(sprintf("Cost after iteration , %d, = %05f;
            Train Acc: %#.3f, Test Acc: %#.3f, \n",
```

```
            i, cost, Tr_acc, Ts_acc))
        }

          end_time <- Sys.time()
          cat(sprintf("Application running time: %#.3f minutes",
                      (end_time - start_time) / 60 ))

return(list("parameters" = parameters, "costs" = costs))
}
```

How many layers, nodes, and layers? There is no specific rule to estimate the number of hidden layers and nodes of a neural network, however, the number of hidden nodes is often based on a relationship between

- Number of input and output nodes.
- Amount of training data.
- Complexity of the learning function.
- Type of hidden unit activation function.
- Regularization.

Too few nodes might result in a simple model resulting in a higher error. Too many nodes will overfit the training data and may not generalize well with unseen data. Typically, the number of hidden nodes should be able to capture 70%–90% of the variance of the input data set. If the NN is a classifier with `sigmoid` activation in the output layer, the output layer has a single node. If it is `softmax` activation, the output layer has one node per class label.

We will use the same image classification data which we used in the previous chapter and use a small network to get our application up and running. Let us see how it performs (Fig. 3.1).

```
layers_dims = c(12288, 3, 1)

two_layer_model = n_layer_model(trainx,
                                trainy,
                                testx,
                                testy,
                                layers_dims,
                                hidden_layer_act = 'tanh',
                                output_layer_act  = 'sigmoid',
                                learning_rate = 0.01,
                                num_iter = 1500,
                                initialization = "random",
                                print_cost = T)
```

```
Cost after iteration 0 = 0.693167
Cost after iteration 1000 = 0.060350
Cost after iteration  1500, = 0.038142;
```

```
                          Train Acc: 99.500, Test Acc: 62.000,
Application running time: 2.834 minutes
```

Let us plot the cost.

Our neural network starts with an initial cost of 0.693 and monotonically reduces to 0.038 after 1500 iterations. The training accuracy of our application is 99.5% and the testing accuracy is 62%, respectively.

Working on our training model above, we will now adjust some hyperparameter settings (i) number of layers, (ii) number of nodes per hidden layer, and (iii) the learning rate.

We will now train a three-layer (deeper) network, with different nodes per hidden layer and a different learning rate; this is our development model.

```
layers_dims = c(12288, 50, 30, 1)

three_layer_model = n_layer_model(trainx,
                                  trainy,
                                  testx,
                                  testy,
                                  layers_dims,
                                  hidden_layer_act = c('relu', 'relu'),
                                  output_layer_act  = 'sigmoid',
                                  learning_rate = 0.045,
                                  num_iter = 1500,
                                  initialization = "random",
                                  print_cost = T)
```

```
Cost after iteration 0 = 0.693108
Cost after iteration 1000 = 0.000377
Cost after iteration  1500, = 0.000200;
                      Train Acc: 100.000, Test Acc: 72.000,
Application running time: 16.995 minutes
```

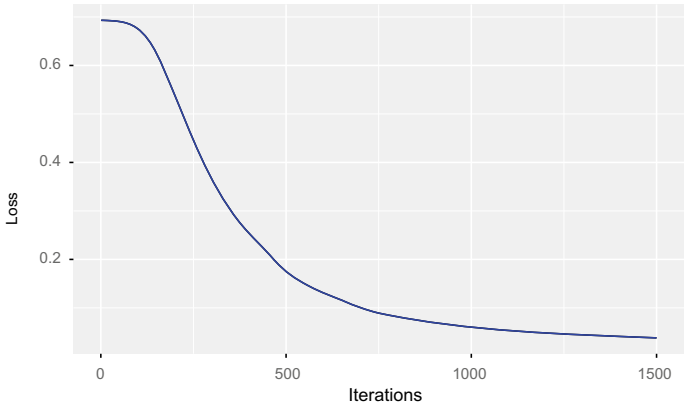

Fig. 3.1 Loss versus iteration for a two-layer neural network with a learning rate of 0.01

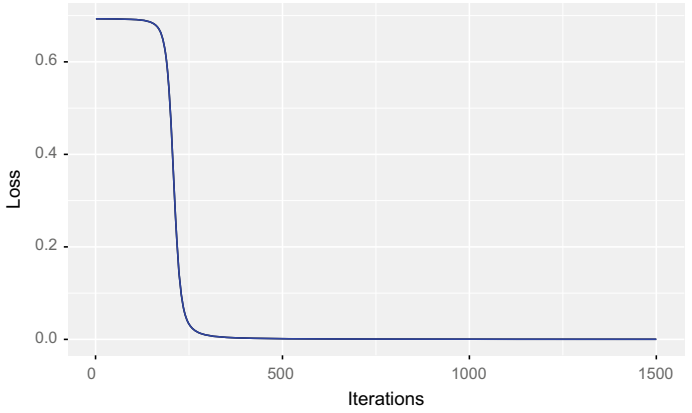

Fig. 3.2 Loss versus iteration for a three-layer neural network with a learning rate of 0.045

Our model happens to give us a better result and the training accuracy has gone up to 100% and the test accuracy has improved by 10% (Fig. 3.2).

Though this is a marginal improvement, we can try to obtain better accuracies by a better initialization of the parameters, searching for the optimal hyperparameters of our network and using better optimization algorithms which we will explore in the subsequent chapters.

Finally, let us train the network using the `softmax` activation in the output layer, and use different activations in the hidden layers. Deeper networks will tend to overfit. Let us see how our deep neural network model performs (Fig. 3.3).

```
layers_dims = c(nrow(trainx), 50, 20, 7, 2)

four_layer_model = n_layer_model(trainx,
                                 trainy,
                                 testx,
                                 testy,
                                 layers_dims,
                                 hidden_layer_act = c('relu', 'relu', 'tanh'),
                                 output_layer_act  = 'softmax',
                                 learning_rate = 0.15,
                                 num_iter = 1500,
                                 initialization = "random",
                                 print_cost = T)
```

```
Cost after iteration 0 = 0.693144
Cost after iteration 1000 = 0.001124
Cost after iteration , 1500, = 0.000510;
              Train Acc: 1.000, Test Acc: 0.640,
Application running time: 17.556 minutes
```

The function `compute_Proba` computes the probability of an image to be a dog. We are using a probability threshold of 50%, i.e., if the computed probability is above the threshold, it predicts the image to be a dog, else it predicts a cat (Fig. 3.4).

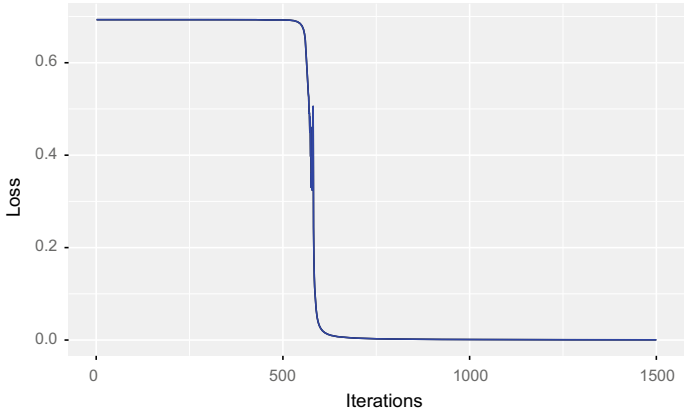

Fig. 3.3 Loss versus iteration for a four-layer neural network with a learning rate of 0.15

Fig. 3.4 Test data images depicting the predicted probabilities and the predicted labels (cat/dog). The errors are labeled in color red

```
compute_Proba <- function(parameters,
                          test_X,
                          hidden_layer_act,
                          output_layer_act){


  score <- forward_prop(test_X,
                        parameters,
                        hidden_layer_act,
                        output_layer_act)[['AL']]
```

```
  Probs <- list(round(score * 100, 2))

  return (Probs)
}

  Prob <- compute_Proba(two_layer_model$parameters,
                        testx,
                        hidden_layer_act = c('relu', 'relu'),
                        output_layer_act = 'sigmoid')

  labels = ifelse(testy == 1, "dog", "cat")


  predicted <- ifelse(
    predict_model(two_layer_model$parameters,
                  testx,
                  hidden_layer_act = c('relu', 'relu'),
                  output_layer_act = 'sigmoid') == 0, 'cat', 'dog')

  error <- ifelse(predicted == labels, 'No', 'Yes')

  index <- c(1:length(labels))

  Probs <- as.vector(unlist(Prob[index]))

  par(mfrow = c(5, 10), mar = rep(0, 4))

  for(i in 1:length(index)){
    image(t(apply(matrix(as.matrix(testx[, index[i]]),
                         c(64, 64, 3),
                         byrow = TRUE), 1, rev)),
          method = 'raster',
          col = gray.colors(12),
          axes = F)
      legend("topright", legend = predicted[i],
             text.col = ifelse(error[i] == 'Yes', 2, 4),
             bty = "n",
             text.font = 1.5)
      legend("bottomright", legend = Probs[i], bty = "n", col = "white")
 }
```

3.2 Overview of Packages for Deep Learning in R

With the rise in popularity of deep learning with R, CRAN has introduced quite a few deep learning packages. An overview of these packages are presented in Table 3.1 below.

In addition, there are two new packages `kerasR` and `keras`. Both these packages provide an R interface to the Python deep learning package `Keras`, which offers Python users a high-level neural networks API, written in Python and is capable of running on top of either TensorFlow, Microsoft Cognitive Toolkit (CNTK), or Theano.

Table 3.1 Different R packages for deep learning

R. Package	Description
nnet	Package for feedforward neural networks with a single hidden layer and multinomial log-linear models
neuralnet	Training of neural networks using backpropagation
h20	Scripting functionality for H2O
RSNNS	Interface to the Stuttgart Neural Network Simulator (SNNS)
tensorflow	Interface to TensorFlow
deepnet	Deep learning toolkit in R
darch	Package for Deep Architectures and Restricted Boltzmann Machines
rnn	Package to implement Recurrent Neural Networks
FCNN4R	Interface to the FCNN library that allows user-extensible ANNs
rcppDL	Implementation of basic machine learning methods with many layers, including dA (Denoising Autoencoder), SdA (Stacked Denoising Autoencoder), RBM (Restricted Boltzmann machine), and DBN (Deep Belief Nets)
deepr	Package to streamline the training, fine-tuning and predicting processes for deep learning based on darch and deepnet
MXNetR	Package for flexible and efficient GPU computing and state-of-the-art deep learning in R

https://cran.r-project.org/web/views/MachineLearning.html

`keras`, is an interface to the Python `Keras` allowing us to enjoy the benefit of `R` programming while having access to the capabilities of the Python `Keras` package.

3.3 Introduction to `keras`

Keras means horn in Greek. It is a reference to a literary image from ancient Greek and Latin literature, first found in the *Odyssey*, where dream spirits (*Oneiroi*, singular *Oneiros*) are divided between those who deceive men with false visions who arrive on Earth through a gate of ivory and those who announce a future that will come to pass and arrive through a gate of horn.

`keras` was initially developed as part of the research effort of project *ONEIROS* (Open-ended Neuro-Electronic Intelligent Robot Operating System). It is a high-level neural networks API developed to enable fast experimentation, and it supports multiple backends, like `TensorFlow`, Microsoft Cognitive Toolkit (`CNTK`) and `Theano`. It has a built-in support for CNNs and RNNs and any combination of both.

The `keras` `R` package is compatible with `R` versions 3.2 and higher.

3.3.1 Installing keras

The keras package uses the Python Keras library. The package can be installed in R using the install.packages("keras").

```
install.packages("keras")
```

The Keras to R interface uses the TensorFlow *backend engine* by default. To install both the core Keras library as well as the TensorFlow backend, we use the install_keras() function.

```
library(keras)
install_keras()
```

A typical keras workflow in R includes

- Defining the data.
- Defining the network.
- Selecting the loss function, the optimizer, and the metrics to be monitored.
- Fit the model.

3.3.2 Pipe Operator in R

The keras package uses the pipe operator (% > %) to connect functions or operations together. In the kerasR package, the pipe operator is replaced with the $ operator.

We will be using the pipe (% > %) operator from the magrittr package in R to add layers in the network. The pipe operator is very useful as it allows us to pass the value on its left as the first argument to the function on its right.

We would need to fit the model to our data. In this case, we train our model for 50 epochs over all the samples in batches of 32 samples, respectively.

3.3.3 Defining a keras Model

We will be using same data consisting of pictures of cat and dogs. In the target variable, a cat image is reflected with a value of 0 and a dog image as 1 value, respectively.The next step is to define our network.

We will use a fully connected neural network, i.e., build a stack of fully connected layers to all nodes. We will use the same activations, which we used in our application, i.e., relu for the the hidden layers and the sigmoid activation for the output layer so that the output is between 0 and 1, depicting predicted probabilities. The model needs to know the input shape (features) it should expect. The first layer, therefore,

in a sequential model needs to be fed this information. The following layers will automatically infer the shape.

Our hidden layers with relu activation will have 100, 50, and 15 nodes respectively. The input layer has 12288 nodes (i.e., features).

To start constructing a model, we have to first initialize a sequential model with the help of the `keras_model_sequential()` function. The sequential model is a linear stack of layers. We create a sequential model by calling the `keras_model_sequential()` function then a series of layer functions. We can use the `summary()` function to print a summary representation of our model. You may use `help(package = keras)` for more details.

In practice, the output layer consist of 1 neuron for regression and binary classification problems and n neurons for a multi-class classification, where n is the number of classes in the target variable. We also specify the activation function in each layer. Keras supports a number of activation functions such as `relu`, `softmax`, `sigmoid`, `tanh`, `elu` (exponential linear unit) among others. There are many other features and arguments that we can use when configuring, compiling, and fitting our model.

3.3.4 *Configuring the* `keras` *Model*

Before compiling the model, we have to specify the loss objective function. Available functions include, among others, `mean_squared_error` for regression problems and `binary_crossentropy` and `categorical_crossentropy` for binary and multi-class classifications, respectively. We will use the `categorical_crossentropy` as our loss function.

The second required argument is the optimizer for estimating and updating the model parameters. Keras support several optimizers such as Stochastic gradient descent (`sgd`) optimizer, Adaptive Moment Estimation (`adam`), `rmsprop`, Adaptive learning rate (`Adadelta`) among others. While calling the optimizer, we can specify the learning rate (`lr`), learning rate decay over each update (`decay`), `momentum`, among other arguments specific to each optimizer. We will use the `adam` optimizer in our Keras model.

We also need to define a metric on which the model will be evaluated. We will use the "accuracy" metric to assess the performance of the model.

3.3.5 *Compile and Fit the Model*

Now that we have set up the architecture of our model, it is time to compile and fit the model to the data. In the crucial step of fitting the model, we specify the epochs—the number of times the algorithm "sees" the entire training data—and the batch size, i.e., the size of sample to be passed through the algorithm in each epoch. A training

sample of size 10, for example, with batch size = 2, will give 5 batches and hence 5 iterations per epoch. If epoch = 4, then we have 20 iterations for training. If we use too few epochs, we run the risk of underfitting and too many may risk overfitting. The early stopping function helps the model from overfitting.

```
set.seed(1)
model <- keras_model_sequential()
model %>%
  layer_dense(units = 100, activation = 'relu', input_shape = c(12288)) %>%
  layer_dense(units = 50, activation = 'relu') %>%
  layer_dense(units = 15, activation = 'relu') %>%
  layer_dense(units = 1, activation = 'sigmoid') %>%
  compile(
    optimizer = optimizer_adam(),
    loss = 'binary_crossentropy',
    metrics = c('accuracy')
  )
summary(model)
```

```
_________________________________________________________________________
Layer (type)                 Output Shape              Param #
=========================================================================
dense_1 (Dense)              (None, 100)               1228900
_________________________________________________________________________
dense_2 (Dense)              (None, 50)                5050
_________________________________________________________________________
dense_3 (Dense)              (None, 15)                765
_________________________________________________________________________
dense_4 (Dense)              (None, 1)                 16
=========================================================================
Total params: 1,234,731
Trainable params: 1,234,731
Non-trainable params: 0
_________________________________________________________________________
```

```
model %>% fit(t(trainx), trainy, epochs = 50, batch_size = 32,
              validation_split = 0.2)
history <- model %>% fit(t(trainx), trainy, epochs = 50, batch_size = 32,
                         validation_split = 0.2)
model %>% evaluate(t(trainx), trainy)
```

```
$loss
[1] 0.7409931

$acc
[1] 0.86
```

We can use the following functions to inspect our model:

- get_config(model)—returns a list that contains the configuration of the model.
- get_layer(model)—return the layer configuration.
- model$layers—returns a list of the model's layers.
- model$inputs—returns a list of the input tensors.

- model$outputs—returns a list of the output tensors.

Let us predict our data on the training and test sets.

```
keras_pred_train <- model %>% predict_classes(t(trainx))
table(Predicted = keras_pred_train, Actual = trainy)
```

```
          Actual
Predicted   0   1
        0 100  28
        1   0  72
```

```
prob <- model %>% predict_proba(t(trainx))

model %>% evaluate(t(testx), (testy))
```

```
$loss
[1] 1.093653
```

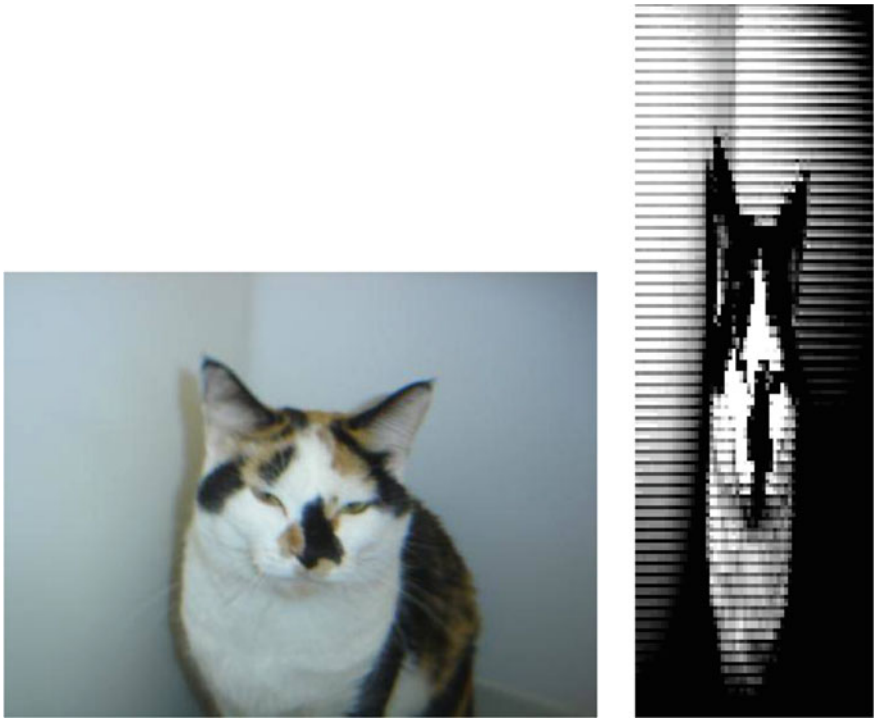

Fig. 3.5 One of the images which both the applications could not label correctly. The left-hand image is the actual image and the right-hand image is the resized image, which models. The resized image could be difficult to identify as a cat or dog, by most humans

```
$acc
[1] 0.62
```

```
# keras evaluation on the test data set
keras_pred_test <- model %>% predict_classes(t(testx))
table(Predicted = keras_pred_test, Actual = testy)
```

```
         Actual
Predicted  0  1
        0 19 12
        1  7 12
```

Let us find out which images were not predicted by the keras model and our application.

```
hidden_layer_act <- c("relu", "relu")
output_layer_act <- "sigmoid"

app_pred_test <- predict_model(three_layer_model$parameters,
    testx, hidden_layer_act, output_layer_act)

table(app_pred = app_pred_test, actual = testy)
```

```
        actual
app_pred  0  1
       0 17  5
       1  9 19
```

```
which(keras_pred_test != testy)
```

```
 [1]  5  6 12 15 20 22 23 28 30 32 34 35 38 39 43 44 45 46 48
```

```
which(app_pred_test != testy)
```

```
 [1]  5 10 12 13 15 20 22 23 24 28 29 32 42 46
```

It appears that our three_layer_model returns a higher accuracy than the keras model.

It also turns out that image number 15 was not predicted by both the models. Let us have a look at this image (Fig. 3.5).

```
images <- list.files(file_path_test)
img <- readImage(file.path(file_path_test, images[15]))

str(testx[[15]])  # actual image
```

```
 num -1.49
```

```
str(matrix(as.matrix(testx[, 15]), c(64, 64, 3), byrow = TRUE))  # resized image
```

```
 num [1:64, 1:192] 0.625 0.629 0.629 0.629 0.607 ...
```

```
EBImage::display(img, method = "raster")
EBImage::display(matrix(as.matrix(testx[, 15]), c(64, 64, 3),
    byrow = TRUE), method = "raster")
```

Indeed both the `keras` and our `three_layer_model` could not figure out whether the blurry image was a cat!

3.4 Conclusion

We have created our own neural network application and successfully used it to classify images. We have also installed `keras` and used the API to model a neural network application.

In the next chapter, we will see how we can initialize the parameters and the role it plays to improve the convergence rate and the accuracy of the neural network.

Chapter 4
Initialization of Network Parameters

A thought is a great big vector of neural activity and they have causal powers.

Geoffrey Hinton

Abstract In this section, we will learn how initialization of the parameters affects a neural network model. We will explore different initialization techniques and visualize the results.

We will be using the following `R` packages:

```
library(ggplot2)
library(gridExtra)
library(InspectChangepoint)
```

4.1 Initialization

Weight initialization can have a profound impact on both the **convergence** rate and the **accuracy** of our network. While working with deep neural networks, initializing the network with the correct weights can make the difference between the network converging in a reasonable time and the loss function "oscillating", and not going anywhere even after thousands of iterations.

To understand why is initialization a problem, let us consider the sigmoid function represented in Fig. 4.1. The `sigmoid` activation function is approximately linear when we are close to zero (represented by the red line in the figure). This means that as the weights tend toward zero, **there will not be any nonlinearity**, which goes against the very ethos and advantage of deep layer neural networks.

© Springer Nature Singapore Pte Ltd. 2019

A. Ghatak, *Deep Learning with R*,
https://doi.org/10.1007/978-981-13-5850-0_4

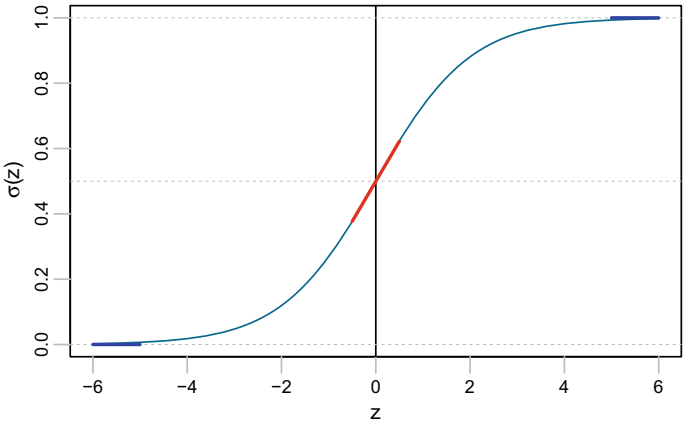

Fig. 4.1 The sigmoid function becomes "flat", at the extremes and "linear" in the middle

If the weights are too small, then the variance of the input signal starts diminishing as it passes through each layer in the network and eventually drops to a very low value, which is no longer useful.

If the weights are too large, then the variance of input data tends to increase with each passing layer and at some point of time, it becomes very large. For very large values, the `sigmoid` function tends to become flat (represented by the blue line in the figure) as we can see in Fig. 4.1. This means that our activations will become saturated and the gradients will start approaching zero.

Therefore, initializing the network with the right weights is very important if we want our neural network to function properly and make sure that the weights are in a reasonable range before we start training the network.

In order to illustrate this fact, let us use our DNN application with four different weight initializations

(a) Initialize weights to zero.
(b) Initialize weights to a random normal distribution $\mathcal{N}(0, \mu = 0, \sigma = 1)$.
(c) The weights are initialized to a random normal distribution but inversely proportional to the square root of the number of neurons in the previous layer (`Xavier` initialization).
(d) Initialize weights to a random normal distribution but inversely proportional to the square root of the number of neurons in the previous layer and directly proportional to the square root of 2 (`He` initialization).

A well-designed initialization can **speedup the convergence** of gradient descent and increase the chances of a lower training and generalization error. Let us go through four different types of parameter initializations and what difference do they make on the cost and convergence.

Before we do that, let us create a spirally distributed planar data set.

```
N <- 400  # number of points per class
D <- 2  # dimensionality
K <- 2  # number of classes
X <- data.frame()  # data matrix (each row = single example)
Y <- data.frame()  # class labels

set.seed(308)

for (j in (1:2)) {
    r <- seq(0.05, 1, length.out = N)  # radius
    t <- seq((j - 1) * 4.7, j * 4.7, length.out = N) + rnorm(N,
        sd = 0.3)  # theta
    Xtemp <- data.frame(x = r * sin(t), y = r * cos(t))
    ytemp <- data.frame(matrix(j, N, 1))
    X <- rbind(scale(X), Xtemp)
    Y <- rbind(Y, ytemp)
}

data <- cbind(X, Y)
colnames(data) <- c(colnames(X), "label")

x_min <- min(X[, 1]) - 0.2
x_max <- max(X[, 1]) + 0.2
y_min <- min(X[, 2]) - 0.2
y_max <- max(X[, 2]) + 0.2
```

```
ggplot(data) + geom_point(aes(x = x,
                              y = y,
                              color = as.character(label)),
                          size = 1) +
  theme_bw(base_size = 15) +
  xlim(x_min, x_max) +
  ylim(y_min, y_max) +
  coord_fixed(ratio = 0.8) +
  theme(axis.ticks=element_blank(),
        panel.grid.major = element_blank(),
        panel.grid.minor = element_blank(),
        axis.text=element_blank(),
        axis.title=element_blank(),
        legend.position = 'none')
```

We will split the data set shown in Fig. 4.2, into training and testing data sets (Figs. 4.3 and 4.4).

```
indexes <- sample(1:800, 600)
train_data <- data[indexes, ]
test_data <- data[-indexes, ]
trainX <- train_data[, c(1, 2)]
trainY <- train_data[, 3]
testX <- test_data[, c(1, 2)]
testY <- test_data[, 3]
trainY <- ifelse(trainY == 1, 0, 1)
testY <- ifelse(testY == 1, 0, 1)
```

We need to be sure about the dimensions of the training set data and test set data.

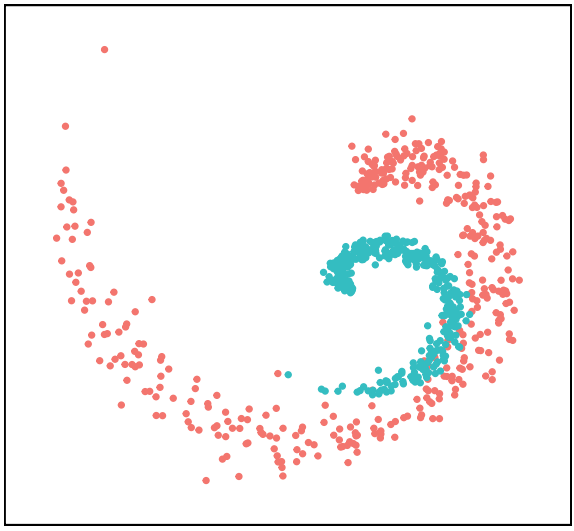

Fig. 4.2 A spiral planar data set created to visualize the effects of different parameter initializations

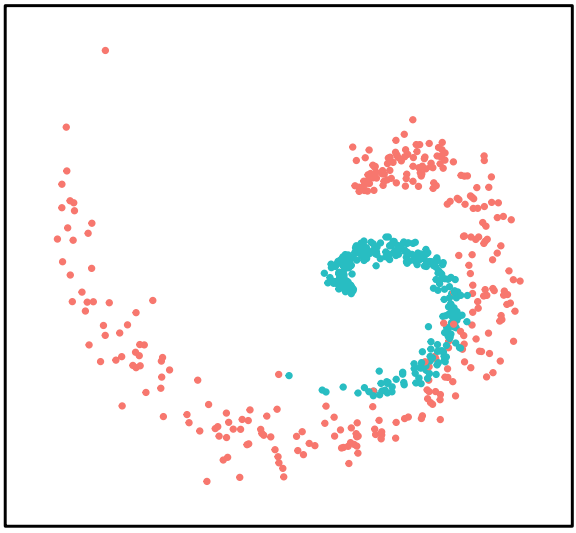

Fig. 4.3 Spiral planar training data set

```
dim(train_data)
```

```
[1] 600   3
```

```
dim(test_data)
```

```
[1] 200   3
```

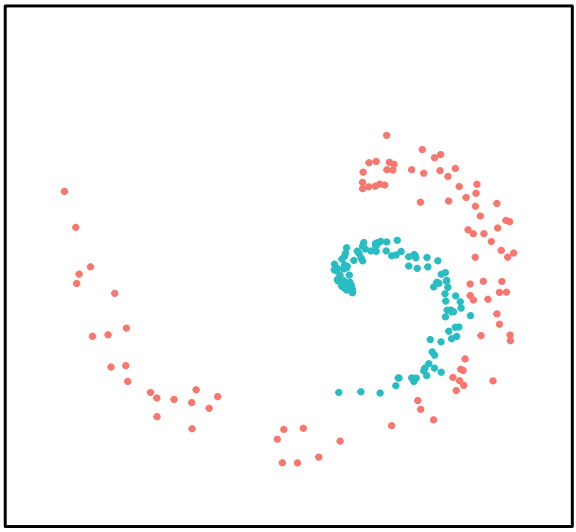

Fig. 4.4 Spiral planar testing data set

4.1.1 Breaking Symmetry

During forward propagation, each unit in the hidden layer gets signal $a_i = \sum_i^n W \cdot x_i$. When we initialize all the weights to the same value, each hidden unit receives the same signal, i.e., if all the weights are initialized to a number n, each unit gets the same signal every time and would not be able to converge during gradient descent.

If all weights are initialized randomly, then each time the model would search for a different path to converge and there would be a better chance to find the global minima. This is what is meant by **breaking symmetry**. The initialization is asymmetric, i.e., it is different and the optimization algorithm will find different solutions to the same problem, thereby seeking a faster convergence.

4.1.2 Zero Initialization

Zero initialization does not serve any purpose because the neural network model does not perform symmetry breaking. If we set all the weights to be zero, then all the neurons of all the layers perform the same calculation, giving the same output. When the weights are zero, the complexity of our network reduces to that of a single neuron.

```
layers_dims <- c(2, 100, 1)

init_zero <- n_layer_model(t(trainX), trainY, t(testX), testY,
    layers_dims, hidden_layer_act = "relu", output_layer_act = "sigmoid",
```

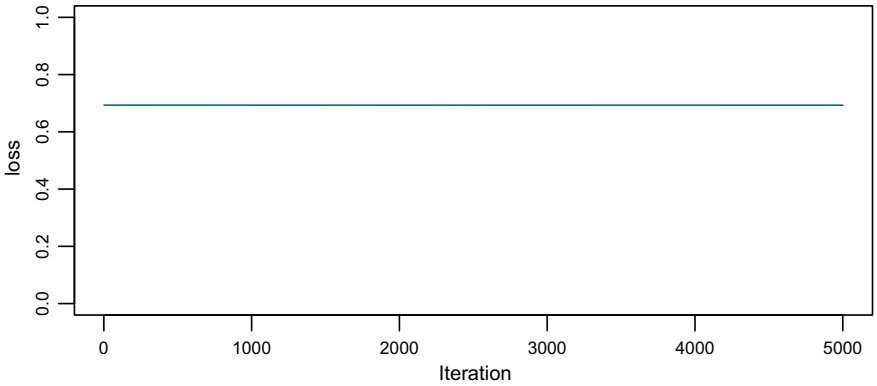

Fig. 4.5 Zero initialization: loss versus iteration

```
    learning_rate = 0.03, num_iter = 5000, initialization = "zero",
    print_cost = T)
```

```
Cost after iteration 0 = 0.693147
Cost after iteration 1000 = 0.693008
Cost after iteration 2000 = 0.693008
Cost after iteration 3000 = 0.693008
Cost after iteration 4000 = 0.693008
Cost after iteration 5000 = 0.693008
Cost after iteration  5000, = 0.693008;
                    Train Acc: 50.833, Test Acc: 47.500,
Application running time: 43.305 minutes
```

From Fig. 4.5 , we can see that zero initialization serves no purpose. The neural network does not break symmetry.

Let us look at the decision boundary for our zero-initialized classifier on the train and test data sets (Fig. 4.6).

```
step <- 0.01

x_min <- min(trainX[, 1]) - 0.2
x_max <- max(trainX[, 1]) + 0.2
y_min <- min(trainX[, 2]) - 0.2
y_max <- max(trainX[, 2]) + 0.2

grid <- as.matrix(expand.grid(seq(x_min, x_max, by = step), seq(y_min,
    y_max, by = step)))
Z <- predict_model(init_zero$parameters, t(grid), hidden_layer_act = "relu",
    output_layer_act = "sigmoid")
Z <- ifelse(Z == 0, 1, 2)

g1 <- ggplot() + geom_tile(aes(x = grid[, 1], y = grid[, 2],
    fill = as.character(Z)), alpha = 0.3, show.legend = F) +
    geom_point(data = train_data, aes(x = x, y = y, color = as.character(trainY)),
```

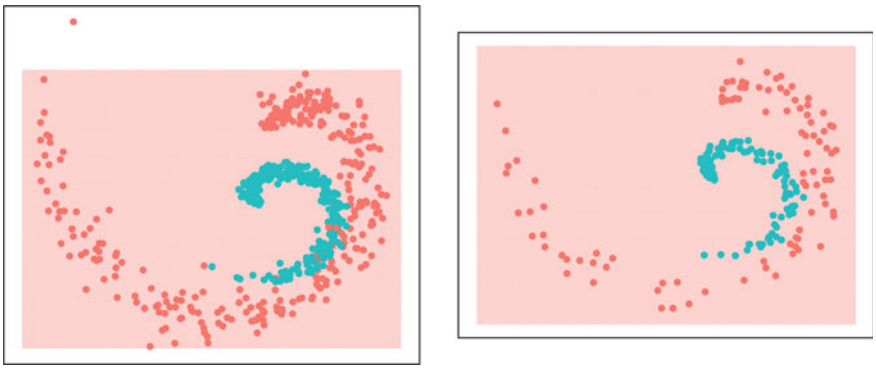

Fig. 4.6 Decision boundary with zero initialization on the training data set (left-hand plot) and the testing data set (right- hand plot)

```
        size = 1) + theme_bw(base_size = 15) + coord_fixed(ratio = 0.8) +
    theme(axis.ticks = element_blank(), panel.grid.major = element_blank(),
        panel.grid.minor = element_blank(), axis.text = element_blank(),
        axis.title = element_blank(), legend.position = "none")

x_min <- min(testX[, 1]) - 0.2
x_max <- max(testX[, 1]) + 0.2
y_min <- min(testX[, 2]) - 0.2
y_max <- max(testX[, 2]) + 0.2

grid <- as.matrix(expand.grid(seq(x_min, x_max, by = step), seq(y_min,
    y_max, by = step)))
Z <- predict_model(init_zero$parameters, t(grid), hidden_layer_act = "relu",
    output_layer_act = "sigmoid")
Z <- ifelse(Z == 0, 1, 2)
g2 <- ggplot() + geom_tile(aes(x = grid[, 1], y = grid[, 2],
    fill = as.character(Z)), alpha = 0.3, show.legend = F) +
    geom_point(data = test_data, aes(x = x, y = y, color = as.character(testY)),
        size = 1) + theme_bw(base_size = 15) + coord_fixed(ratio = 0.8) +
    theme(axis.ticks = element_blank(), panel.grid.major = element_blank(),
        panel.grid.minor = element_blank(), axis.text = element_blank(),
        axis.title = element_blank(), legend.position = "none")

grid.arrange(g1, g2, ncol = 2, nrow = 1)
```

As expected, zero initialization is not able to find any decision boundary.

4.1.3 Random Initialization

One of the ways is to assign the weights from a Gaussian distribution which would have zero mean and some finite variance. This breaks the symmetry and gives better accuracy because now, every neuron is no longer performing the same computation. In this method, the weights are initialized very close to zero.

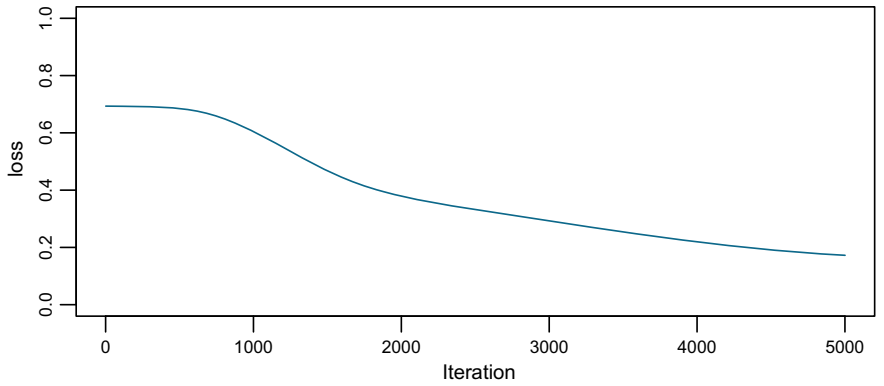

Fig. 4.7 Random initialization loss versus iteration

```
layers_dims <- c(2, 100, 1)

init_random <- n_layer_model(t(trainX),
                             trainY,
                             t(testX),
                             testY,
                             layers_dims,
                             hidden_layer_act = 'relu',
                             output_layer_act  = 'sigmoid',
                             learning_rate = 0.03,
                             num_iter = 5000,
                             initialization = "random",
                             print_cost = T)
```

```
Cost after iteration 0 = 0.693335
Cost after iteration 1000 = 0.603099
Cost after iteration 2000 = 0.378896
Cost after iteration 3000 = 0.293113
Cost after iteration 4000 = 0.218871
Cost after iteration 5000 = 0.172451
Cost after iteration  5000, = 0.172451;
                    Train Acc: 96.333, Test Acc: 97.500,
Application running time: 43.305 minutes
```

The random (Gaussian normal) initialization yields a decreasing cost and returns a training accuracy of 96.3% and a corresponding testing accuracy of 97.5% albeit, it takes 43.3 minutes to converge (Fig. 4.7).

Let us look at the decision boundary and plot our random (Gaussian normal) initialized classifier on the train and test data sets (Fig. 4.8).

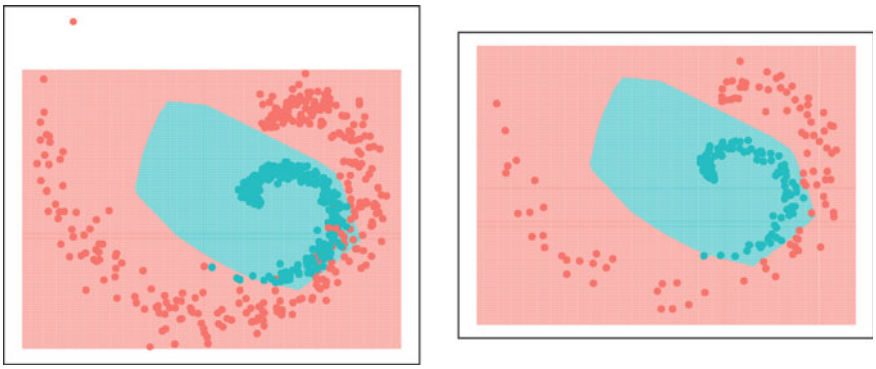

Fig. 4.8 Decision boundary with random initialization on the training data set (left-hand plot) and on the testing data set (right-hand plot)

4.1.4 `Xavier` Initialization

Let us consider a linear neuron represented as $y = w_1x_1 + w_2x_2 + \cdots + w_nx_n$. We would want the variance to remain the same with each passing layer. This helps us to keep the activation values from exploding to a high value or vanishing to zero. We would therefore need to initialize the weights in such a way that the variance remains the same for both x and y, by a process known as `Xavier` initialization. For this purpose, we can write

$$var(y) = var(w_1x_1 + w_2x_2 + \cdots + w_nx_n)$$

The right-hand side of the above equation can be generalized as

$$var(w_ix_i) = E[x_i]^2var(w_i) + E[w_i]^2var(x_i) + var(w_i)var(x_i)$$

Considering that the input values and the weight parameter values are coming from a Gaussian distribution with zero mean, the above equation reduces to

$$var(w_ix_i) = var(x_i)var(w_i)$$

or,

$$var(y) = var(w_1)var(x_1) + var(w_2)var(x_2) + \cdots + var(w_n)var(x_n)$$

Since they are all identically distributed, we can represent the above equation as

$$var(y) = n \times var(w_i) \times var(x_i)$$

If the variance of y needs to be the same as that of x, then the term $n \times var(w_i)$ should be equal to 1, or

$$var(w_i) = \frac{1}{n}$$

We therefore need to pick the parameter weights from a Gaussian distribution with zero mean and a variance of $1/n$, where n is the number of input neurons.

In the original paper, the authors [14], take the average of the number input neurons and the output neurons. So the formula becomes

$$\begin{aligned} var(w_i) &= \frac{1}{n_{avg}} \\ \text{where,} n_{[avg]} &= (n[l-1] + n_{[l]})/2 \\ w_i &= \sqrt{\frac{2}{n^{[l-1]} + n^{[l]}}} \end{aligned} \tag{4.1.1}$$

Since it is computationally expensive to implement, we only take the number of input neurons of the previous layer and therefore, Eq. 4.1.1 can be written as

$$w_i = \sqrt{\frac{2}{n^{[l-1]}}} \tag{4.1.2}$$

```
layers_dims <- c(2, 100, 1)

init_Xavier <- n_layer_model(t(trainX),
                             trainY,
                             t(testX),
                             testY,
                             layers_dims,
                             hidden_layer_act = 'relu',
                             output_layer_act  = 'sigmoid',
                             learning_rate = 0.03,
                             num_iter = 5000,
                             initialization = "Xavier",
                             print_cost = T)
```

```
Cost after iteration 0 = 0.911371
Cost after iteration 1000 = 0.237626
Cost after iteration 2000 = 0.157498
Cost after iteration 3000 = 0.126888
Cost after iteration 4000 = 0.111465
Cost after iteration 5000 = 0.102719
Cost after iteration  5000, = 0.102719;
                    Train Acc: 97.667, Test Acc: 99.000,
Application running time: 1.090 minutes
```

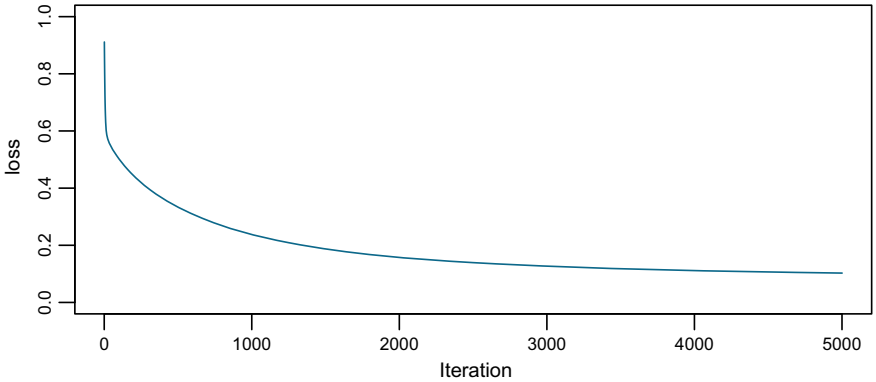

Fig. 4.9 Xavier initialization: loss versus iteration

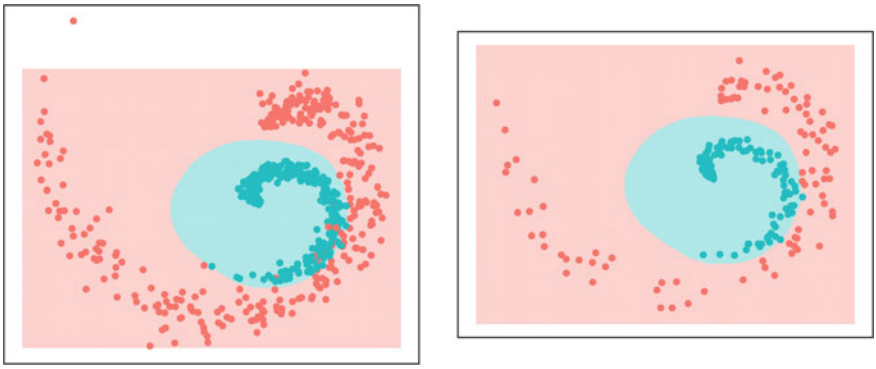

Fig. 4.10 Decision boundary with Xavier initialization on the training data set (left-hand plot) and the testing data set (right- hand plot)

The `Xavier` initialization yields a decreasing cost and returns a higher training accuracy of 97.66% and testing accuracy is 99%. The time to converge has dropped from 55 min to just about a minute (Fig. 4.9). The decision boundary for our data set, using Xavier initialisation is shown in Fig. 4.10.

4.1.5 `He` Initialization

`He` initialization is named after the first author of [15], 2015. The underlying idea behind both, `He` and `Xavier` initialization is to preserve the variance of activation values between layers. In this method, the weights are initialized as a function of the size of the previous layer, which helps in attaining a global minimum of the loss function faster and more efficiently. The weights are still random but differ in range

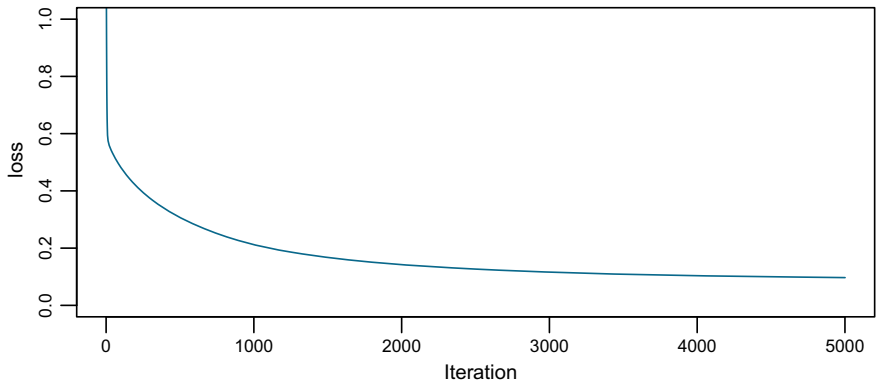

Fig. 4.11 He initialization loss: versus iteration

depending on the size of the previous layer of neurons thus providing a controlled initialization and hence, a faster and more efficient gradient descent. Both the `He` and `Xavier` initialization methods are able to converge faster than random initialization, but with `He` initialization, the errors start to reduce earlier (Fig. 4.11).

```
layers_dims <- c(2, 100, 1)

init_He <- n_layer_model(t(trainX),
                         trainY,
                         t(testX),
                         testY,
                         layers_dims,
                         hidden_layer_act='relu',
                         output_layer_act  = 'sigmoid',
                         learning_rate = 0.03,
                         num_iter = 5000,
                         initialization = "He",
                         print_cost = T)
```

```
Cost after iteration 0 = 1.235932
Cost after iteration 1000 = 0.212091
Cost after iteration 2000 = 0.142321
Cost after iteration 3000 = 0.116169
Cost after iteration 4000 = 0.103672
Cost after iteration 5000 = 0.097017
Cost after iteration  5000, = 0.097017;
                    Train Acc: 97.333, Test Acc: 99.000,
Application running time: 1.038 minutes
```

The `He` initialization yields a steeper decline of the cost and returns a training accuracy of 97.33% with a corresponding testing accuracy of 99%. The decision boundary for our data set, using He initialization is shown in Fig. 4.12. The time to converge is also lower than all other initializations (Fig. 4.13 and Tables 4.1, 4.2).

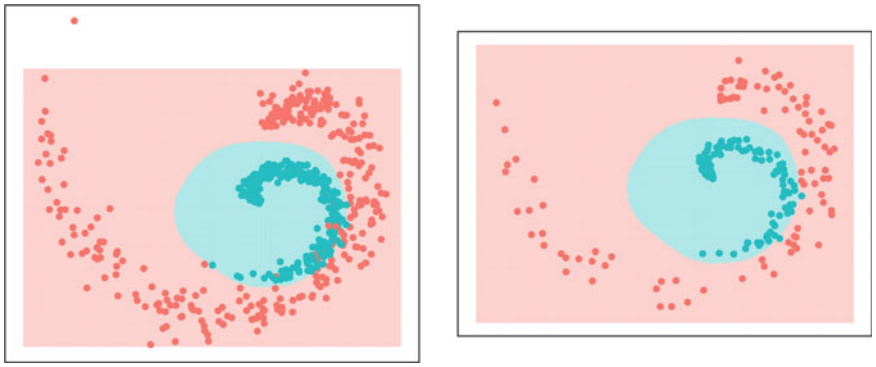

Fig. 4.12 Decision boundary with He initialization on the training data set (left-hand plot) and the testing data set (right- hand plot)

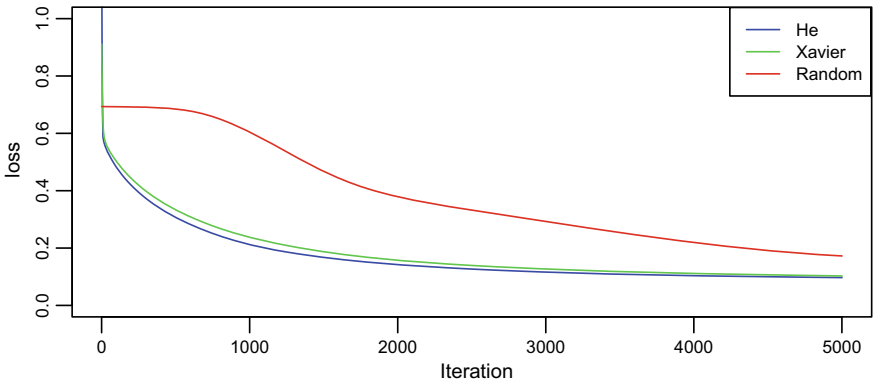

Fig. 4.13 Convergence pattern of different initialization methods

Table 4.1 Time to converge for different initializations

	Random	Xavier	He
Cost	0.172	0.102	0.097
Time (minutes)	43.300	1.090	1.030

There is no specific rule for selecting any specific initialization method, though the `He` initialization works well for networks with `relu` activations. Our learnings are the following:

(a) Different initializations lead to different results.
(b) A well-chosen initialization can speedup the convergence of gradient descent.
(c) A well-chosen initialization method can increase the odds of the gradient descent converging to a lower training (and generalization) error.

Table 4.2 Train and test accuracies with different initializations

Initialization	Train accuracy	Test accuracy
Zero	50.83	47.5
Random	96.30	97.5
Xavier	97.60	99.0
He	97.30	99.0

(d) The parameter weights $W^{[l]}$ should be initialized randomly to break the symmetry and makes sure that different hidden units can learn different things.
(e) We should not initialize the parameter weights to large values.
(f) `He` initialization works best for networks with `relu` activations.
(g) It is appropriate to initialize the biases $b^{[l]}$ to zeros. Symmetry is still broken so long as $W^{[l]}$ is initialized randomly.

4.2 Dealing with NaNs

Having a model which yields NaNs or Infs is quite common if some of the components in the model are not set properly. NaNs are hard to deal with because, it may be caused by a bug or an error in the code or because of the numerical stability in the computational environment (including libraries, etc.). In some cases, it could relate to the algorithm itself. Let us outline some of the common issues which can cause the model to yield NaNs, and some of the ways to get around this problem.

4.2.1 Hyperparameters and Weight Initialization

Most frequently, the cause would be that some of the hyperparameters, especially learning rates, are set incorrectly. A high learning rate can result in NaN outputs so the first and easiest solution is to lower it. One suggested method is to keep halving the learning rate till the NaNs disappear.

The penalty (λ) in the regularization term can also play a part where the model throws up NaN values. Using a wider hyperparameter space with one or two training epochs, each could be tried out to see if the NaNs disappear.

Some models can be very sensitive to the weight initialization. If the weights are not initialized correctly, the model can end up yielding NaNs.

4.2.2 Normalization

Sometimes, this may be obviated by normalizing the input values (though, normalization is a norm which must be strictly followed).

4.2.3 Using Different Activation Functions

Try using other activation functions like tanh. Unlike ReLUs, the outputs from tanh have an upper bound in value and may be a solution. Adding more nonlinearity can also help.

4.2.4 Use of NanGuardMode, DebugMode, or MonitorMode

If adjusting the hyperparameters do not work help, it can be still be sought from Theano's `NanGuardMode`, by changing the mode of the Theano function. This will monitor all input/output variables in each node, and raise an error if NaNs are detected. Similarly, Theano's DebugMode and MonitorMode can also help.

4.2.5 Numerical Stability

This may happen due to zero division or by any operation that is making a number(s) extremely large. Some functions like, $\frac{1}{log(p(x)+1)}$ could result in NaNs for those nodes, which have learned to yield a low probability $p(x)$ for some input x. It is important to find what are the function input values for the given cost (or any other) function are and why we are getting that input. Scaling the input data, weight initialization, and using an adaptive learning rate is some of the suggested solutions.

4.2.6 Algorithm Related

If the above methods fail, there could be a good chance that something has gone wrong in the algorithm. In that case, we need to inspect the mathematics in the algorithm and try to find out if everything has been derived correctly.

4.2.7 NaN Introduced by AllocEmpty

AllocEmpty is used by many operations such as scan to allocate some memory without properly clearing it. The reason for that is that the allocated memory will subsequently be overwritten. However, this can sometimes introduce NaN depending on the operation and what was previously stored in the memory it is working on. For instance, trying to zero out memory using a multiplication before applying an operation could cause a NaN if NaN is already present in the memory, since $0 * \text{NaN} \rightarrow \text{NaN}$.

4.3 Conclusion

We have explored different initialization techniques used in neural networks and learnt how they affect both convergence time and accuracy. In deep neural networks where it takes a long time to train a model, initialization of the parameters makes a big difference.

We have also learnt the hazards of encountering NaN values and how to counter them.

In the next chapter, we will discuss different optimization techniques which will further enhance the performance of the deep neural network models.

Chapter 5
Optimization

In A.I., the holy grail was how do you generate internal representations.

Geoffrey Hinton

Abstract In the previous chapter, we explored how initialization of the parameters affects the outcome of the model. In this chapter, we will explore ways to address the above issues by

- Implementing optimization algorithms— mini-batch gradient descent, momentum, RMSprop, and Adam, and check for their convergence.
- ℓ_2-regularization, dropout regularization, and batch normalization gradient checking.
- How to adjust train/dev/test data sets and analyze bias/variance.
- Use TensorFlow for deep learning.

5.1 Introduction

There are quite a few important and sometimes subtle choices, which we need to make in the process of building and training a neural network. We need to decide which loss function to use, which activations to use, how many layers should we use, how many neurons should we have in the layers, which optimization algorithm is best suited for our network, how do we initialize our parameter weights, choice of the regularization parameter, choice of the learning rate, etc.

We can also include the training time as a hyperparameter, i.e., if the training time exceeds a certain value, we can ask the algorithm to stop iterating over the epochs. This condition is important in real-time models when we do not have the luxury of time.

© Springer Nature Singapore Pte Ltd. 2019

A. Ghatak, *Deep Learning with R*,
https://doi.org/10.1007/978-981-13-5850-0_5

The overarching message is overfitting, the deeper the network we formulate, we have the larger issue of the model overfitting. Deep neural networks can also have the problem of computational cost and convergence due to a phenomenon called vanishing/exploding gradients, and we will discuss how we can overcome this problem.

5.2 Gradient Descent

Up till now, we have been using the simplest update using gradient descent represented as

$$W := W - \alpha \frac{\partial \mathcal{J}}{\partial W} \tag{5.2.1}$$

where α is a fixed hyperparameter called the learning rate to determine how small or large a step should be taken to arrive at the global minima.

As we get closer to the global minima, the derivative of the cost with respect to the weight gets smaller. It turns out that this simple gradient descent update approach can be very slow in finding the global minima after a certain number of iterations.

There are three variants of gradient descent and each differs in the amount of data we use

- Batch Gradient Descent—searches for all the examples for all parameter updates,
- Mini-Batch Gradient Descent—searches for a subset of all the examples over a cycle, so that it includes all the examples in mini-batches,
- Stochastic Gradient Descent—searches for a randomly selected example.

Since we would be dealing with a lot of data, we make a trade-off between the accuracy of the parameter update and the time it takes to perform an update.

5.2.1 Gradient Descent or Batch Gradient Descent

The batch gradient descent computes the gradient of the cost function with respect to the structural parameters w for the entire training data set.

The gradient descent (batch gradient descent) rule is represented as

$$\begin{aligned} &\text{for layers in } 1, 2, \ldots, L \\ W^{[layer]} &= W^{[layer]} - \lambda dW^{[layer]} \\ b^{[layer]} &= b^{[layer]} - \lambda db^{[layer]} \end{aligned} \tag{5.2.2}$$

Since we have to calculate the gradients for the entire data set to perform one update, this method can be very slow and can be computationally expensive, espe-

cially with large data sets. Batch gradient descent also does not allow us to update our model online with new observations streaming in.

The function `update_params` from the previous chapter updates the parameters using batch gradient descent.

5.2.2 Stochastic Gradient Descent

A variant of the batch gradient descent is Stochastic Gradient Descent (SGD), which is a modified version of the batch gradient descent where each batch has a single observation $x^{(i)}$ and label $y^{(i)}$ (i.e., `mini_batch_size`= 1), chosen at random from the training data set, so that the parameters start getting modified after calculating the gradient of every single observation. This process is repeated and the weights are modified, till it has worked through the entire data set. It then, it repeats the above procedure for the number of iterations as defined.

Batch gradient descent performs computations on the complete data set before each parameter update. SGD performs one update at a time and is therefore much faster. It can also be used to learn with online streaming data. However, SGD performs frequent updates with a high variance that causes the objective function to fluctuate heavily and this complicates convergence to the minimum because it keeps overshooting. However, if we slowly decrease the learning rate, SGD can have the same convergence behavior as batch gradient descent, and converges to a local/global minimum.

5.2.3 Mini-Batch Gradient Descent

We often get faster results if we use a subset of the training examples to perform each update at each step. The mini-batch gradient descent uses an intermediate number (between one, as in SGD, and the complete set of observations, as in batch gradient descent) of examples for each step. Mini-batch gradient descent therefore, takes the best of both the batch gradient descent and SGD and performs an update for every mini-batch size. The advantages of this method are the following:

(a) it reduces the variance of the parameters leading to a more stable convergence,
(b) it can make use of highly optimized matrix optimizations techniques present in many deep learning libraries to allow computation of the gradient with respect to a mini-batch, very efficient.

The commonly used mini-batch sizes are between 50 and 256. This method is typically the choice when training a neural network with a large data set.

With mini-batch gradient descent, we will loop over the mini-batches instead of looping over individual training examples. There are two stages involved in building a mini-batch

- Shuffle—Shuffle the data set (both X and Y values simultaneously), so that the observations are split randomly.
- Partition—Partition the data set into the defined mini-batch size set by the hyperparameter `mini_batch_size`. It should be noted that the last mini-batch size may or may not be the same as the defined `mini_batch_size`, because the number of mini-batches may not.

If the hyperparameter `mini_batch_size` is well tuned, it usually outperforms either the batch gradient descent and/or the SGD, particularly when the training set is large.

The function `random_mini_batches` segregates the observations as per the "mini_batch_size", which has a value 1, when we use stochastic gradient descent and the defined number of observations per mini-batch, for the mini-batch gradient descent method.

The general rule is

- For a relatively small training data set, with $\leq$2000 observations, use batch gradient descent.
- For a larger training data set, use mini-batch sizes 64, 128, 256, 512, . . ., i.e., a power of 2 (due to the computer architecture).

```
random_mini_batches <- function(X, Y, mini_batch_size, seed){

    set.seed(seed)
    # Get number of training samples
    m = dim(X)[2]

    # Initialize mini batches
    mini_batches = list()

    # Create  a list of random numbers
    rand_sample = c(sample(m))

    # Randomly shuffle the training data
    shuffled_X = X[, rand_sample]
    shuffled_Y = Y[rand_sample]

    # Compute number of mini batches
    num_minibatches = floor(m / mini_batch_size)
    batch = 0

    for(i in 0:(num_minibatches - 1)){
        batch = batch + 1
        # Set the lower & upper bound of the mini batches
        lower = (i * mini_batch_size) + 1
        upper = ((i + 1) * mini_batch_size)
        mini_batch_X = shuffled_X[, lower:upper]
        mini_batch_Y = shuffled_Y[lower:upper]

        mini_batch = list("mini_batch_X" = mini_batch_X,
                          "mini_batch_Y" = mini_batch_Y)
        mini_batches[[batch]] = mini_batch
    }
```

```
    # If the batch size does not divide evenly with mini batch size
    if(m %% mini_batch_size != 0){
      # Set the start and end of last batch
      start = floor(m / mini_batch_size) * mini_batch_size
      end = start + m %% mini_batch_size
      mini_batch_X = shuffled_X[, (start + 1):end]
      mini_batch_Y = shuffled_Y[(start + 1):end]
      mini_batch_last = list("mini_batch_X" = mini_batch_X,
                             "mini_batch_Y" = mini_batch_Y)
      mini_batches[[batch + 1]] <- c(mini_batch, mini_batch_last)
    }

  return(mini_batches)
}
```

5.3 Parameter Updates

Optimization is the process of finding the minimum (or maximum) of a function, subject to certain constraints. In neural networks, we want to minimize the cost function by converging to the minima at the earliest.

Once the gradient is computed during backpropagation, the gradients are used to perform a parameter update. In this section, we will discuss several approaches which, we use for performing parameter update. A detailed analysis of parameter updates is however outside the scope of this book.

5.3.1 Simple Update

The simplest way to update the weights is to change them along the negative gradient direction (since the gradient indicates the direction of increase). If we represent the gradient as $\frac{\partial \mathcal{J}}{\partial W}$, the simplest update can be written as

$$W := W - \alpha \frac{\partial \mathcal{J}}{\partial W} \tag{5.3.1}$$

where, α is a fixed hyparameter called the the learning rate, i.e., how small/large a step should be taken to arrive at the global minima.

5.3.2 Momentum Update

This is an approach by which we can improve our convergence on deep networks and has its motivations from physics. Here, the cost can be interpreted as the height of a hilly terrain as height is related to potential energy in physics. Potential energy

is represented as $mass \times gravitational\ acceleration \times height$ and therefore, the cost is related to the potential energy and is proportional to "height".

Let us conceptualize our weight to that of a ball rolling down a hill. By initializing our weights to either zero or any random number is the same as initializing the initial velocity of the ball at some location. The ball (weight) is now ready to roll down the "hill" with an initial velocity V.

In physics, the force acting on a particle is proportional to the gradient of the potential energy. Since the cost is represented as potential energy, we can intuitively state that the force felt by our weights is the negative gradient of the cost function. Also since force, $F = mass \times acceleration$, the negative gradient is proportional to the acceleration of the weight of the ball. Acceleration therefore just changes the ball's position by altering its velocity. A motion which is driven by velocity is better suited to counteract the effects of a wildly fluctuating gradient, by smoothing the trajectory of the ball over its history. Velocity therefore serves as a form of memory, thereby allowing us to cumulatively calculate the movement toward the minimum direction while canceling out oscillating accelerations in the **orthogonal directions** (refer saddle points in Chap. 1). Velocity therefore is the exponentially weighted average of the gradient on the previous steps.

We are now introducing two hyperparameters—**Velocity** (V) and **Momentum** (β). The hyperparameter, β is used to dampen the velocity so as to reduce the kinetic energy of our weights, or else, our weights would never come to a stop at the bottom of a hill. The term, momentum (β), is a misnomer here since it is more related to the coefficient of friction, in physics.

In essence, the mini-batch gradient descent makes a parameter update with a subset of the observations and therefore, the direction of the update has some inherent variance. This results in some "oscillation" in the path taken by mini-batch gradient descent toward convergence. Momentum reduces these oscillations by taking into account the past gradients to smooth out the update. We store the "direction" of the previous gradients in the variable V, which is the exponentially weighted average of the gradient in the previous steps. We therefore update V and use the updated velocity to update our weights

$$\begin{aligned} V_{dW^{[l]}} &= \beta \times V_{dW^{[l]}} + (1-\beta)\frac{\partial \mathcal{J}}{\partial W^{[l]}} \\ W^{[l]} &:= W^{[l]} - \alpha \times V_{dW^{[l]}} \quad \alpha \textit{ is the learning rate} \end{aligned} \tag{5.3.2}$$

In the above equation, we are using the momentum hyperparameter β to determine what fraction of the previous velocity we should retain in the new update, and add this "memory" (which is composed of past gradients) to our current gradient. Since the momentum term increases the step size while using momentum, we may require a reduced learning rate as with respect to mini-batch gradient descent.

We usually assign a value of 0.9 for the hyperparameter β but can also be set to either of 0.5, 0.9, 0.95, 0.99.

```
initialize_velocity <- function(parameters) {
    L = length(parameters)
    v = list()
    for (layer in 1:L) {
        v[[paste("dW", layer, sep = "")]] = 0 * parameters[[paste("W",
            layer, sep = "")]]
        v[[paste("db", layer, sep = "")]] = 0 * parameters[[paste("b",
            layer, sep = "")]]
    }

    return(v)
}
```

```
update_params_with_momentum <- function(parameters, gradients,
    velocity, beta, learning_rate) {

    L = length(parameters)/2

    for (l in 1:L) {
        velocity[[paste("dW", l, sep = "")]] = beta * velocity[[paste("dW",
            l, sep = "")]] + (1 - beta) * gradients[[paste("dW",
            l, sep = "")]]
        velocity[[paste("db", l, sep = "")]] = beta * velocity[[paste("db",
            l, sep = "")]] + (1 - beta) * gradients[[paste("db",
            l, sep = "")]]

        parameters[[paste("W", l, sep = "")]] = parameters[[paste("W",
            l, sep = "")]] - learning_rate * velocity[[paste("dW",
            l, sep = "")]]
        parameters[[paste("b", l, sep = "")]] = parameters[[paste("b",
            l, sep = "")]] - learning_rate * velocity[[paste("db",
            l, sep = "")]]
    }

    return(list(parameters = parameters, Velocity = velocity))
}
```

With momentum update, we update our weights in the direction of the velocity. Over the first few iterations, V increases and our descent takes longer steps. As we approach the global minima, our velocity slows down as the derivative of the cost with respect to the weight gets smaller and reaches a value of zero, when the weights reach the minima. It should be noted that the velocity acts independently for each weight, because each weight is changing independently.

5.3.3 Nesterov Momentum Update

Nesterov's accelerated gradient is a first-order optimization method which has a better convergence rate than gradient descent for general convex functions [16], and it was proved that it worked better than classical momentum updates during training neural networks (Fig. 5.1). This was further confirmed by [17], who provided an alternative formulation.

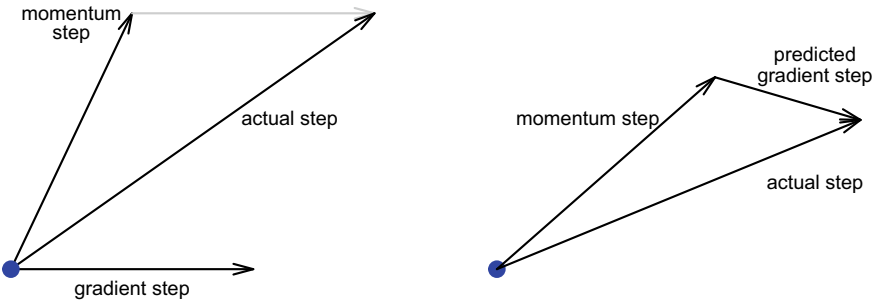

Fig. 5.1 Momentum update and Nesterov update: the left-hand figure shows a typical momentum update which is the sum of the gradient vector and the momentum vector; the right-hand plot is the Nesterov update where the actual step is the sum of the momentum vector and the vector of the predicted gradient

This is a modification of the momentum update and has been gaining popularity of late, due to stronger convergence. The idea behind Nesterov momentum is that weights having a certain value, will be perturbed by an amount $\mu \times V$, irrespective of the gradient (which is fixed at that particular location). It therefore makes sense that the next set of values of the weights would be $W + \mu \times V$. This is a kind of, prediction, of the values of the next set of weights. It therefore makes sense to compute the gradient of $W + \mu \times V$, instead of the gradient of cost with respect to "old" values of our weights.

With the momentum update, we are taking the gradient of the weights at the current position; with the Nesterov momentum, we are refining our update by taking the gradient of the predicted set of weights (Fig. 5.1).

We therefore take the gradient of the predicted weights instead of the earlier weights V and use the updated velocity to update our weights

$$\begin{aligned} V_{prev} &= V \quad \text{\#make a copy} \\ V &= (\mu \times V) - \lambda \frac{\partial (W + \mu V)}{\partial W} \quad \text{velocity update} \\ W &= W - (\mu \times V_{prev}) + (1 + \mu) \times V \quad \text{position of the weight changes} \end{aligned} \tag{5.3.3}$$

5.3.4 Annealing the Learning Rate

With a high learning rate, the neural network system will have a high kinetic energy and the weights vector has the possibility to bounce around the deeper and narrower "crevaces" of the loss function. While training deep neural networks, it is therefore helpful to anneal (decay) the learning rate over time. The challenge is to know when to decay and by how much—if we decay it slowly there are chances that our system

will bounce around chaotically with little improvement; if we decay it very fast, our system will be unable to reach the best position.

We normally use one of the three types of learning rate decay

- **Step decay**—Reduce the learning rate by some factor every few epochs. A typical heuristic could be to observe the validation error, while training with a fixed learning rate, and **reduce** the learning rate by a constant (say 0.5) whenever there is no improvement in the validation error. Another method is to reduce the learning rate by half every 5 epochs, or by 0.1 every 20 epochs. It may be appreciated that choosing the correct decay will depend on the type of the problem, the complexity of the model and the skill of the data scientist.
- **Exponential decay**—as the name signifies, it is represented as $\alpha = \alpha_0 exp^{-kt}$, where α_0 and k are hyperparameters and t is the iteration number or epoch number.
- $\frac{1}{t}$ **decay**—this is represented as $\alpha = \frac{\alpha_0}{(1+kt)}$ where, α_0 and k are the hyperparameters and t is the iteration number

If we can afford the computational cost, a slower decay and longer training time could be a good option.

5.3.5 Second-Order Methods

The gradient descent training algorithm requires many iterations and does not necessarily produce the fastest convergence. Some of the other techniques are: (1) Newton's method, (2) Conjugate gradient (3) Quasi Newton, and (4) Levenberg Marquardt. We will touch upon the Newton's method.

Newton's Method

The Newton's method (also known as the Newton–Raphson method) is a second-order algorithm, making use of the Hessian[1] matrix. This method can help to find better training directions by using the second-order derivatives of the loss function.

The convergence is based on a small number called the tolerance and identified by ϵ, such that

$$\begin{aligned} f(w^{(t+1)}) - f(w^{(t)}) &< \epsilon \\ f(w^{(t)} + \Delta w^{(t)}) - f(w^{(t)}) &< \epsilon \end{aligned} \tag{5.3.4}$$

Using `Taylor's expansion` in Eq. 5.3.4 we get

$$f(w^{(t)} + \Delta w^{(t)}) \approx f(w^{(t)}) + \Delta w^{(t)} \left.\frac{\partial f}{\partial w}\right|_{w^{(t)}} + \frac{\Delta (w^{(t)})^2}{2} \left.\frac{\partial^2 f}{\partial w^2}\right|_{w^{(t)}} \tag{5.3.5}$$

[1] Hessian is a square matrix of second-order partial derivatives.

Equation 5.3.5 is a quadratic approximation where, $\frac{\partial f}{\partial w}\Big|_{w^{(t)}}$ and $\frac{\partial^2 f}{\partial w^2}\Big|_{w^{(t)}}$ are called the **gradient** and **Hessian** of the function f at time step $w^{(t)}$.

Since $w^{(t+1)} = x^t + \Delta w^{(t)}$, we can rewrite Eq. (5.3.5) as a function of $\Delta w^{(t)}$. Representing the gradient and the Hessian by $\mathbf{g_t}$ and $\mathbf{H_t}$, we can write

$$h_t(\Delta w^{(t)}) = f(w^{(t)}) + \Delta w^{(t)}\mathbf{g_t} + \frac{1}{2}\Delta(w^{(t)})^2\mathbf{H_t} \tag{5.3.6}$$

As our intent, as stated in the beginning is to minimize $f(w)$, we can do that by differentiating Eq. 5.3.6 and setting it to zero

$$\frac{\partial h_t(\Delta w^{(t)})}{\partial \Delta w^{(t)}} = \mathbf{g_t} + \Delta w^{(t)}\mathbf{H_t} \tag{5.3.7}$$

Solving for $\Delta w^{(t)}$, we get

$$\Delta w^{(t)} = -\mathbf{g_t}\mathbf{H_t^{-1}} \tag{5.3.8}$$

Since $x^{t+1} = x^t + \Delta x^t$, and applying the learning rate η we can write

$$w^{(t+1)} = w^{(t)} - \eta(\mathbf{g_t}\mathbf{H_t^{-1}}) \tag{5.3.9}$$

Newton's method has its drawbacks—the exact evaluation of the Hessian and its inverse are computationally very expensive.

5.3.6 *Per-Parameter Adaptive Learning Rate Methods*

In the previous approaches, we have discussed so far, we have manipulated the learning rate equally for all parameters. Tuning the learning rate is computationally expensive and therefore, if we can adaptively tune the learning rates, per parameter, it would ease the computational cost. In this section, we will discuss some common adaptive methods which can be used.

Adagrad (Adaptive Gradient Algorithm) Update

Adagrad (Accumulating Historical Gradients) [18] is an algorithm that adapts the learning rate to the parameter by performing larger updates for infrequent parameters and smaller updates for frequent parameters. It is therefore well suited for sparse data.

RMSProp (Root Mean Square Propagation) Update

RMSProp is a *Exponentially Weighted Moving Average of Gradients*. While `AdaGrad` works well for simple convex functions, the flatter regions can force `AdaGrad` to decrease the learning rate before it reaches a minimum. Therefore, simply using a naive accumulation of gradients is not a good solution. Reference

[7], suggested `RMSProp` as an adaptive learning rate method wherein, instead of an accumulation of historical gradients, an Exponentially Weighted Moving Averages (EWMA) of gradients, can enable us to discard measurements that we made at an earlier time step. In that context, `RMSprop` divides the learning rate by an exponentially decaying average of squared gradients.

More specifically, our update to the gradient accumulation vector is now as follows:

$$\begin{aligned} &\text{For } t \text{ in iteration compute dW, db using the current mini-batch:} \\ s_{dW} &= \beta s_{dW} + (1-\beta)[dW]^2 \\ s_{db} &= \beta s_{db} + (1-\beta)[db]^2 \\ W &:= W - \alpha \frac{dW}{\sqrt{s_{dW} + \epsilon}} \end{aligned} \tag{5.3.10}$$

The value of ϵ is set at $10e$-08, so that we do not get a divide by zero situation. Reference [7] suggests a value of 0.9 for β, and a value of 0.001 for the learning rate α.

Adam (Adaptive Moment Estimation) Update

Reference [19], suggested `Adam` in which, adaptive learning rates are computed for each parameter. `Adam` stores (a) an exponentially decaying average of past squared gradients and (b) an exponentially decaying average of past gradients. If momentum can be envisaged as a ball running down a slope, `Adam` is akin to a heavy ball with friction, which prefers a flat minima in the error surface. Reference [19], describe `Adam` as combining the advantages of both `AdaGrad` and `RMSProp`.

Some of the popular deep learning libraries recommend the following hyperparameters values for `Adam`:

- TensorFlow: learning_rate $= 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 1e$-08
- Keras: learning_rate $= 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 1e$-08, decay $= 0$.

We compute the decaying averages of past and past squared gradients as follows:

$$\begin{aligned} &\text{initialize } v_{dW}, s_{dW}, v_{db}, s_{db} = 0 \\ &\text{For } t \text{ in iteration compute dW, db using the current mini-batch:} \\ v_{dW} &= \beta_1 v_{dW} + (1-\beta_1) dW \qquad -\beta_1 \text{ is the momentum} \\ s_{dW} &= \beta_2 s_{dW} + (1-\beta_2)[dW]^2 \qquad -\beta_2 \text{ is the RMSProp } \beta \\ v_{dW}^{corrected} &= \frac{v_{dW}}{(1-\beta_1^t)} \\ s_{dW}^{corrected} &= \frac{s_{dW}}{(1-\beta_2^t)} \\ W &:= W - \alpha \frac{v_{dW}^{corrected}}{\sqrt{s_{dW}^{corrected}} + \epsilon} \end{aligned} \tag{5.3.11}$$

Each update of `Adam` involves the following steps

(1) Compute the gradient and its element-wise square using the current parameters.
(2) Update the exponential moving average of the first-order moment and the second-order moment.
(3) Compute an unbiased average of the first-order moment and second-order moment.
(4) Compute weight update: first-order moment unbiased average divided by the square root of second-order moment unbiased average (and scale by learning rate).
(5) Apply update to the weights.

```
initialize_adam <- function(parameters) {

    L = length(parameters)/2
    v = list()
    s = list()

    for (layer in 1:L) {
        v[[paste("dW", layer, sep = "")]] = 0 * parameters[[paste("W",
            layer, sep = "")]]
        v[[paste("db", layer, sep = "")]] = 0 * parameters[[paste("b",
            layer, sep = "")]]

        s[[paste("dW", layer, sep = "")]] = 0 * parameters[[paste("W",
            layer, sep = "")]]
        s[[paste("db", layer, sep = "")]] = 0 * parameters[[paste("b",
            layer, sep = "")]]
    }

    return(list(V = v, S = s))
}
```

```
update_params_with_adam <- function(parameters, gradients, v,
    s, t, beta1, beta2, learning_rate, epsilon) {

    L = length(parameters)/2
    v_corrected = list()
    s_corrected = list()

    for (layer in 1:L) {
        v[[paste("dW", layer, sep = "")]] = beta1 * v[[paste("dW",
            layer, sep = "")]] + (1 - beta1) * (gradients[[paste("dW",
            layer, sep = "")]])
        v[[paste("db", layer, sep = "")]] = beta1 * v[[paste("db",
            layer, sep = "")]] + (1 - beta1) * gradients[[paste("db",
            layer, sep = "")]]

        v_corrected[[paste("dW", layer, sep = "")]] = v[[paste("dW",
            layer, sep = "")]]/(1 - beta1^t)
        v_corrected[[paste("db", layer, sep = "")]] = v[[paste("db",
            layer, sep = "")]]/(1 - beta1^t)

        s[[paste("dW", layer, sep = "")]] = beta2 * s[[paste("dW",
            layer, sep = "")]] + (1 - beta2) * (gradients[[paste("dW",
            layer, sep = "")]])^2
        s[[paste("db", layer, sep = "")]] = beta2 * s[[paste("db",
```

```
            layer, sep = "")]] + (1 - beta2) * (gradients[[paste("db",
            layer, sep = "")]])^2

        s_corrected[[paste("dW", layer, sep = "")]] = s[[paste("dW",
            layer, sep = "")]]/(1 - beta2^t)
        s_corrected[[paste("db", layer, sep = "")]] = s[[paste("db",
            layer, sep = "")]]/(1 - beta2^t)

        parameters[[paste("W", layer, sep = "")]] = parameters[[paste("W",
            layer, sep = "")]] - learning_rate * (v_corrected[[paste("dW",
            layer, sep = "")]])/(sqrt(s_corrected[[paste("dW",
            layer, sep = "")]]) + epsilon)
        parameters[[paste("b", layer, sep = "")]] = parameters[[paste("b",
            layer, sep = "")]] - learning_rate * (v_corrected[[paste("db",
            layer, sep = "")]])/(sqrt(s_corrected[[paste("db",
            layer, sep = "")]]) + epsilon)
    }

    return(list(parameters = parameters, Velocity = v, S = s))
}
```

It should be noted that the mini-batch/SGD implementation requires three for-loops in total

- Number of iterations.
- Number of training observations.
- Number of layers (update all parameters from $W^{[1]}, b^{[1]}$ to $W^{[L]}, b^{[L]}$).

```
model <- function(X,
                  Y,
                  X_test,
                  Y_test,
                  layers_dims,
                  hidden_layer_act,
                  output_layer_act,
                  optimizer,
                  learning_rate,
                  mini_batch_size,
                  num_epochs,
                  initialization,
                  beta,
                  beta1,
                  beta2,
                  epsilon,
                  print_cost = F){
  costs <- NULL
  t = 0
  set.seed = 1
  seed = 10
  parameters = initialize_params(layers_dims, initialization)
  v = initialize_adam(parameters)[["V"]]
  s = initialize_adam(parameters)[["S"]]
  velocity = initialize_velocity(parameters)

  start_time <- Sys.time()

  for(i in 0:num_epochs){
        seed = seed + 1
```

```
        minibatches = random_mini_batches(X, Y, mini_batch_size, seed)

        for(batch in 1:length(minibatches)){
            mini_batch_X = (minibatches[[batch]][['mini_batch_X']])
            mini_batch_Y = minibatches[[batch]][['mini_batch_Y']]

            AL = forward_prop(mini_batch_X, parameters, hidden_layer_act,
                              output_layer_act)[['AL']]

            caches = forward_prop(mini_batch_X, parameters, hidden_layer_act,
                                  output_layer_act)[['caches']]

            cost <- compute_cost(AL, mini_batch_X, mini_batch_Y, num_classes = 0,
                            output_layer_act)

            gradients = back_prop(AL, mini_batch_Y, caches, hidden_layer_act,
                                  output_layer_act)

            if(optimizer == 'gd'){
              parameters = update_params(parameters, gradients, learning_rate)
            }
            else if(optimizer == 'momentum'){
              parameters = update_params_with_momentum(parameters,
                                                       gradients,
                                                       velocity,
                                                       beta,
                                                       learning_rate
                                                       )[["parameters"]]
              velocity = update_params_with_momentum(parameters,
                                                     gradients,
                                                     velocity,
                                                     beta,
                                                     learning_rate
                                                     )[["Velocity"]]
            }
            else if(optimizer == 'adam'){
            t = t + 1
            parameters = update_params_with_adam(parameters,
                                                 gradients,
                                                 v,
                                                 s,
                                                 t,
                                                 beta1,
                                                 beta2,
                                                 learning_rate,
                                                 epsilon
                                                 )[["parameters"]]
            v = update_params_with_adam(parameters,
                                        gradients,
                                        v,
                                        s,
                                        t,
                                        beta1,
                                        beta2,
                                        learning_rate,
                                        epsilon
                                        )[["Velocity"]]
            s = update_params_with_adam(parameters,
                                        gradients,
                                        v,
```

```
                                        s,
                                        t,
                                        beta1,
                                        beta2,
                                        learning_rate,
                                        epsilon
                                        )[["S"]]
            }
        }

        if(print_cost == T & i %% 1000 == 0){
            print(paste0("Cost after iteration " , i, ' = ', cost, sep = ' '))
            }
        if(print_cost == T & i %% 100 == 0){
            costs = c(costs, cost)
            }
    }

    Y_prediction_train = predict_model(parameters, X, hidden_layer_act,
                                       output_layer_act)
    Y_prediction_test = predict_model(parameters, X_test, hidden_layer_act,
                                      output_layer_act)

    cat(sprintf("train accuracy: %05f, \n",
                (100 - mean(abs(Y_prediction_train - Y)) * 100)))
    cat(sprintf("test accuracy: %05f, \n",
                (100 - mean(abs(Y_prediction_test - Y_test)) * 100)))
    cat(sprintf("Cost after: %d, iterations is: %05f, \n", i, cost))

    end_time <- Sys.time()
          cat(sprintf("Application running time: %#.3f minutes",
                      end_time - start_time ))

return(list("parameters" = parameters, "costs" = costs))
}
```

Let us normalize the data

```
scale.trainX <- t(scale(trainX))
scale.testX <- t(scale(testX))
```

Let us now create four different models using the optimization methods: (1) batch gradient descent, (2) mini-batch gradient descent, (3) mini-batch gradient descent with momentum, and (4) mini-batch gradient descent with adam, respectively.

For batch gradient descent optimization, we will be using all the observations and therefore, the argument `mini_batch_size` will be set to 600, i.e., the number of observations in the training data set (Fig. 5.2).

```
layers_dims <- c(2, 100, 1)

model_batch_gd <- model(scale.trainX,
                        trainY,
                        scale.testX,
                        testY,
                        layers_dims,
                        hidden_layer_act = 'relu',
                        output_layer_act = 'sigmoid',
                        optimizer = 'gd',
```

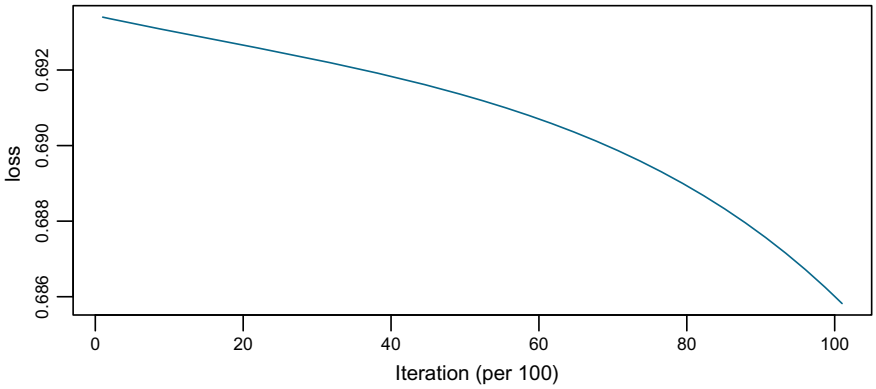

Fig. 5.2 Batch Gradient descent optimizer: loss versus iteration

```
                        learning_rate = 0.001,
                        mini_batch_size = 600,
                        num_epochs = 10000,
                        initialization = 'random',
                        beta = 0.9,
                        beta1 = 0.9,
                        beta2 = 0.999,
                        epsilon = 1e-8,
                        print_cost = T)
```

```
[1] "Cost after iteration 0 = 0.693399526383027 "
[1] "Cost after iteration 1000 = 0.692996097339685 "
[1] "Cost after iteration 2000 = 0.692616936479433 "
[1] "Cost after iteration 3000 = 0.692225661133013 "
[1] "Cost after iteration 4000 = 0.691787142978714 "
[1] "Cost after iteration 5000 = 0.691269259982158 "
[1] "Cost after iteration 6000 = 0.690634881182416 "
[1] "Cost after iteration 7000 = 0.689838379901064 "
[1] "Cost after iteration 8000 = 0.688821489352451 "
[1] "Cost after iteration 9000 = 0.687510626499713 "
[1] "Cost after iteration 10000 = 0.685819368589529 "
train accuracy: 50.833333,
test accuracy: 47.500000,
Cost after: 10000, iterations is: 0.685819,
Application running time: 1.580 minutes
```

For mini-batch gradient descent, we will choose a mini-batch size of 64 (Figs. 5.3, 5.4, 5.5).

```
layers_dims <- c(2, 100, 1)

model_minibatch_gd <- model(scale.trainX,
                            trainY,
```

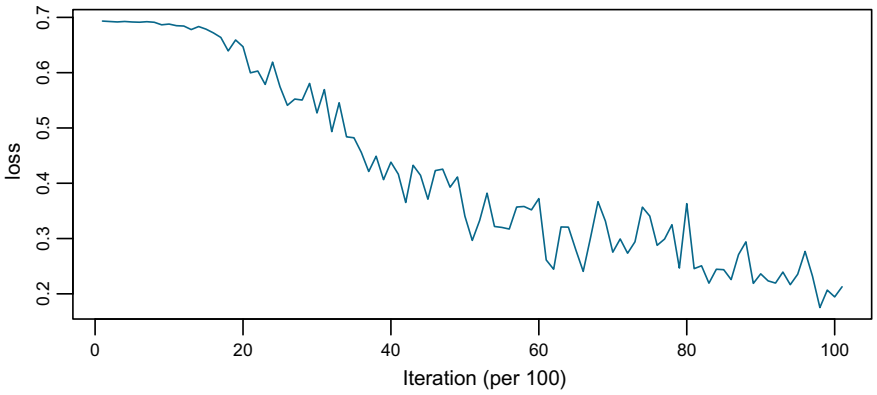

Fig. 5.3 Mini-batch gradient descent optimizer: loss versus iteration. The gradient descent optimizer oscillates wildly and does not quite do well in reducing the objective cost function

```
                        scale.testX,
                        testY,
                        layers_dims,
                        hidden_layer_act = 'relu',
                        output_layer_act = 'sigmoid',
                        optimizer = 'gd',
                        learning_rate = 0.001,
                        mini_batch_size = 64,
                        num_epochs = 10000,
                        initialization = 'random',
                        beta = 0.9,
                        beta1 = 0.9,
                        beta2 = 0.999,
                        epsilon = 1e-8,
                        print_cost = T)
```

```
[1] "Cost after iteration 0 = 0.693321360452555 "
[1] "Cost after iteration 1000 = 0.684982371935506 "
[1] "Cost after iteration 2000 = 0.599731869142396 "
[1] "Cost after iteration 3000 = 0.569371253111551 "
[1] "Cost after iteration 4000 = 0.41637987226805 "
[1] "Cost after iteration 5000 = 0.296604362374641 "
[1] "Cost after iteration 6000 = 0.261217834087639 "
[1] "Cost after iteration 7000 = 0.299227003420015 "
[1] "Cost after iteration 8000 = 0.245556139754028 "
[1] "Cost after iteration 9000 = 0.223421205824919 "
[1] "Cost after iteration 10000 = 0.212745419311865 "
train accuracy: 92.666667,
test accuracy: 93.000000,
Cost after: 10000, iterations is: 0.212745,
Application running time: 3.236 minutes
```

```
layers_dims <- c(2, 100, 1)

model_minibatch_gd_mom <- model(scale.trainX,
                                trainY,
                                scale.testX,
                                testY,
                                layers_dims,
                                hidden_layer_act = 'relu',
                                output_layer_act = 'sigmoid',
                                optimizer = 'momentum',
                                learning_rate = 0.001,
                                mini_batch_size = 64,
                                num_epochs = 10000,
                                initialization = 'random',
                                beta = 0.9,
                                beta1 = 0.9,
                                beta2 = 0.999,
                                epsilon = 1e-8,
                                print_cost = T)
```

```
[1] "Cost after iteration 0 = 0.693320562283126 "
[1] "Cost after iteration 1000 = 0.685022531415745 "
[1] "Cost after iteration 2000 = 0.600111907605888 "
[1] "Cost after iteration 3000 = 0.569681803345362 "
[1] "Cost after iteration 4000 = 0.416619273843368 "
[1] "Cost after iteration 5000 = 0.296770206347163 "
[1] "Cost after iteration 6000 = 0.261311267794582 "
[1] "Cost after iteration 7000 = 0.299345591302726 "
[1] "Cost after iteration 8000 = 0.245693797051899 "
[1] "Cost after iteration 9000 = 0.223527223844926 "
[1] "Cost after iteration 10000 = 0.21282563453313 "
train accuracy: 92.666667,
test accuracy: 93.000000,
Cost after: 10000, iterations is: 0.212826,
Application running time: 2.469 minutes
```

```
layers_dims <- c(2, 100, 1)

model_minibatch_adam <- model(scale.trainX,
                              trainY,
                              scale.testX,
                              testY,
                              layers_dims,
                              hidden_layer_act = 'relu',
                              output_layer_act = 'sigmoid',
                              optimizer = 'adam',
                              learning_rate = 0.001,
                              mini_batch_size = 64,
                              num_epochs = 10000,
                              initialization = 'random',
                              beta = 0.9,
                              beta1 = 0.9,
                              beta2 = 0.999,
```

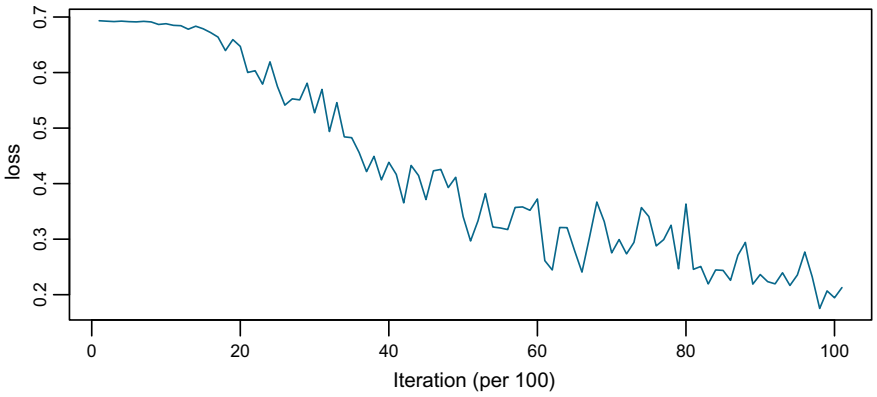

Fig. 5.4 Mini-batch gradient descent with momentum: loss versus iteration

```
                                epsilon = 1e-8,
                                print_cost = T)
```

```
[1] "Cost after iteration 0 = 0.692243953244077 "
[1] "Cost after iteration 1000 = 0.082159559525978 "
[1] "Cost after iteration 2000 = 0.0205592965269963 "
[1] "Cost after iteration 3000 = 0.100252805396719 "
[1] "Cost after iteration 4000 = 0.0208339453640888 "
[1] "Cost after iteration 5000 = 0.0386408118308484 "
[1] "Cost after iteration 6000 = 0.0369781948814698 "
[1] "Cost after iteration 7000 = 0.0354628106862388 "
[1] "Cost after iteration 8000 = 0.0713437232696845 "
[1] "Cost after iteration 9000 = 0.0467632093035248 "
[1] "Cost after iteration 10000 = 0.0878683736127119 "
train accuracy: 97.500000,
test accuracy: 95.500000,
Cost after: 10000, iterations is: 0.087868,
Application running time: 3.373 minutes
```

```
# Plotting decision boundary on the training data
s.trainX.df <- data.frame(scale(trainX))
s.testX.df <- data.frame(scale(testX))

data <- cbind(s.trainX.df, trainY)
class <- data[, 3]
data <- data[, 1:2]
plot(data, col = class + 1, pch = class + 1, cex = 0.5, xaxt = "n",
    yaxt = "n", xlab = "x1", ylab = "x2", cex.lab = 0.7)
axis(side = 1, col = "black", cex.axis = 0.7)
axis(side = 2, col = "black", cex.axis = 0.7)

r <- sapply(data, range, na.rm = TRUE)
```

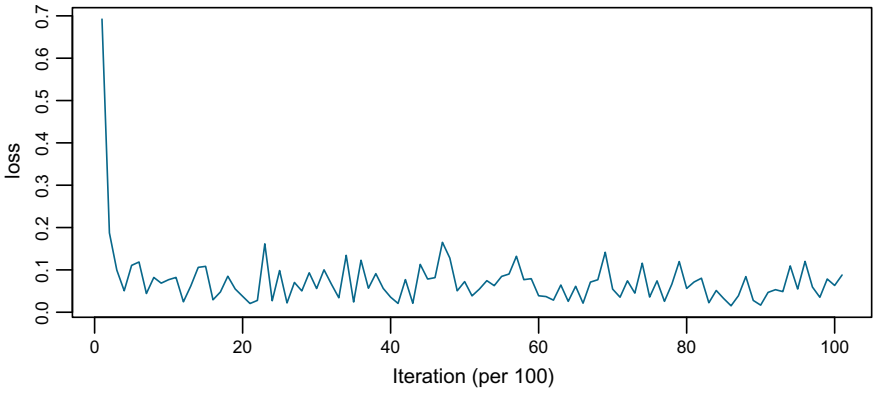

Fig. 5.5 Mini-batch gradient descent with Adam optimization: loss versus iteration. Mini-batch gradient descent with Adam optimization clearly outperforms the others by converging to a lower value of the loss, much earlier

```
xs <- seq(r[1, 1], r[2, 1], length.out = 100)
ys <- seq(r[1, 2], r[2, 2], length.out = 100)
grid <- cbind(rep(xs, each = 100), rep(ys, times = 100))
colnames(grid) <- colnames(r)
grid <- as.data.frame(grid)

predicted <- predict_model(model_minibatch_adam$parameters, t(grid),
    hidden_layer_act = "relu", output_layer_act = "sigmoid")
points(grid, col = predicted + 1, pch = ".")

z <- matrix(predicted, nrow = 100, byrow = TRUE)
z <- matrix(predicted, nrow = 100, byrow = TRUE)
contour(xs, ys, z, add = T, levels = 1:(k - 1) + 0.5)
```

5.4 Vanishing Gradient

We can run the risk of a deep layer neural network to train indefinitely or perform inaccurately. The reason is that iterative optimization algorithms slowly make their way to the local optima by perturbing weights in a direction inferred from the gradient, so that the cost (difference between the predicted and actual values) decreases. The gradient descent algorithm updates the weights by the negative of the gradient multiplied by a hyperparameter known as the learning rate (values between 0 and 1). Another hyperparameter is the number of iterations; if the number of iterations is small we may have inaccurate results, and if the number is large, the time to train can be very long. Therefore it is a trade-off between training time and accuracy. Moreover, if the gradient at each step is small a larger number of repetitions will be needed till convergence as the weight is not changing enough at each iteration. It is also difficult to represent very small gradients as numerical values in machines,

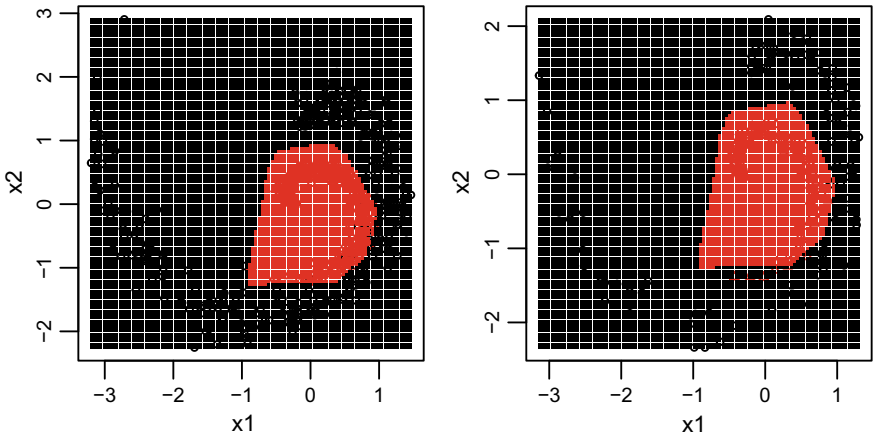

Fig. 5.6 Plot of the decision boundary using the Adam optimizer model on the training data (left-hand plot) and the testing data (right-hand plot)

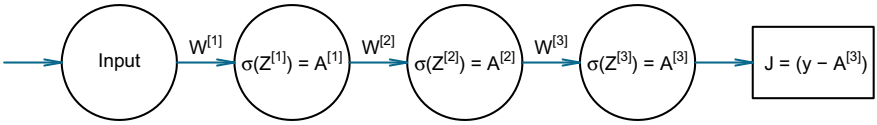

Fig. 5.7 A three-layer neural network with sigmoid activation

due to a problem known as **underflow**. With very small gradients, this becomes a problem and often gives rise to the **vanishing gradient problem**. The activation functions, `sigmoid`, and `tanh` suffer from the vanishing gradient problem. We can also have **exploding gradients** if the gradients are larger than 1.

Let us try to understand the vanishing gradient problem by considering the sigmoid function, which squeezes any input value into an output between the range of (0, 1), where it asymptotes. This is perfect for representations of probabilities and classification. We will consider a three-layer neural network activated by the sigmoid activation function in Fig. 5.7

In Fig. 5.8, the maximum value for the derivative of the `sigmoid` activation function is 0.25. The derivative also asymptotes to zero at the tails and therefore the minimum value is zero. We can say that the the output of the derivative of the `sigmoid` function is between (0, 0.25].

The error J in Fig. 5.7 aggregates the total error of the network. We now need to perform backpropagation to modify our weights through gradient descent to minimize J. The derivative to the first weight can be written as

$$Z^{[1]} = W^{[1]}X + b^{[1]}$$

$$A^{[1]} = \sigma(Z^{[1]})$$

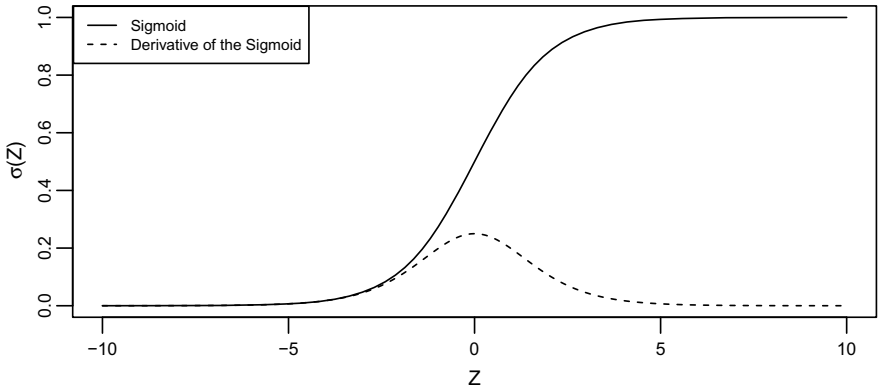

Fig. 5.8 Plot of the derivative of the Sigmoid activation function

$$\frac{\partial A^{[1]}}{\partial Z^{[1]}} = [A^{[1]}(1 - A^{[1]})] = \sigma'(Z^{[1]})$$

$$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$$

$$\frac{\partial Z^{[2]}}{\partial A^{[1]}} = W^{[2]}$$

$$A^{[2]} = \sigma(Z^{[2]})$$

$$\frac{\partial A^{[2]}}{\partial Z^{[2]}} = [A^{[2]}(1 - A^{[2]})] = \sigma'(Z^{[2]})$$

$$\frac{\partial J}{\partial W^{[1]}} = \frac{\partial J}{\partial A^{[2]}} \frac{\partial A^{[2]}}{\partial Z^{[2]}} \frac{\partial Z^{[2]}}{\partial A^{[1]}} \frac{\partial A^{[1]}}{\partial Z1} \frac{\partial Z^{[1]}}{\partial W^{[1]}}$$

$$= [(A^{[2]} - Y)] * \underbrace{[\sigma'(Z^{[2]})]}_{\leq 0.25} * \underbrace{[W^{[2]}]}_{<1} * \underbrace{[\sigma'(Z^{[1]})]}_{\leq 0.25} * [X]$$

Here

- $\sigma'(Z^{[2]}) \leq 0.25$
- $W^{[2]} < 1$
- $\sigma'(Z^{[1]}) \leq 0.25$

And the multiplication of the above three numbers will be much smaller than 0.25 In Fig. 5.8, the derivative of the sigmoid function values is between 0 and 0.25. By multiplying these two derivatives together, we are multiplying two values which are between 0 and 0.25, thereby resulting in an even smaller number. Also, we generally

initialize our weights from a standard Gaussian distribution with mean 0 and standard deviation 1 and thus having values between 1 and -1.

The gradient at a layer is the products of the gradients at prior layers.

In the above equation, we are now multiplying four values which are between 0 and 1, which will tend to become a very small number (even if we employ some other method for initialization of weights) and will result in the vanishing gradient problem. In deep neural networks, the first layer is the furthest from the error, and therefore the derivative will have more sigmoid derivatives, and thus ending up in very small values. The first layer are therefore the slowest to train. Since the other hidden layers and the output layer are each functionally dependent on the previous layers, inaccuracies in the previous layers will have an impact on the downstream layers by building on the inaccuracy and therefore corrupting the entire neural net. (This is more so in the case of ConvNets as the early layers perform high-level feature detection which the downstream layers analyze further).

One of the reasons for the unpopularity of neural networks was the vanishing gradient problem as only the `sigmoid` and `tanh` activation functions were being used. However, with the usage of other activation functions like the `relu`, `lrelu`, etc., this problem has been largely minimized.

So how do these activation functions solve the problem of the vanishing gradient?

As we have seen in Sect. 2.4.3, the `relu` outputs a zero for all input values less than zero and mimics the input when the input is greater than zero. The derivative of the `relu` is zero when the input is less than zero, and when the input is greater or equal to zero, the derivative is equal to one. Our derivatives will now no longer vanish because it is not bounded.

However, the `relu` function can "die" with an output of zero for a negative value input. This can sometimes cause problems in backpropagation because the gradients will be zero for one negative value input to the `relu` function. The "dead" `relu` will now output the same value, i.e., zero, and therefore cannot modify the weights in any way, since not only is the output for any negative input zero, the derivative is too.

The event that the weighted sums ending up negative, does not happen all the time, and we can indeed initialize our weights to be only positive and normalize our input values so that they are between 0 and 1. Another way around is to use the `lrelu`. `lrelu` has a very small gradient instead of a zero gradient when the input is negative, thereby giving a chance for the network to continue learning, albeit slowly with negative inputs.

The vanishing gradient problem cannot be solved with any one technique and we can only delay its occurrence.

`R` uses `IEEE 754` double-precision floating-point numbers and the upper bound of these floating-point numbers is 10^{308}.

5.5 Regularization

Regularization is a method by which we impose constraints on our neural network with an objective to prevent overfitting. One of the ways overfitting happens is when the magnitude of the weights becomes large, which makes the network output function to oscillate wildly. It then captures the underlying noise present in the data instead of the signal present in the data.

One of the ways to prevent the weights from getting inflated is by modifying our objective function and adding an additional term, which penalizes large weights. If we denote our neural network as f, and if the loss function, we are optimizing is (say), the MSE, we can represent the cost as

$$J = \frac{1}{n} \sum_i (y_i - f(x_i))^2$$

Large weights can be penalized by adding the the ℓ_2-regularization term $R(f)$ to the above equation

$$R(f) = \frac{1}{2} \lambda \sum w^2$$

The hyperparameter λ controls the overall magnitude of the regularization term. The $\frac{1}{2}$ multiplier is added only for convenience, for cancelation when taking it's derivative. Adding this term to the cost equation, we now have:

$$J = \frac{1}{n} \sum_i (y_i - f(x_i))^2 + R(f)$$

The net result of adding the regularization term is is to help gradient descent find such parameters which do not result in large weights and thereby preventing the model output to exhibit wild swings.

There are also other regularization terms including ℓ_1-distance or the "Manhattan distance", and each of them has different properties, but approximately the same effect.

Overfitting is not only associated with very large estimated (structural) parameters, but also when the number of features is very high compared to the number of examples. Deep learning models generally tend to have a large capacity due to the number of layers and nodes in the network and also due to the fact that the number of features can be very large. We have also seen in Sect. 1.5 that large capacity algorithms have a serious problem of overfitting, which implies that the trained network does not generalize well to unseen data.

The objective function of a model is to minimize the cost, which is dependent on

- How well the model fits the data.
- Model capacity (measure of the sum of the parameters).

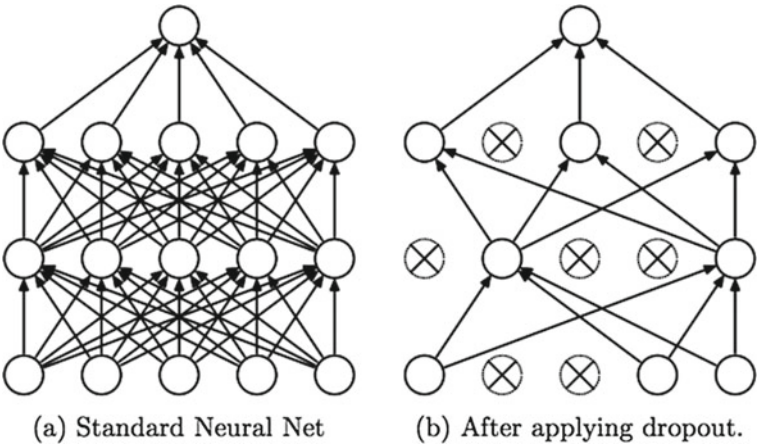

Fig. 5.9 **Left**: A two layer network. **Right**: Same network with dropout. Crossed units have been dropped [20]

The total cost of an algorithm is therefore the sum of the *Measure of fit* and *capacity of the model*.

The measure of fit to training data in a classification algorithm is the *error rate* and the capacity of the algorithm is measured by the ℓ_1 / ℓ_2 norms represented by $\|w\|_1$ and $\|w\|_2^2$, respectively.

5.5.1 Dropout Regularization

Dropout is a regularization technique introduced by [20] in 2014. When dropout is applied to a layer, some percentage of its neurons (commonly having values between 20 and 50%) are randomly deactivated or "dropped out", and their connections are deactivated. This reduces the network's tendency of overdependence on some neurons. This in effect, forces the network to learn a more balanced representation, and therefore prevents overfitting. Dropout regularization is depicted below, from its original publication (Fig. 5.9).

The term "dropout" refers to canceling or removing the units (hidden and visible) in a neural network. By dropping a unit, we temporarily remove it from the network along with its incoming and outgoing connections.

> Use dropout only during training. Dropout is not used during test time. Dropout is applied both during forward and backward propagation.

During training time, we divide each dropout layer by probability value defined as `keep_prob` to keep the same expected value for the activations. For example, if `keep_prob` is 0.5, then we will on average shut down half the nodes, so the output will be scaled by 0.5 since only the remaining half are contributing to the solution.

Dividing by 0.5 is equivalent to multiplying by 2. Hence, the output now has the same expected value. You can check that this works even when `keep_prob` is other values than 0.5.

The idea behind dropout is that at each iteration, we train a different model that uses only a subset of the total number of neurons in the model. With dropout, our neurons thus become less sensitive to activation of a specific neuron because another neuron may be dropped during another iteration.

5.5.2 ℓ_2 Regularization

As we have in Sect. 5.5, an appropriate way to avoid overfitting is the ℓ_2 regularization method. This is done by modifying the cost function as follows:

From:

$$J = -\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)}\log\left(a^{[L](i)}\right) + (1-y^{(i)})\log\left(1-a^{[L](i)}\right)\right) \tag{1}$$

To:

$$J_{regularized} = \underbrace{-\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)}\log\left(a^{[L](i)}\right) + (1-y^{(i)})\log\left(1-a^{[L](i)}\right)\right)}_{\text{cross-entropy cost}} + \underbrace{\frac{1}{m}\frac{\lambda}{2}\sum_{l}\sum_{k}\sum_{j}W_{k,j}^{[l]2}}_{\text{L2 regularization cost}} \tag{2}$$

ℓ_2 regularization relies on the assumption that a model with small weights is simpler than a model with large weights. By penalizing the squared values of the sum of the weights (weight decay) in the cost function, it drives all the weights to smaller values by making it costly to have large weights.

Let us modify some of our earlier functions to accommodate both the dropout and ℓ_2 regularization methods into our model.

To incorporate a random dropout of some of the activation nodes (for dropout regularization), we will have to amend our earlier `forward_prop` function to include regularization, as follows:

```
forward_prop_Reg <- function(X,
                             parameters,
                             hidden_layer_act,
                             output_layer_act,
                             keep_prob){

  dropout_matrix <- list()
  caches <- list()
  A <- X
  L <- length(parameters) / 2

  for(l in 1:(L - 1)){
    A_prev <- A
```

```
    W <- parameters[[paste("W", l, sep = "")]]
    b <- parameters[[paste("b", l, sep = "")]]
    actForward <- f_prop_helper(A_prev, W, b, hidden_layer_act[[l]])
    A <- actForward[['A']]
    caches[[l]] <-actForward

    # Randomly drop some activation units
    # Create a matrix with the same shape as A
    set.seed(1)
    i = dim(A)[1]
    j = dim(A)[2]
    k <- rnorm(i * j)
    # Convert values in k to between 0 and 1
    k = (k - min(k)) / (max(k) - min(k))
    # Create a matrix of D
    D <- matrix(k, nrow = i, ncol = j)
    # Find D which is less than equal to keep_prob
    D <- D < keep_prob
    # Shut down those neurons which are less than keep_prob
    A <- A * D
    # Scale the value of neurons that have not been
    # shut down to keep the expected values
    A <- A / keep_prob
    dropout_matrix[[paste("D", l, sep = "")]] <- D
  }

  W <- parameters[[paste("W", L, sep = "")]]
  b <- parameters[[paste("b", L, sep = "")]]

  actForward = f_prop_helper(A, W, b, output_layer_act)
  AL <- actForward[['A']]
  caches[[L]] <- actForward

return(list("AL" = AL, "caches" = caches, "dropout_matrix" = dropout_matrix))
}
```

The cost function will now need to include the ℓ_2 regularization cost (penalty term for the weights), in addition to the cross-entropy cost.

```
compute_cost_with_Reg <- function(AL, X, Y, num_classes, parameters,
    lambd, output_layer_act) {

    # Cross-Entropy cost
    if (output_layer_act == "sigmoid") {
        m = length(Y)
        cross_entropy_cost = -(1/m) * sum(Y * log(AL) + (1 -
            Y) * log(1 - AL))
    } else if (output_layer_act == "softmax") {
        m = ncol(X)
        y.mat <- matrix(Y, ncol = 1)
        y <- matrix(0, nrow = m, ncol = num_classes)
        for (i in 0:(num_classes - 1)) {
            y[y.mat[, 1] == i, i + 1] <- 1
        }
        correct_logprobs <- -log(AL)
        cross_entropy_cost <- sum(correct_logprobs * y)/m
    }

    # Regularization cost
    L <- length(parameters)/2
```

```
    L2_Reg_Cost = 0

    for (l in 1:L) {
        L2_Reg_Cost = L2_Reg_Cost + sum(parameters[[paste("W",
            l, sep = "")]]^2)
    }
    L2_Reg_Cost = lambd/(2 * m) * L2_Reg_Cost
    cost = cross_entropy_cost + L2_Reg_Cost

    return(cost)
}
```

Since we have changed the cost, we also have to change the backward propagation, and the gradients have to be computed with respect to the new cost.

```
back_prop_Reg_helper  <- function(dA,
                                  cache,
                                  X,
                                  Y,
                                  num_classes,
                                  hidden_layer_act,
                                  lambd){

  forward_cache <-cache[['forward_cache']]
  activation_cache <- cache[['activation_cache']]
  A_prev <- forward_cache[['A_prev']]
  m = dim(A_prev)[2]
  activation_cache <- cache[['activation_cache']]

  if(hidden_layer_act == "relu"){
    dZ <- derivative_relu(dA, activation_cache)
  }
  else if(hidden_layer_act == "sigmoid"){
    dZ <- derivative_sigmoid(dA, activation_cache)
  }
  else if(hidden_layer_act == "tanh"){
    dZ <- derivative_tanh(dA, activation_cache)
  }
  else if(hidden_layer_act == "softmax"){
    dZ <- derivative_softmax(dAL, activation_cache, X, Y, num_classes)
  }

  W <- forward_cache[['W']]
  b <- forward_cache[['b']]
  m = dim(A_prev)[2]

  if(hidden_layer_act == 'softmax'){
    dW = 1 / m * t(dZ) %*% t(A_prev) + (lambd / m) * W
    db = 1 / m * colSums(dZ)
    dA_prev = t(W) %*% t(dZ)
  }
  else{
  dW = 1 / m * dZ %*% t(A_prev) + (lambd / m) * W
  db = 1 / m * rowSums(dZ)
  dA_prev = t(W) %*% dZ
  }

return(list("dA_prev" = dA_prev, "dW" = dW, "db" = db))
}
```

```
back_prop_Reg <- function(AL,
                          X,
                          Y,
                          num_classes,
                          caches,
                          hidden_layer_act,
                          output_layer_act,
                          keep_prob,
                          dropout_matrix,
                          lambd){

  gradients = list()
  L = length(caches)
  m = dim(AL)[2]

  if(output_layer_act == "sigmoid"){
    dAL = -((Y/AL) - (1 - Y)/(1 - AL))
  }
  else if(output_layer_act == 'softmax') {
    y.mat <- matrix(Y, ncol = 1)
    y <- matrix(0, nrow=ncol((X)), ncol = num_classes)
    for (i in 0:(num_classes - 1)) {
      y[y.mat[, 1] == i,i+1] <- 1
    }
    dAL = (AL - y)
  }

  current_cache = caches[[L]]$cache
  if(lambd == 0){
    loop_back_reg_vals <-  back_prop_Reg_helper(dAL,
                                                current_cache,
                                                X, Y,
                                                num_classes,
                                                hidden_layer_act =
                                                  output_layer_act,
                                                lambd)
  }
  else {
    loop_back_reg_vals = back_prop_Reg_helper(dAL, current_cache,
                                              X, Y,
                                              num_classes,
                                              hidden_layer_act =
                                                output_layer_act,
                                              lambd)
  }

  if(output_layer_act == "sigmoid"){
    gradients[[paste("dA", L, sep = "")]] <- loop_back_reg_vals[['dA_prev']]
  }
  else if(output_layer_act == "softmax"){
    gradients[[paste("dA", L, sep = "")]] <- (loop_back_reg_vals[['dA_prev']])
  }
  gradients[[paste("dW", L, sep = "")]] <- loop_back_reg_vals[['dW']]
  gradients[[paste("db", L, sep = "")]] <- loop_back_reg_vals[['db']]

  for(l in (L-1):1){
    current_cache = caches[[l]]$cache

    if (lambd == 0 & keep_prob < 1){
      D <- dropout_matrix[[paste("D", l, sep = "")]]
```

```
      # Multiply gradient with dropout matrix &
      # divide by keep_prob to keep expected value same
      gradients[[paste('dA', l + 1, sep = "")]] =
        gradients[[paste('dA', l + 1, sep = "")]] * D / keep_prob
      loop_back_vals <- back_prop_Reg_helper(gradients[[paste('dA',
                                                              l + 1,
                                                              sep = "")]],
                                             current_cache,
                                             X,
                                             Y,
                                             num_classes,
                                             hidden_layer_act[[l]],
                                             lambd)
    }
    else if(lambd != 0 & keep_prob == 1){
      loop_back_vals = back_prop_Reg_helper(gradients[[paste('dA',
                                                             l + 1,
                                                             sep = "")]],
                                            current_cache,
                                            X,
                                            Y,
                                            num_classes,
                                            hidden_layer_act[[l]],
                                            lambd)
    }
    else if(lambd == 0 & keep_prob == 1){
          loop_back_vals = back_prop_Reg_helper(gradients[[paste('dA',
                                                                 l + 1,
                                                                 sep = "")]],
                                                current_cache,
                                                X,
                                                Y,
                                                num_classes,
                                                hidden_layer_act[[l]],
                                                lambd = 0)
    }

    gradients[[paste("dA", l, sep = "")]] <- loop_back_vals[['dA_prev']]
    gradients[[paste("dW", l, sep = "")]] <- loop_back_vals[['dW']]
    gradients[[paste("db", l, sep = "")]] <- loop_back_vals[['db']]
  }

return(gradients)
}
```

```
model_with_Reg <- function(X,
                           Y,
                           X_test,
                           Y_test,
                           num_classes,
                           layers_dims,
                           hidden_layer_act,
                           output_layer_act,
                           optimizer,
                           learning_rate,
                           mini_batch_size,
                           num_epochs,
                           initialization,
                           beta,
                           beta1,
```

```
                          beta2,
                          epsilon,
                          keep_prob,
                          lambd,
                          verbose = F){

start_time <- Sys.time()
costs <- NULL
converged = FALSE
param <- NULL
t = 0
iter = 0
set.seed = 1
seed = 10
parameters = initialize_params(layers_dims, initialization)
v = initialize_adam(parameters)[["V"]]
s = initialize_adam(parameters)[["S"]]
velocity = initialize_velocity(parameters)

for(i in 0:num_epochs){
  seed = seed + 1
  iter = iter + 1
  minibatches = random_mini_batches(X, Y, mini_batch_size, seed)

  for(batch in 1:length(minibatches)){
    mini_batch_X = (minibatches[[batch]][['mini_batch_X']])
    mini_batch_Y = minibatches[[batch]][['mini_batch_Y']]

    if(keep_prob == 1){
      AL = forward_prop(mini_batch_X,
                        parameters,
                        hidden_layer_act,
                        output_layer_act)[['AL']]
      caches = forward_prop(mini_batch_X,
                            parameters,
                            hidden_layer_act,
                            output_layer_act)[['caches']]
    }
    else if(keep_prob < 1){
    AL = forward_prop_Reg(mini_batch_X,
                          parameters,
                          hidden_layer_act,
                          output_layer_act,
                          keep_prob)[['AL']]
    caches = forward_prop_Reg(mini_batch_X,
                              parameters,
                              hidden_layer_act,
                              output_layer_act,
                              keep_prob)[['caches']]
    dropout_matrix = forward_prop_Reg(mini_batch_X,
                                      parameters,
                                      hidden_layer_act,
                                      output_layer_act,
                                      keep_prob)[['dropout_matrix']]
    }
    # Compute Cost
      cost <- compute_cost_with_Reg(AL,
                                    mini_batch_X,
                                    mini_batch_Y,
                                    num_classes,
```

```
                                    parameters,
                                    lambd,
                                    output_layer_act)

  # Backward propagation
  if(lambd == 0 & keep_prob == 1){
    gradients = back_prop_Reg(AL,
                              mini_batch_X,
                              mini_batch_Y,
                              num_classes,
                              caches,
                              hidden_layer_act,
                              output_layer_act,
                              keep_prob = 1,
                              dropout_matrix = NULL,
                              lambd = 0)
  }
  else if(lambd != 0 & keep_prob == 1){
    gradients = back_prop_Reg(AL,
                              mini_batch_X,
                              mini_batch_Y,
                              num_classes,
                              caches,
                              hidden_layer_act,
                              output_layer_act,
                              keep_prob = 1,
                              dropout_matrix = NULL,
                              lambd)
  }
  else if(lambd == 0 & keep_prob < 1){
    gradients = back_prop_Reg(AL,
                              mini_batch_X,
                              mini_batch_Y,
                              num_classes,
                              caches,
                              hidden_layer_act,
                              output_layer_act,
                              keep_prob = 1,
                              dropout_matrix,
                              lambd = 0)
  }

  if(optimizer == 'gd'){
    parameters = update_params(parameters, gradients, learning_rate)
  }
  else if(optimizer == 'momentum'){
    parameters = update_params_with_momentum(parameters,
                                             gradients,
                                             velocity,
                                             beta,
                                             learning_rate)[["parameters"]]
    velocity = update_params_with_momentum(parameters,
                                           gradients,
                                           velocity,
                                           beta,
                                           learning_rate)[["Velocity"]]
  }
  else if(optimizer == 'adam'){
    t = t + 1
    parameters = update_params_with_adam(parameters,
```

```
                                          gradients,
                                          v,
                                          s,
                                          t,
                                          beta1,
                                          beta2,
                                          learning_rate,
                                          epsilon)[["parameters"]]
      v = update_params_with_adam(parameters,
                                  gradients,
                                  v,
                                  s,
                                  t,
                                  beta1,
                                  beta2,
                                  learning_rate,
                                  epsilon)[["Velocity"]]
      s = update_params_with_adam(parameters,
                                  gradients,
                                  v,
                                  s,
                                  t,
                                  beta1,
                                  beta2,
                                  learning_rate,
                                  epsilon)[["S"]]
    }
  }

  if(verbose == T & (iter - 1) %% 10000 == 0){
    print(paste0("Cost after iteration " , iter - 1, ' = ', cost, sep = ' '))
  }
  if((iter - 1) %% 100 == 0){
    costs = c(costs,cost)
  }
}

if(output_layer_act != 'softmax'){
  pred_train <- predict_model(parameters,
                              X,
                              hidden_layer_act,
                              output_layer_act)
  Tr_acc <- mean(pred_train == Y) * 100

  pred_test <- predict_model(parameters,
                             X_test,
                             hidden_layer_act,
                             output_layer_act)
  Ts_acc <- mean(pred_test == Y_test) * 100

  cat(sprintf("Cost after iteration  %d, = %05f;
              Train Acc: %#.3f, Test Acc: %#.3f, \n",
              i, cost, Tr_acc, Ts_acc))
  }
else if(output_layer_act != 'softmax'){
  pred_train <- predict_model(parameters,
                              X,
                              hidden_layer_act,
                              output_layer_act)
  Tr_acc <- mean((pred_train - 1) == Y)
```

```
    pred_test <- predict_model(parameters,
                               X_test,
                               hidden_layer_act,
                               output_layer_act)
    Ts_acc <- mean((pred_test - 1) == Y_test)

    cat(sprintf("Cost after iteration , %d, = %05f;
                Train Acc: %#.3f, Test Acc: %#.3f, \n",
                i, cost, Tr_acc, Ts_acc))
  }
  end_time <- Sys.time()

  cat(sprintf("Application running time: %#.3f minutes", end_time - start_time ))

return(list("parameters" = parameters, "costs" = costs))
}
```

```
layers_dims <- c(2, 100, 1)

model_batch_with_no_Reg <- model_with_Reg(scale.trainX,
                                          trainY,
                                          scale.testX,
                                          testY,
                                          num_classes = length(unique(trainY)),
                                          layers_dims,
                                          hidden_layer_act = c('relu'),
                                          output_layer_act = 'sigmoid',
                                          optimizer = 'gd',
                                          learning_rate = 0.0001,
                                          mini_batch_size = 600,
                                          num_epochs = 15000,
                                          initialization = 'He',
                                          beta = 0.9,
                                          beta1 = 0.9,
                                          beta2 = 0.999,
                                          epsilon = 1e-8,
                                          keep_prob = 1,
                                          lambd = 0,
                                          verbose = T)
```

```
[1] "Cost after iteration 0 = 1.47048780895349 "
[1] "Cost after iteration 10000 = 0.540938410678508 "
Cost after iteration  15000, = 0.522032;
                Train Acc: 70.167, Test Acc: 66.500,
Application running time: 2.169 minutes
```

```
layers_dims <- c(2, 100, 1)

model_mb_with_no_Reg <- model_with_Reg(scale.trainX,
                                       trainY,
                                       scale.testX,
                                       testY,
                                       num_classes = length(unique(trainY)),
                                       layers_dims,
                                       hidden_layer_act = c('relu'),
                                       output_layer_act = 'sigmoid',
                                       optimizer = 'gd',
                                       learning_rate = 0.0001,
```

```
                                    mini_batch_size = 64,
                                    num_epochs = 15000,
                                    initialization = 'He',
                                    beta = 0.9,
                                    beta1 = 0.9,
                                    beta2 = 0.999,
                                    epsilon = 1e-8,
                                    keep_prob = 1,
                                    lambd = 0,
                                    verbose = T)
```

```
[1] "Cost after iteration 0 = 1.30788569097192 "
[1] "Cost after iteration 10000 = 0.350172075944783 "
Cost after iteration  15000, = 0.228268;
                Train Acc: 91.667, Test Acc: 90.500,
Application running time: 4.418 minutes
```

```
layers_dims <- c(2, 100, 1)

model_mb_with_Mom_no_Reg <- model_with_Reg(scale.trainX,
                                           trainY,
                                           scale.testX,
                                           testY,
                                           num_classes = length(unique(trainY)),
                                           layers_dims,
                                           hidden_layer_act = 'relu',
                                           output_layer_act = 'sigmoid',
                                           optimizer = 'momentum',
                                           learning_rate = 0.0001,
                                           mini_batch_size = 64,
                                           num_epochs = 15000,
                                           initialization = 'He',
                                           beta = 0.9,
                                           beta1 = 0.9,
                                           beta2 = 0.999,
                                           epsilon = 1e-8,
                                           keep_prob = 1,
                                           lambd = 0,
                                           verbose = T)
```

```
[1] "Cost after iteration 0 = 1.31294231878059 "
[1] "Cost after iteration 10000 = 0.350180602995558 "
Cost after iteration  15000, = 0.228286;
                Train Acc: 91.667, Test Acc: 90.500,
Application running time: 4.379 minutes
```

```
layers_dims <- c(2, 100, 1)

model_mb_with_Adam_no_Reg <- model_with_Reg(scale.trainX,
                                            trainY,
                                            scale.testX,
                                            testY,
                                            num_classes = length(unique(trainY)),
                                            layers_dims,
                                            hidden_layer_act = 'relu',
                                            output_layer_act = 'sigmoid',
```

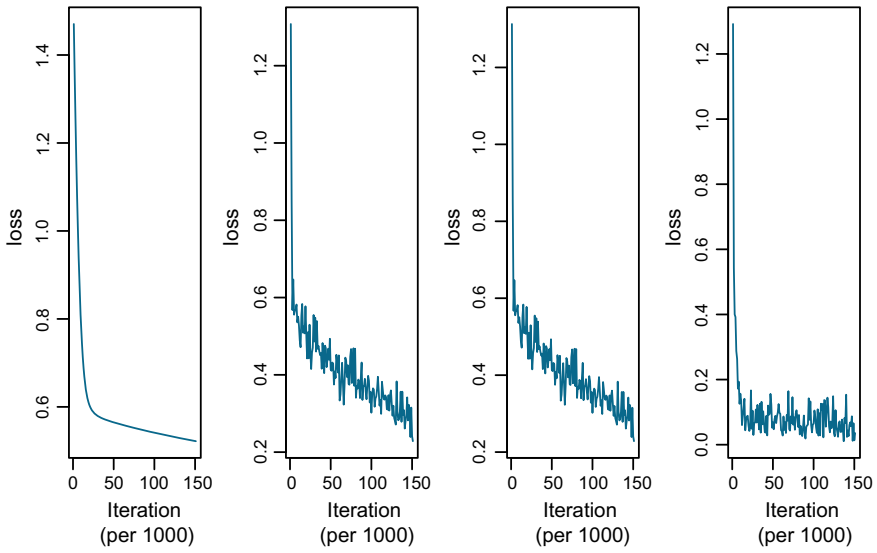

Fig. 5.10 Gradient descent with no regularization: loss versus iteration. Starting from left batch gradient descent, mini-batch gradient descent, mini-batch gradient descent with momentum, and mini-batch gradient descent with Adam

```
                                        optimizer = 'adam',
                                        learning_rate = 0.0001,
                                        mini_batch_size = 64,
                                        num_epochs = 15000,
                                        initialization = 'He',
                                        beta = 0.9,
                                        beta1 = 0.9,
                                        beta2 = 0.999,
                                        epsilon = 1e-8,
                                        keep_prob = 1,
                                        lambd = 0,
                                        verbose = T)
```

```
[1] "Cost after iteration 0 = 1.29201230219011 "
[1] "Cost after iteration 10000 = 0.104942563682952 "
Cost after iteration  15000, = 0.035551;
                Train Acc: 97.333, Test Acc: 95.000,
Application running time: 5.825 minutes
```

Now, let us try out our algorithm with the dropout probability `keep_prob` = 0.8, i.e., at every iteration we will shut down the neurons of layers 1 and 2 with 20% probability.

```
layers_dims <- c(2, 100, 1)
model_batch_dropout <- model_with_Reg(scale.trainX,
                                      trainY,
                                      scale.testX, testY,
```

```
                                    num_classes = length(unique(trainY)),
                                    layers_dims,
                                    hidden_layer_act = 'relu',
                                    output_layer_act = 'sigmoid',
                                    optimizer = 'gd',
                                    learning_rate = 0.0001,
                                    mini_batch_size = 600,
                                    num_epochs = 15000,
                                    initialization = 'He',
                                    beta = 0.9,
                                    beta1 = 0.9,
                                    beta2 = 0.999,
                                    epsilon = 1e-8,
                                    keep_prob = 0.8,
                                    lambd = 0,
                                    verbose = T)
```

```
[1] "Cost after iteration 0 = 1.70923892863993 "
[1] "Cost after iteration 10000 = 0.539889679943393 "
Cost after iteration  15000, = 0.516557;
                Train Acc: 72.167, Test Acc: 70.500,
Application running time: 8.729 minutes
```

```
model_mb_dropout <- model_with_Reg(scale.trainX,
                                   trainY,
                                   scale.testX,
                                   testY,
                                   num_classes = length(unique(trainY)),
                                   layers_dims,
                                   hidden_layer_act = 'relu',
                                   output_layer_act = 'sigmoid',
                                   optimizer = 'gd',
                                   learning_rate = 0.0001,
                                   mini_batch_size = 64,
                                   num_epochs = 15000,
                                   initialization = 'He',
                                   beta = 0.9,
                                   beta1 = 0.9,
                                   beta2 = 0.999,
                                   epsilon = 1e-8,
                                   keep_prob = 0.8,
                                   lambd = 0,
                                   verbose = T)
```

```
[1] "Cost after iteration 0 = 1.49581082005902 "
[1] "Cost after iteration 10000 = 0.334804409527771 "
Cost after iteration  15000, = 0.223873;
                Train Acc: 91.333, Test Acc: 92.000,
Application running time: 8.917 minutes
```

```
model_mb_mom_dropout <- model_with_Reg(scale.trainX,
                                       trainY,
                                       scale.testX,
                                       testY,
                                       num_classes = length(unique(trainY)),
                                       layers_dims,
```

```
                                    hidden_layer_act = 'relu',
                                    output_layer_act = 'sigmoid',
                                    optimizer = 'momentum',
                                    learning_rate = 0.0001,
                                    mini_batch_size = 64,
                                    num_epochs = 15000,
                                    initialization = 'He',
                                    beta = 0.9,
                                    beta1 = 0.9,
                                    beta2 = 0.999,
                                    epsilon = 1e-8,
                                    keep_prob = 0.8,
                                    lambd = 0,
                                    verbose = T)
```

```
[1] "Cost after iteration 0 = 1.50436105247378 "
[1] "Cost after iteration 10000 = 0.33481409021011 "
Cost after iteration  15000, = 0.223898;
                Train Acc: 91.333, Test Acc: 92.000,
Application running time: 8.964 minutes
```

```
model_mb_adam_dropout <- model_with_Reg(scale.trainX,
                                    trainY,
                                    scale.testX,
                                    testY,
                                    num_classes = length(unique(trainY)),
                                    layers_dims,
                                    hidden_layer_act = 'relu',
                                    output_layer_act = 'sigmoid',
                                    optimizer = 'adam',
                                    learning_rate = 0.0001,
                                    mini_batch_size = 64,
                                    num_epochs = 15000,
                                    initialization = 'He',
                                    beta = 0.9,
                                    beta1 = 0.9,
                                    beta2 = 0.999,
                                    epsilon = 1e-8,
                                    keep_prob = 0.8,
                                    lambd = 0,
                                    verbose = T)
```

```
[1] "Cost after iteration 0 = 1.47770684841945 "
[1] "Cost after iteration 10000 = 0.110124099690202 "
Cost after iteration  15000, = 0.042310;
                Train Acc: 97.667, Test Acc: 96.000,
Application running time: 10.476 minutes
```

L2 Regularization

We will now set the penalty $\lambda = 0.6$ in ℓ_2 regularization and compare our results (Fig. 5.12).

```
layers_dims <- c(2, 100, 1)
model_batch_l2 <- model_with_Reg(scale.trainX,
                                 trainY,
```

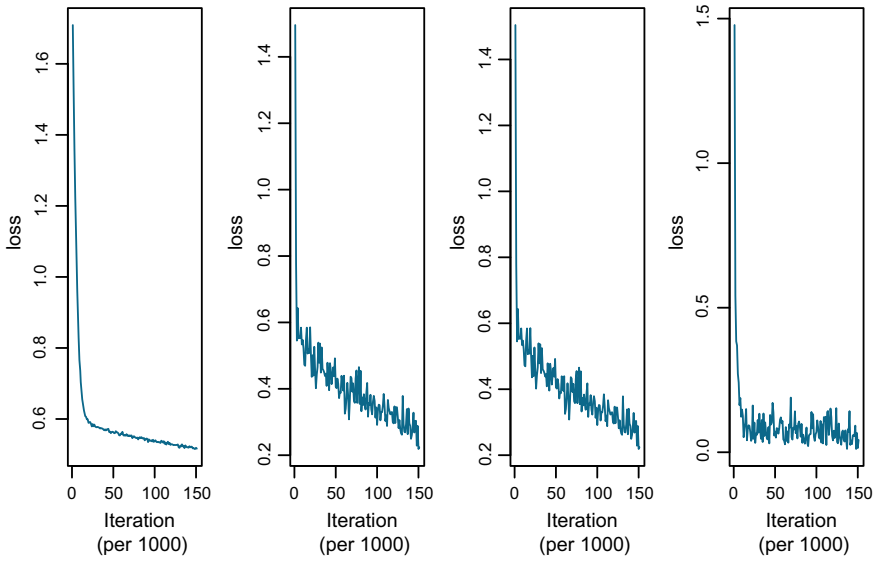

Fig. 5.11 Gradient descent with dropout regularization, (dropout probability = 0.2): loss versus iteration plot. Starting from left batch gradient descent, mini-batch gradient descent, mini-batch gradient descent with momentum, and mini-batch gradient descent with Adam

```
                                scale.testX,
                                testY,
                                num_classes = length(unique(trainY)),
                                layers_dims,
                                hidden_layer_act = 'relu',
                                output_layer_act = 'sigmoid',
                                optimizer = 'gd',
                                learning_rate = 0.0001,
                                mini_batch_size = 600,
                                num_epochs = 15000,
                                initialization = 'He',
                                beta = 0.9,
                                beta1 = 0.9,
                                beta2 = 0.999,
                                epsilon = 1e-8,
                                keep_prob = 1,
                                lambd = 0.6,
                                verbose = T)
```

```
[1] "Cost after iteration 0 = 1.58639145691037 "
[1] "Cost after iteration 10000 = 0.656553136817669 "
Cost after iteration  15000, = 0.637578;
                Train Acc: 70.167, Test Acc: 66.500,
Application running time: 2.187 minutes
```

```
model_minibatch_l2 <- model_with_Reg(scale.trainX,
                                     trainY,
```

```
                                    scale.testX,
                                    testY,
                                    num_classes = length(unique(trainY)),
                                    layers_dims,
                                    hidden_layer_act = 'relu',
                                    output_layer_act = 'sigmoid',
                                    optimizer = 'gd',
                                    learning_rate = 0.0001,
                                    mini_batch_size = 64,
                                    num_epochs = 15000,
                                    initialization = 'He',
                                    beta = 0.9,
                                    beta1 = 0.9,
                                    beta2 = 0.999,
                                    epsilon = 1e-8,
                                    keep_prob = 1,
                                    lambd = 0.6,
                                    verbose = T)
```

```
[1] "Cost after iteration 0 = 2.39443387207555 "
[1] "Cost after iteration 10000 = 1.26264735343363 "
Cost after iteration  15000, = 1.068367;
                Train Acc: 91.500, Test Acc: 91.000,
Application running time: 2.868 minutes
```

```
model_minibatch_mom_l2 <- model_with_Reg(scale.trainX,
                                         trainY,
                                         scale.testX,
                                         testY,
                                         num_classes = length(unique(trainY)),
                                         layers_dims,
                                         hidden_layer_act = 'relu',
                                         output_layer_act = 'sigmoid',
                                         optimizer = 'momentum',
                                         learning_rate = 0.0001,
                                         mini_batch_size = 64,
                                         num_epochs = 15000,
                                         initialization = 'He',
                                         beta = 0.9,
                                         beta1 = 0.9,
                                         beta2 = 0.999,
                                         epsilon = 1e-8,
                                         keep_prob = 1,
                                         lambd = 0.6,
                                         verbose  = T)
```

```
[1] "Cost after iteration 0 = 2.39952012968536 "
[1] "Cost after iteration 10000 = 1.26266151545024 "
Cost after iteration  15000, = 1.068387;
                Train Acc: 91.500, Test Acc: 91.000,
Application running time: 3.149 minutes
```

```
model_minibatch_adam_l2 <- model_with_Reg(scale.trainX,
                                          trainY,
                                          scale.testX,
                                          testY,
```

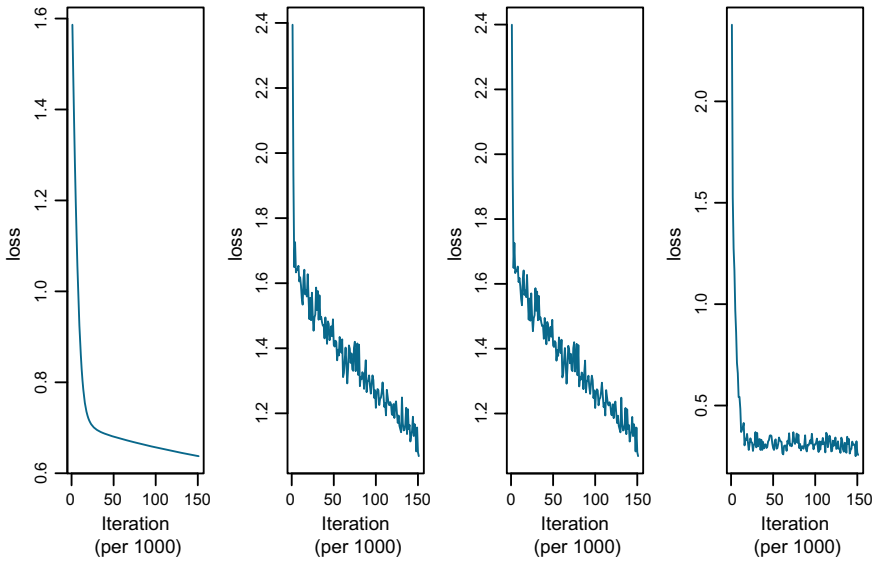

Fig. 5.12 Gradient descent with l_2 regularization, ($l_2 = 0.6$): loss vs. iteration plot. Starting from left batch gradient descent, mini-batch gradient descent, mini-batch gradient descent with momentum, and mini-batch gradient descent with Adam

```
num_classes = length(unique(trainY)),
layers_dims,
hidden_layer_act = 'relu',
output_layer_act = 'sigmoid',
optimizer = 'adam',
learning_rate = 0.0001,
mini_batch_size = 64,
num_epochs = 15000,
initialization = 'He',
beta = 0.9,
beta1 = 0.9,
beta2 = 0.999,
epsilon = 1e-8,
keep_prob = 1,
lambd = 0.6,
verbose = T)
```

```
[1] "Cost after iteration 0 = 2.37800789224143 "
[1] "Cost after iteration 10000 = 0.316931955547848 "
Cost after iteration  15000, = 0.256161;
                Train Acc: 96.500, Test Acc: 95.500,
Application running time: 4.026 minutes
```

Figure 5.12 depicts the loss, using l_2 regularization for batch gradient descent, mini-batch gradient descent, mini-batch gradient descent with momentum and mini-batch gradient descent with Adam.

5.5.3 Combining Dropout and ℓ_2 Regularization?

It is still not clear whether using both at the same time is beneficial or if it makes things more complicated. While ℓ_2 regularization is implemented to penalize high values of the structural parameters, dropout involves a random process of switching-off or dropping some nodes, which cannot be expressed as a penalty term. However, they both try to avoid the network's overreliance on spurious correlations, which are one of the consequences of overtraining.

Having said that, it would be prudent to say that more detailed research is necessary to determine how and when, they can work together or, end up negating each other. So far, it seems the results tend to vary in a case-by-case fashion.

You may try your experiments and maybe, you can add to the knowledge base, on their suitability or otherwise, and the conditions of use, by means of further research on this topic.

5.6 Gradient Checking

As we have seen so far, in neural networks, we apply the forward propagation function, calculate the error function $J(\theta)$, and then compute the derivatives of the error function $\frac{d}{d\theta}J(\theta)$, with the backpropagation function.

Backpropagation is a difficult algorithm to debug and to get right and therefore, it is difficult to check whether our backpropagation is actually working.

Consider that we want to minimize $J(\theta)$ using gradient descent. An iteration of gradient descent can be written as

$$\theta := \theta - \alpha \frac{d}{d\theta} J(\theta). \tag{5.6.1}$$

How do we check if our implementation of $\frac{d}{d\theta}J(\theta)$ is correct? We know from calculus that the mathematical definition of the derivative is

$$\frac{d}{d\theta} J(\theta) = \lim_{\epsilon \to 0} \frac{J(\theta + \epsilon) - J(\theta - \epsilon)}{2\epsilon}. \tag{5.6.2}$$

To derive confidence in our implementation, let us once again implement forward propagation for some given input values.

But before we do that, we write two functions, which will unlist the elements in the caches and convert them to a vector.

```
dic2vec_params <- function(parameters) {
    p = unlist(list(W1 = parameters[["W1"]], b1 = parameters[["b1"]],
        W2 = parameters[["W2"]], b2 = parameters[["b2"]]))
    return(p)
```

```
}

dic2vec_grads <- function(gradientss) {
    g = unlist(list(dW1 = gradients[["dW1"]], db1 = gradients[["db1"]],
        dW2 = gradients[["dW2"]], db2 = gradients[["db2"]]))
    return(g)
}
```

The `forward_prop_check` function computes the cost and stores the network parameters in a cache, to be used during our gradient checking.

```
forward_prop_check <- function(X, Y, parameters) {
    m = dim(X)[2]
    W1 = parameters[["W1"]]
    b1 = parameters[["b1"]]
    W2 = parameters[["W2"]]
    b2 = parameters[["b2"]]

    Z1 = sweep(W1 %*% X, 1, b1, "+")
    A1 = relu(Z1)[["A"]]
    Z2 = sweep(W2 %*% A1, 1, b2, "+")
    A2 = sigmoid(Z2)[["A"]]

    logprobs = -(log(A2) %*% Y + log(1 - A2) %*% (1 - Y))
    cost = 1/m * sum(logprobs)

    cache = list(Z1 = Z1, A1 = A1, W1 = W1, b1 = b1, Z2 = Z2,
        A2 = A2, W2 = W2, b2 = b2)
    return(list(cost = cost, cache = cache))
}
```

As forward propagation is relatively easy to implement, we are quite sure about computing the cost J correctly. Therefore we can use our computed J values to verify the code for computing $\frac{\partial J}{\partial W}$ to arrive at the approximated computed value of the gradient. In the following function, we calculate this value and store it in `gradapprox` as follows:

$gradapprox = \frac{J(W+\epsilon)-J(W-\epsilon)}{2\epsilon}$. The calculated gradients are stored in `grad`. Finally we compute the relative difference between `gradapprox` and the `grad` using the ℓ_2 norm as follows-

$diff = \frac{\|grad-gradapprox\|_2}{\|grad\|_2+\|gradapprox\|_2}$

The ℓ_2 norm of a vector $\mathbf{a} = [a_1, a_2, a_3]$ is $\|\mathbf{a}\| = \sqrt{a_1^2 + a_2^2 + a_3^2}$. In R, the function to calculate the ℓ_2 norm is `vector.norm()` from the `InspectChangepoint` package.

```
gradient_check <- function(parameters, gradients, X, Y, epsilon = 1e-07) {
    parameters_values = dic2vec_params(parameters)
    grad = dic2vec_grads(gradients)

    num_parameters = length(parameters_values)
    J_plus = rep(0, num_parameters)
    J_minus = rep(0, num_parameters)
    gradapprox = rep(0, num_parameters)

    for (i in 1:num_parameters) {
```

```
        thetaplus = parameters_values
        length(thetaplus)
        thetaplus[i] = thetaplus[i] + epsilon
        J_plus[[i]] = forward_prop_check(X, Y, relist(unlist(thetaplus),
            parameters))[["cost"]]
        # forward_prop(X, Y, relist(unlist(thetaplus),
        # parameters))[['cost']]
        thetaminus = parameters_values
        thetaminus[i] = thetaminus[i] - epsilon
        J_minus[[i]] = forward_prop_check(X, Y, relist(unlist(thetaminus),
            parameters))[["cost"]]

        gradapprox[i] = (J_plus[i] - J_minus[i])/(2 * epsilon)
    }
    num = vector.norm((grad - gradapprox))
    den = vector.norm(grad) + vector.norm(gradapprox)
    diff = num/den

    if (diff < 1e-07) {
        print("The gradient is correct!")
    } else {
        print("The gradient is wrong!")
    }
    return(diff)
}
```

We will be using the `back_prop` function, to get are our network gradients and use them in the `gradient_check` function, above.

```
set.seed(1)
index = sample(1:600, 1)
X = as.matrix(scale.trainX[, index])
Y = trainY[index]

parameters = initialize_params(layers_dims = c(2, 100, 2), initialization = "random")
AL = forward_prop(X, parameters, "relu", "sigmoid")[["AL"]]
caches = forward_prop(X, parameters, "relu", "sigmoid")[["caches"]]
gradients <- back_prop(AL, Y, caches, hidden_layer_act = "relu",
    output_layer_act = "sigmoid")

gradient_check(parameters, gradients, X, Y, epsilon = 1e-07)
```

```
[1] "The gradient is correct!"
```

```
[1] 1.070501e-08
```

Our implementation of backpropagation is doing good! The difference is 1.07-08

> Gradient checking does not work when applying dropout regularization. Use `keep-prob` = 1 during gradient checking and then change it during training. Epsilon = $10e-7$ is a common value used for the difference between analytical gradient and numerical gradient. If the difference is less than $10e-7$, then the implementation of backpropagation is correct.

Since gradient checking is very slow we apply it on one or a few training examples. We need to turn it off when training a neural network after making sure that the backpropagation's implementation is correct.

5.7 Conclusion

We have come far from our simple neural network algorithm.

We are now aware about most of the hyperparameters which go into a neural network. We also have a fair idea about vanishing and exploding gradients, and we have also devised a method wherein we can check if our backpropagation algorithm is working.

We are now ready to experiment with deeper networks, which we will discuss in the next chapter.

Chapter 6
Deep Neural Networks-II

The purpose of computing is insights, not numbers.
R.W. Hamming

Abstract We will implement a multi-layered neural network with different hyper-parameters

- Hidden layer activations
- Hidden layer nodes
- Output layer activation
- Learning rate
- Mini-batch size
- Initialization
- Value of β
- Values of β_1
- Value of β_2
- Value of ϵ
- Value of `keep_prob`
- Value of λ
- Model training time

6.1 Revisiting DNNs

In the previous chapter, we have explored many different optimization algorithms and regularization techniques. We will now explore multi-label classification using the `softmax` function in the output layer.

© Springer Nature Singapore Pte Ltd. 2019

A. Ghatak, *Deep Learning with R*,
https://doi.org/10.1007/978-981-13-5850-0_6

```
DNN_model <- function(X,
                      Y,
                      X_test,
                      Y_test,
                      layers_dims,
                      hidden_layer_act,
                      output_layer_act,
                      optimizer,
                      learning_rate,
                      mini_batch_size,
                      num_epochs,
                      initialization,
                      beta,
                      beta1,
                      beta2,
                      epsilon,
                      keep_prob,
                      lambd,
                      print_cost = F){

  start_time <- Sys.time()
  costs <- NULL
  converged = FALSE
  param <- NULL
  t = 0
  iter = 0
  set.seed = 1
  seed = 10
  num_classes = length(unique(Y))
  parameters = initialize_params(layers_dims, initialization)
  v = initialize_adam(parameters)[["V"]]
  s = initialize_adam(parameters)[["S"]]
  velocity = initialize_velocity(parameters)

  for(i in 0:num_epochs){
    seed = seed + 1
    iter = iter + 1
    minibatches = random_mini_batches(X, Y, mini_batch_size, seed)

    for(batch in 1:length(minibatches)){
      mini_batch_X = (minibatches[[batch]][['mini_batch_X']])
      mini_batch_Y = minibatches[[batch]][['mini_batch_Y']]

      if(keep_prob == 1){
        AL = forward_prop(mini_batch_X, parameters, hidden_layer_act,
                          output_layer_act)[['AL']]
        caches = forward_prop(mini_batch_X, parameters, hidden_layer_act,
                              output_layer_act)[['caches']]
      }
      else if(keep_prob < 1){
        AL = forward_prop_Reg(mini_batch_X, parameters, hidden_layer_act,
                                        output_layer_act, keep_prob)[['AL']]
        caches = forward_prop_Reg(mini_batch_X, parameters,
                                            hidden_layer_act,
                                            output_layer_act,
                                            keep_prob)[['caches']]
        dropout_matrix = forward_prop_Reg(mini_batch_X, parameters,
                                                    hidden_layer_act,
                                                    output_layer_act,
                                                    keep_prob)[['dropout_matrix']]
      }
```

```
        cost <- compute_cost_with_Reg(AL, mini_batch_X, mini_batch_Y,
                                      num_classes,
                                      parameters,
                                      lambd,
                                      output_layer_act)

      # Backward propagation
      if(lambd == 0 & keep_prob == 1){
        gradients = back_prop(AL, mini_batch_Y, caches,
                              hidden_layer_act, output_layer_act)
      }
      else if(lambd != 0 & keep_prob == 1){
        gradients = back_prop_Reg(AL, mini_batch_X, mini_batch_Y,
                                          num_classes,
                                          caches, hidden_layer_act,
                                          output_layer_act, keep_prob = 1,
                                          dropout_matrix, lambd)
      }
      else if(lambd == 0 & keep_prob < 1){
        gradients = back_prop_Reg(AL, mini_batch_X, mini_batch_Y,
                                          num_classes, caches,
                                          hidden_layer_act,
                                          output_layer_act, keep_prob,
                                          dropout_matrix, lambd = 0)
      }

      if(optimizer == 'gd'){
        parameters = update_params(parameters, gradients, learning_rate)
      }
      else if(optimizer == 'momentum'){
        parameters = update_params_with_momentum(parameters, gradients, velocity,
                                                 beta,
                                                 learning_rate)[["parameters"]]
        velocity = update_params_with_momentum(parameters, gradients, velocity,
                                               beta,
                                               learning_rate)[["Velocity"]]
      }
      else if(optimizer == 'adam'){
        t = t + 1
        parameters = update_params_with_adam(parameters, gradients, v, s, t,
                                             beta1, beta2,
                                             learning_rate,
                                             epsilon)[["parameters"]]

        v = update_params_with_adam(parameters, gradients, v, s, t,
                                    beta1, beta2,
                                    learning_rate,
                                    epsilon)[["Velocity"]]
        s = update_params_with_adam(parameters, gradients, v, s, t,
                                    beta1, beta2,
                                    learning_rate,
                                    epsilon)[["S"]]
      }
    }
        costs <- append(costs, list(cost))

        if(print_cost == T & i %% 1000 == 0){
          cat(sprintf("Cost after epoch %d = %05f\n", i, cost))
        }
  }
```

```
  if(output_layer_act != 'softmax'){
      pred_train <- predict_model(parameters, X,
                                  hidden_layer_act,
                                  output_layer_act)
      Tr_acc <- mean(pred_train == Y) * 100
      pred_test <- predict_model(parameters, X_test,
                                 hidden_layer_act,
                                 output_layer_act)
      Ts_acc <- mean(pred_test == Y_test) * 100
      cat(sprintf("Cost after epoch  %d, = %05f;
              Train Acc: %#.3f, Test Acc: %#.3f, \n",
              i, cost, Tr_acc, Ts_acc))
      }
      else if(output_layer_act == 'softmax'){
        pred_train <- predict_model(parameters, X,
                                    hidden_layer_act, output_layer_act)
        Tr_acc <- mean((pred_train - 1) == Y)
        pred_test <- predict_model(parameters, X_test,
                                   hidden_layer_act, output_layer_act)
        Ts_acc <- mean((pred_test - 1) == Y_test)
        cat(sprintf("Cost after epoch , %d, = %05f;
                    Train Acc: %#.3f, Test Acc: %#.3f, \n",
                    i, cost, Tr_acc, Ts_acc))
      }

  end_time <- Sys.time()
  cat(sprintf("Application running time: %#.3f seconds\n",
              end_time - start_time ))

return(list("parameters" = parameters, "costs" = costs))
}
```

We will use the `MNIST`[1] data for multi-label classification.
The labels in the MNIST data set are a one-hot encoded matrix.

```
file_path <- "~/data/MNIST"

train <- read.csv(paste0(file_path, "mnist_train_images.csv",
    sep = ""), header = FALSE)
ytrain <- read.csv(paste0(file_path, "mnist_train_labels.csv",
    sep = ""), header = FALSE)

test <- read.csv(paste0(file_path, "mnist_test_images.csv", sep = ""),
    header = FALSE)
ytest <- read.csv(paste0(file_path, "mnist_test_labels.csv",
    sep = ""), header = FALSE)

dim(train)
```

[1] Downloaded from http://yann.lecun.com/exdb/mnist/.

```
[1] 55000   784
```

```
dim(ytrain)
```

```
[1] 55000    10
```

```
dim(test)
```

```
[1] 10000   784
```

```
dim(ytest)
```

```
[1] 10000    10
```

```
train <- data.matrix(train)
test <- data.matrix(test)
```

The labels in the MNIST data set are one-hot encoded. We will convert the one-hot encoded labels to a single column of labels.

```
ytr = ytrain
colnames(ytr) <- c(0:9)
head(ytr)
```

```
  0 1 2 3 4 5 6 7 8 9
1 0 0 0 0 0 0 0 1 0 0
2 0 0 0 1 0 0 0 0 0 0
3 0 0 0 0 1 0 0 0 0 0
4 0 0 0 0 0 0 1 0 0 0
5 0 1 0 0 0 0 0 0 0 0
6 0 0 0 0 0 0 0 0 1 0
```

```
w <- which(ytr == 1, arr.ind = T)
ytr$label <- toupper(names(ytr)[w[order(w[, 1]), 2]])
ytr = as.numeric(ytr$label)
head(ytr)
```

```
[1] 7 3 4 6 1 8
```

```
yts = ytest
colnames(yts) <- c(0:9)
head(yts)
```

```
  0 1 2 3 4 5 6 7 8 9
1 0 0 0 0 0 0 0 1 0 0
2 0 0 1 0 0 0 0 0 0 0
3 0 1 0 0 0 0 0 0 0 0
4 1 0 0 0 0 0 0 0 0 0
5 0 0 0 0 1 0 0 0 0 0
6 0 1 0 0 0 0 0 0 0 0
```

```
w <- which(yts == 1, arr.ind = T)
yts$label <- toupper(names(yts)[w[order(w[, 1]), 2]])
yts = as.numeric(yts$label)
head(yts)
```

```
[1] 7 2 1 0 4 1
```

The `MNIST` data set has 55000 observations in the training set and 10000 observations in the test set. We will consider a subset of the data set for training and validation.

The data is in gray scale with each image having 0 to 255 pixels. We therefore need to scale the data by the number of pixels.

We will use a multi-layered network with `relu` activations in the two hidden layers, having 30 and 10 nodes, respectively, and `softmax` activation in the output layer.

```
train_X <- t(train) / 256
test_X <- t(test) / 256

X_train <- train_X[, 1:5000]; y_train <- ytr[1:5000]
X_test <- test_X[, 1:1000]; y_test <- yts[1:1000]

dnn_1 <- DNN_model(X_train,
                   y_train,
                   X_test,
                   y_test,
                   layers_dims = c(nrow(X_train), 30, 10, 10),
                   hidden_layer_act = c('relu', 'relu'),
                   output_layer_act = 'softmax',
                   optimizer = 'adam',
                   learning_rate = 0.001,
                   mini_batch_size = 32,
                   num_epochs = 300,
                   initialization = 'He',
                   beta = 0.9,
                   beta1 = 0.9,
                   beta2 = 0.999,
                   epsilon = 1e-8,
                   keep_prob = 1,
                   lambd = 0.0001,
                   print_cost = T)
```

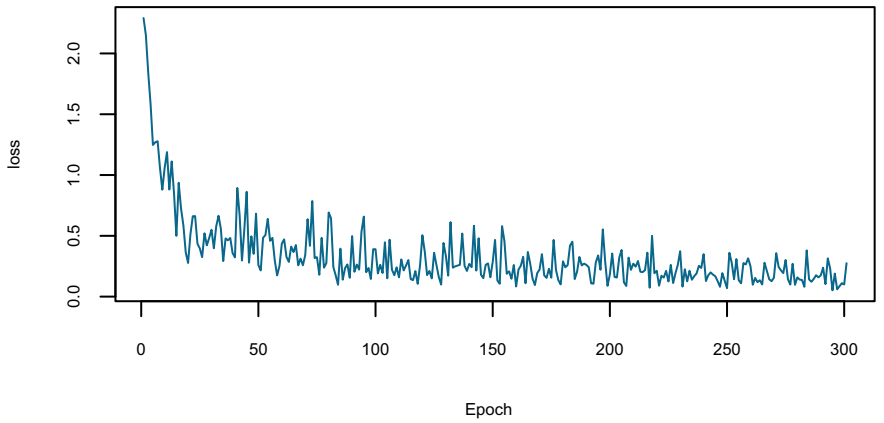

Fig. 6.1 Plot of loss versus epochs for a multi-layered network with relu activations in the two hidden layers having 30 and 10 nodes respectively, and softmax activation in the output layer

```
Cost after epoch 0 = 2.291371
Cost after epoch 300 = 0.275142;
                    Train Acc: 0.962, Test Acc: 0.861,
Application running time: 14.974 seconds
```

The loss for this model is plotted in Fig. 6.1.

```
dnn_2 <- DNN_model(X_train,
                   y_train,
                   X_test,
                   y_test,
                   layers_dims = c(nrow(X_train), 30, 10, 10, 10),
                   hidden_layer_act = c('relu', 'relu', 'relu'),
                   output_layer_act = 'softmax',
                   optimizer = 'adam',
                   learning_rate = 0.001,
                   mini_batch_size = 32,
                   num_epochs = 300,
                   initialization = 'He',
                   beta = 0.9,
                   beta1 = 0.9,
                   beta2 = 0.999,
                   epsilon = 1e-8,
                   keep_prob = 0.8,
                   lambd = 0,
                   print_cost = T)
```

```
Cost after epoch 0 = 2.139287
Cost after epoch , 300, = 0.054083;
                    Train Acc: 0.988, Test Acc: 0.866,
Application running time: 10.012 seconds
```

With `softmax` activation in the output layer and a deeper network with dropout regularization, keep_prob = 0.8, we have improved our train-set accuracy by 2.6%.

For our designed application, this is quite magnificent. I would urge you to play with the hyperparameters to achieve a train set accuracy of 97%, using only 10% of the available training data set with `softmax` in the output layer. The loss for a 3 layered model is plotted in Fig. 6.2.

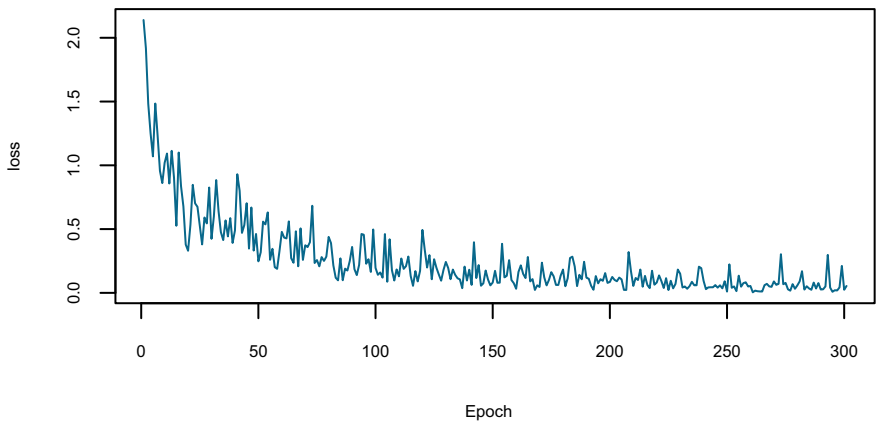

Fig. 6.2 Plot of loss versus epochs for a multi-layered network with relu activations in three hidden layers having 30, 10 and 10 nodes respectively, and softmax activation in the output layer

6.2 Modeling Using `keras`

```
scale.trainX.mat <- scale(trainX)
scale.testX.mat <- scale(testX)

set.seed(1)
model <- keras_model_sequential() %>%
  layer_dense(units = 400, activation = "relu", input_shape = 2) %>%
  layer_dense(units = 200, activation = "relu", input_shape = 2) %>%
  layer_dense(units = 50, activation = "relu") %>%
  layer_dense(units = 1, activation = 'sigmoid') %>%

  compile(
    optimizer = optimizer_adam(),
    loss = 'binary_crossentropy',
    metrics = 'accuracy'
  )
learn <- model %>% fit(scale.trainX.mat, trainY,
              epochs = 300,
              batch_size = 64,
              validation_split = 0.2,
              verbose = FALSE)
learn
```

```
Trained on 480 samples, validated on 120 samples (batch_size=64,epochs=300)

 Final epoch (plot to see history):
     acc: 0.9771
    loss: 0.05486
 val_acc: 0.95
val_loss: 0.1831
```

```
plot(learn)
```

The loss and accuracy for a 3 layered `keras` model is plotted in Fig. 6.3.

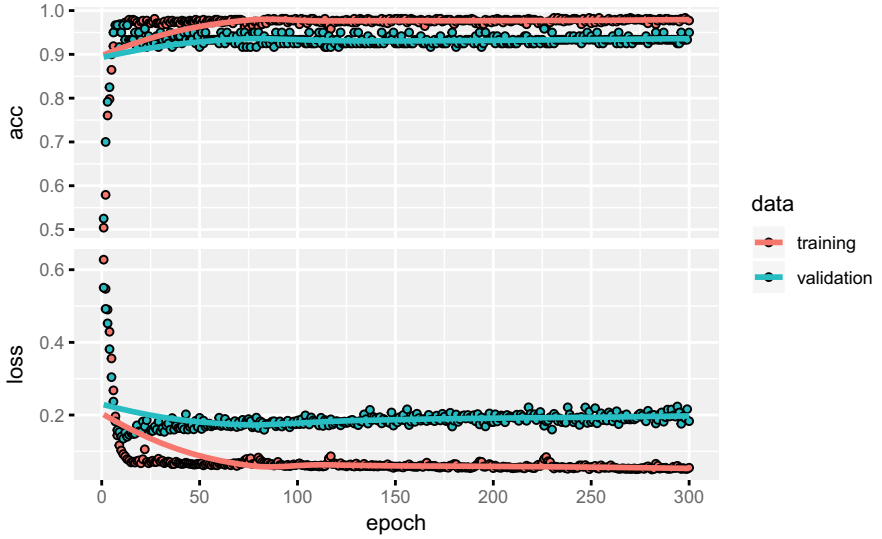

Fig. 6.3 Plot of accuracy and loss with respect to epochs using a keras model with three hidden layers having 400, 200 and 50 nodes with sigmoid activation in the output layer

```
set.seed(1)
model <- keras_model_sequential() %>%
  layer_dense(units = 175, activation = "relu", input_shape = 2) %>%
  layer_dense(units = 1, activation = 'sigmoid') %>%

  compile(
    optimizer = optimizer_adam(),
    loss = 'binary_crossentropy',
    metrics = 'accuracy'
  )
learn <- model %>% fit((scale.trainX.mat), trainY,
              epochs = 300,
              batch_size = 64,
              validation_split = 0.2,
              verbose = FALSE)
learn
```

```
Trained on 480 samples, validated on 120 samples
(batch_size=64, epochs=300)
Final epoch (plot to see history):
     acc: 0.9771
    loss: 0.06767
 val_acc: 0.9417
val_loss: 0.1491
```

```
plot(learn)
```

The loss and accuracy with a single layer keras model is plotted in Fig. 6.4.

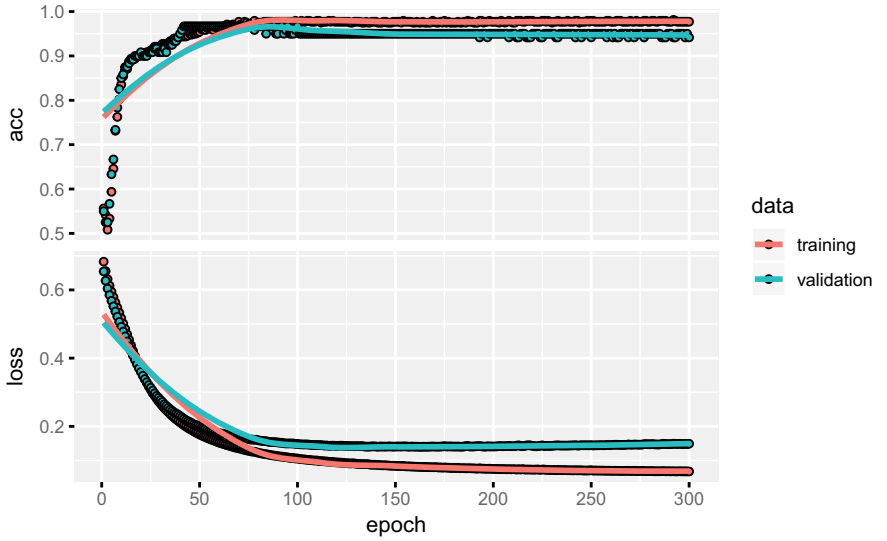

Fig. 6.4 Plot of accuracy and loss with respect to epochs using a keras model with one hidden layer having 175 nodes with sigmoid activation in the output layer

6.2.1 Adjust Epochs

Here, we incorporate `callback_early_stopping(patience = 100)` to stop training if the loss has not improved after 100 epochs.

```
set.seed(1)
model <- keras_model_sequential() %>%
  layer_dense(units = 175, activation = "relu", input_shape = 2) %>%
  layer_dense(units = 1, activation = 'sigmoid') %>%
  compile(
    optimizer = optimizer_adam(),
    loss = 'binary_crossentropy',
    metrics = 'accuracy'
  )

learn <- model %>% fit((scale.trainX.mat), trainY,
                       epochs = 300,
                       batch_size = 64,
                       validation_split = 0.2,
                       verbose = FALSE,
                       callbacks = list(callback_early_stopping(patience = 100)))
learn
```

```
Trained on 480 samples, validated on 120 samples
(batch_size=64, epochs=268)
Final epoch (plot to see history):
     acc: 0.9792
    loss: 0.06945
 val_acc: 0.95
val_loss: 0.1449
```

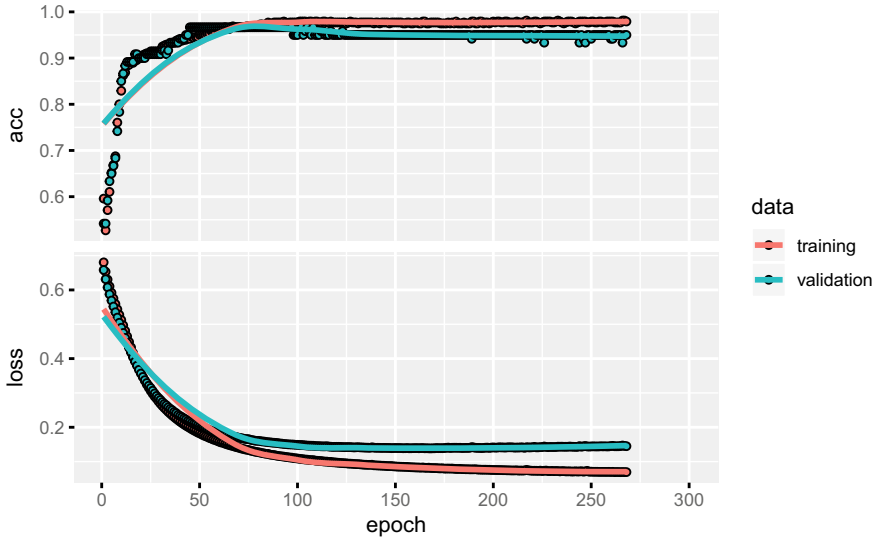

Fig. 6.5 Plot of accuracy and loss with respect to epochs using a keras model with one hidden layer having 175 nodes with sigmoid activation in the output layer and, adjusting the epoch to stop training the model if the accuracy does not improve after 100 epochs

```
plot(learn)
```

The loss and accuracy with a single layer `keras` model, with early stopping is plotted in Fig. 6.5.

6.2.2 Add Batch Normalization

```
set.seed(1)
model <- keras_model_sequential() %>%
  layer_dense(units = 175, activation = "relu", input_shape = 2) %>%
  layer_batch_normalization() %>%
  layer_dense(units = 1, activation = 'sigmoid') %>%
  compile(
    optimizer = optimizer_adam(),
    loss = 'binary_crossentropy',
    metrics = 'accuracy'
  )

learn <- model %>% fit((scale.trainX.mat), trainY,
                       epochs = 300,
                       batch_size = 64,
                       validation_split = 0.2,
                       verbose = FALSE,
                       callbacks = list(callback_early_stopping(patience = 100)))
learn
```

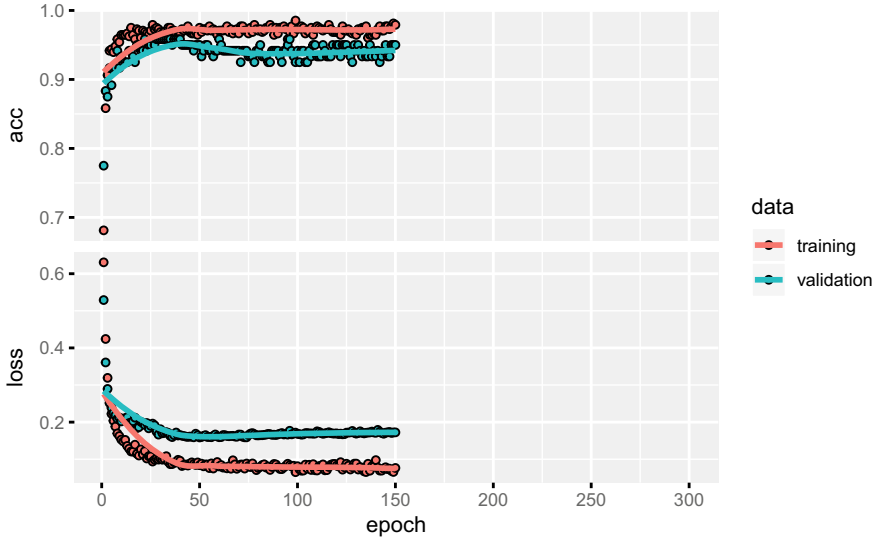

Fig. 6.6 Plot of accuracy and loss with respect to epochs using a keras model with one hidden layer having 175 nodes with sigmoid activation in the output layer and, adjusting the epoch to stop training the model if the accuracy does not improve after 100 epochs and, using batch normalisation

```
Trained on 480 samples, validated on 120 samples (batch_size=64, epochs=150)
Final epoch (plot to see history):
     acc: 0.9792
    loss: 0.07647
 val_acc: 0.95
val_loss: 0.1723
```

```
plot(learn)
```

The loss and accuracy with a single layer `keras` model, with early stopping and batch normalization is plotted in Fig. 6.6.

6.2.3 Add Dropout

```
set.seed(1)
model <- keras_model_sequential() %>%
  layer_dense(units = 175, activation = "relu", input_shape = 2) %>%
  layer_batch_normalization() %>%
  layer_dropout(rate = 0.2) %>%
  layer_dense(units = 1, activation = 'sigmoid') %>%
  compile(
    optimizer = optimizer_adam(),
    loss = 'binary_crossentropy',
    metrics = 'accuracy'
  )
```

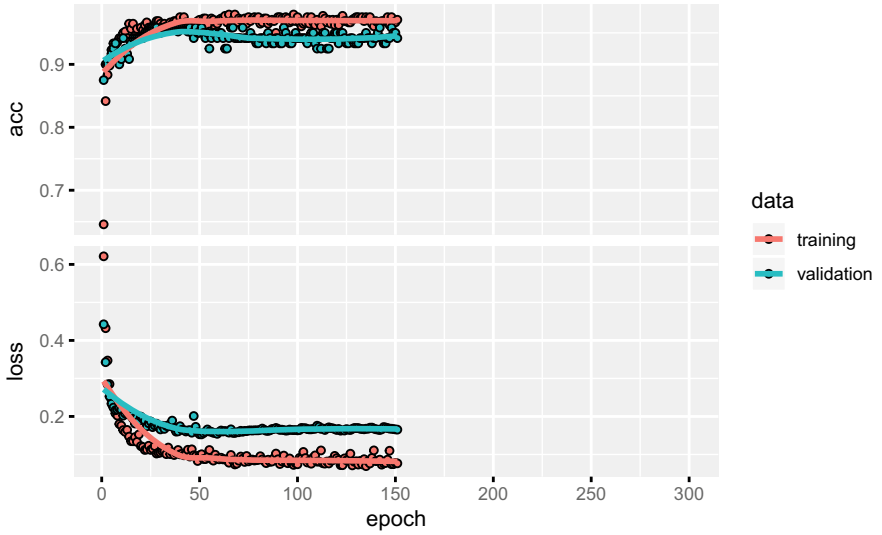

Fig. 6.7 Plot of accuracy and loss with respect to epochs using a keras model with one hidden layer having 175 nodes with sigmoid activation in the output layer and, adjusting the epoch to stop training the model if the accuracy does not improve after 100 epochs and, using batch normalisation and, with a dropout regularization of 0.2

```
learn <- model %>% fit((scale.trainX.mat), trainY,
                       epochs = 300,
                       batch_size = 64,
                       validation_split = 0.2,
                       verbose = FALSE,
                       callbacks = list(callback_early_stopping(patience = 100)))
learn
```

```
Trained on 480 samples, validated on 120 samples (batch_size=64, epochs=151)
Final epoch (plot to see history):
     acc: 0.9708
    loss: 0.07657
 val_acc: 0.9417
val_loss: 0.1653
```

```
plot(learn)
```

The loss and accuracy with a single layer keras model, with early stopping, batch normalization and dropout regularization is plotted in Fig. 6.7.

6.2.4 Add Weight Regularization

```
set.seed(1)
model <- keras_model_sequential() %>%
  layer_dense(units = 175, activation = "relu",
```

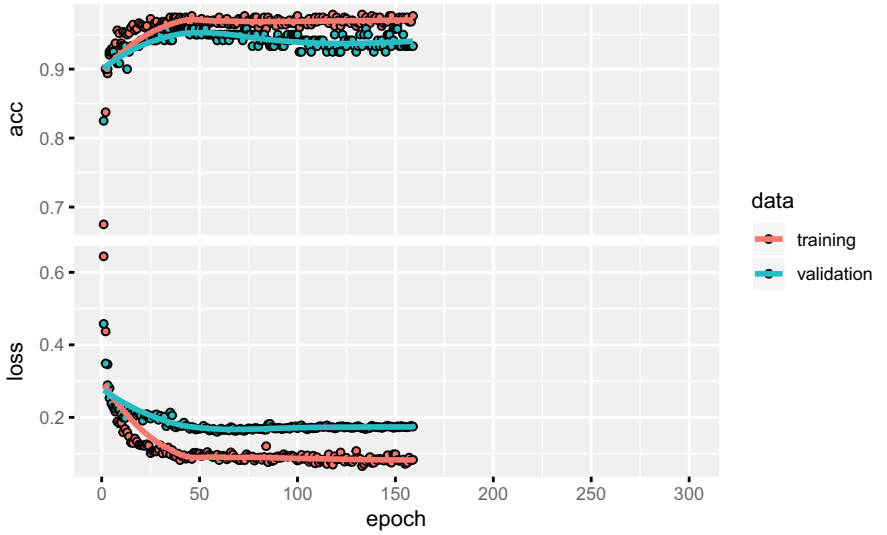

Fig. 6.8 Plot of accuracy and loss with respect to epochs using a keras model with one hidden layer having 175 nodes with sigmoid activation in the output layer and, adjusting the epoch to stop training the model if the accuracy does not improve after 100 epochs and, using batch normalisation and, with a dropout regularization of 0.2 and, adding parameter weight regularization of 0.001

```
              input_shape = 2,
              kernel_regularizer = regularizer_l2(0.001)) %>%
  layer_batch_normalization() %>%
  layer_dropout(rate = 0.2) %>%
  layer_dense(units = 1, activation = 'sigmoid') %>%
  compile(
    optimizer = optimizer_adam(),
    loss = 'binary_crossentropy',
    metrics = 'accuracy'
  )
learn <- model %>% fit((scale.trainX.mat), trainY,
                       epochs = 300,
                       batch_size = 64,
                       validation_split = 0.2,
                       verbose = FALSE,
                       callbacks = list(callback_early_stopping(patience = 100)))
learn
```

```
Trained on 480 samples, validated on 120 samples (batch_size=64, epochs=159)
Final epoch (plot to see history):
     acc: 0.9771
    loss: 0.08318
 val_acc: 0.9333
val_loss: 0.1745
```

```
plot(learn)
```

The loss and accuracy with a single layer `keras` model, with early stopping, batch normalization, dropout regularization and parameter weight regularization is plotted in Fig. 6.8.

6.2.5 Adjust Learning Rate

```
set.seed(1)
model <- keras_model_sequential() %>%
  layer_dense(units = 175,
              activation = "relu", input_shape = 2,
              kernel_regularizer = regularizer_l2(0.001)) %>%
  layer_batch_normalization() %>%
  layer_dropout(rate = 0.2) %>%
  layer_dense(units = 1, activation = 'sigmoid') %>%
  compile(
    optimizer = optimizer_adam(),
    loss = 'binary_crossentropy',
    metrics = 'accuracy'
  )

learn <- model %>% fit((scale.trainX.mat), trainY,
                       epochs = 300,
                       batch_size = 64,
                       validation_split = 0.2,
                       verbose = FALSE,
                       callbacks = list(callback_early_stopping(patience = 100),
                                        callback_reduce_lr_on_plateau()))
learn
```

```
Trained on 480 samples, validated on 120 samples (batch_size=64, epochs=158)
Final epoch (plot to see history):
     acc: 0.9729
    loss: 0.08223
 val_acc: 0.9417
val_loss: 0.1585
      lr: 0.000000000001
```

```
plot(learn)
```

6.2.6 Prediction

```
model %>% predict(scale.testX.mat[1:10, ])
```

```
              [,1]
 [1,] 0.0144612212
 [2,] 0.0084884493
 [3,] 0.0061556785
 [4,] 0.0049234540
 [5,] 0.0037987749
 [6,] 0.0037381186
 [7,] 0.0008490850
 [8,] 0.0002616270
 [9,] 0.0001423903
[10,] 0.0002294233
```

```
predictions <- model %>% evaluate(scale.testX.mat, testY)
```

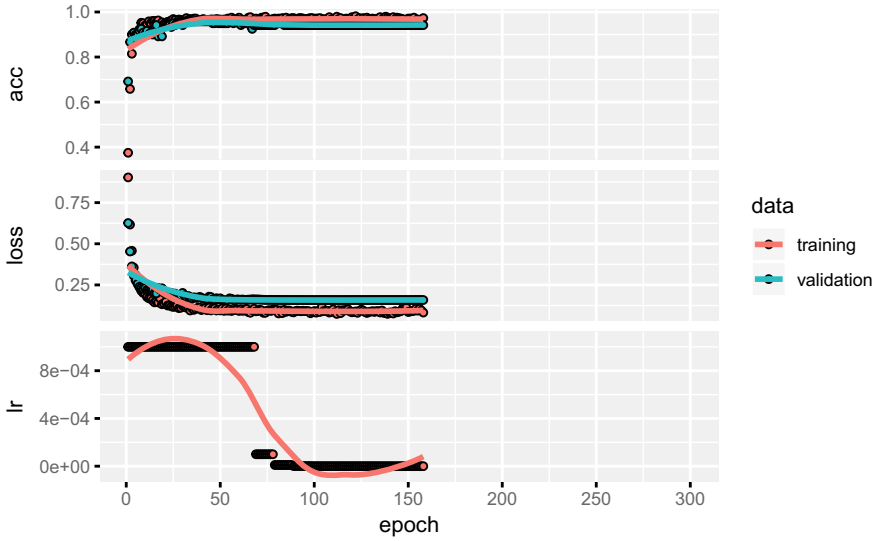

Fig. 6.9 Plot of accuracy and loss with respect to epochs using a keras model with one hidden layer having 175 nodes with sigmoid activation in the output layer and, adjusting the epoch to stop training the model if the accuracy does not improve after 100 epochs and, using batch normalisation and, with a dropout regularization of 0.2 and, adding parameter weight regularization of 0.001 and, adjusting the learning rate

The loss and accuracy with a single layer `keras` model, with early stopping, batch normalization, dropout regularization, parameter weight regularization and learning rate adjustment is plotted in Fig. 6.9.

6.3 Introduction to `TensorFlow`

`TensorFlow` is a general purpose, open source,numerical computing library. It is a hardware independent, lower level mathematical library for building deep neural network architectures. The `keras R` package makes it easy to use Keras and `TensorFlow` in `R`.

`TensorFlow` was developed by the Google Brain Team at Google's, Machine Intelligence research organization. `TensorFlow` is an open-source software library for numerical computation using data flow graphs. Nodes in the graph represent mathematical operations, while the graph edges represent the multidimensional data arrays or **tensors**, which communicate between them. The `TensorFlow` architecture allows us to deploy computation to one or more CPUs or GPUs in a desktop, server, or mobile device with a single API.

Like `Keras`, `TensorFlow` is also a Machine Learning framework that has the ability to speed up network development, significantly and is mainly designed for DNN models.

Table 6.1 Tensor arrays (36 is an arbitrary number used here)

Dimension	R Object
0D	36 a scalar
1D	c(36, 36, 36) a vector
2D	matrix(32, nrow = 2, ncol = 2)
3D	array(36, dim = c(2, 3, 2)
4D	array(36, dim = c(2,2,2,2)

6.3.1 What is `Tensor` 'Flow'?

A `TensorFlow` is a data flow graph consisting of nodes representing computational units. The flow of tensors consists of the following:

- Defining the graph.
- Compiling the graph.
- Executing the graph.

Nodes are the representations and tensors flow between the nodes.

The `R` interface to `TensorFlow` lets us use the `Keras` and `Estimator` APIs and also provides full access to the core `TensorFlow` API- `Keras` API, `Estimator` API and `Core` API. `TensorFlow` does not require all the data to be in the RAM.

`Tensors` are a generalization of vectors and matrices with any arbitrary dimension (also called *axis*). We use vectors to operate on 1D tensors and use matrices to operate on 2D `tensors`. `array` objects use higher dimensions. `Tensors` are therefore multidimensional arrays and the data is stored in these arrays.

A `tensor` containing a single number is called a `scalar` or zero-dimensional `tensor` (0D). A 1D `tensor` is a `vector` (Tables 6.1 and 6.2).

> An *n*D vector and an *n*D `tensor` are not the same. An *n*D vector has only one axis and an *n*D `tensor` has *n* axes.

`TensorFlow` has several units, namely (1) High-Level APIs, (2) Estimators, (3) Estimators, (4) Accelerators, (5) Low-Level APIs, (6) ML Concepts, (7) Debugging, (8) TensorBoard, and (9) Misc.

The High Level APIs include

- Keras,
- Eager Execution,
- Importing Data,
- Estimators.

Table 6.2 Each row is an observation and samples is always the first argument

Data	Tensor
Scalar	1D tensor
Time Series or sequence data	2D tensors of shape (samples, features)
Vector data	3D tensors of shape(samples, timesteps, features)
Images	4D tensors of shape (samples, channels (RGB), height, width)
Video	5D tensors of shape (samples, frames, channels, height, width)

6.3.2 `Keras`

`Keras` is a high-level API to build and train deep learning models. It is used for fast prototyping, advanced research, and production. It is user-friendly, is modular (able to connect configurable building blocks) and it is easy to write customized building blocks.

In this section, we will explore the following in TensorFlow:

- Initialize variables.
- Start a session.
- Train algorithms.
- Implement a Neural Network using the `MNIST` data.

Writing and running programs in `TensorFlow` has the following steps:

- Create Tensors (variables) that are not yet executed/evaluated.
- Write operations between those Tensors.
- Initialize your Tensors.
- Create a Session.
- Run the Session.

When we create a variable for the loss, we simply define the loss as a function of other quantities, but do not evaluate its value.

6.3.3 *Installing and Running `TensorFlow`*

```
library(tensorflow)
# install_tensorflow()
```

6.4 Modeling Using TensorFlow

Before we carry out any computation, we need to start an interactive session.

```
sess = tf$InteractiveSession()
a = tf$constant(2)
b = tf$constant(10)
c = tf$multiply(a, b)
print(sess$run(c))
```

```
[1] 20
```

6.4.1 Importing MNIST Data Set from TensorFlow

```
datasets <- tf$contrib$learn$datasets
mnist <- datasets$mnist$read_data_sets("MNIST-data", one_hot = TRUE)

train_images <- mnist$train$images
train_labels <- mnist$train$labels

label_1 <- train_labels[1, ]
image_1 <- train_images[1, ]

label_1
```

```
 [1] 0 0 0 0 0 0 0 1 0 0
```

```
length(image_1)
```

```
[1] 784
```

```
image_1[250:300]
```

```
 [1] 0.00000000 0.00000000 0.00000000 0.00000000 0.00000000 0.54901963
 [7] 0.98431379 0.99607849 0.99607849 0.99607849 0.99607849 0.99607849
[13] 0.99607849 0.99607849 0.99607849 0.99607849 0.99607849 0.99607849
[19] 0.99607849 0.99607849 0.99607849 0.74117649 0.09019608 0.00000000
[25] 0.00000000 0.00000000 0.00000000 0.00000000 0.00000000 0.00000000
[31] 0.00000000 0.00000000 0.00000000 0.88627458 0.99607849 0.81568635
[37] 0.78039223 0.78039223 0.78039223 0.78039223 0.54509807 0.23921570
[43] 0.23921570 0.23921570 0.23921570 0.23921570 0.50196081 0.87058830
[49] 0.99607849 0.99607849 0.74117649
```

Let us identify the different labels from the MNIST data.

```
grayscale <- colorRampPalette(c("white", "blue"))
par(mar = c(1, 1, 1, 1), mfrow = c(8, 8), pty = "s", xaxt = "n",
    yaxt = "n")

for (i in 1:40) {
    z <- array(train_images[i, ], dim = c(28, 28))
```

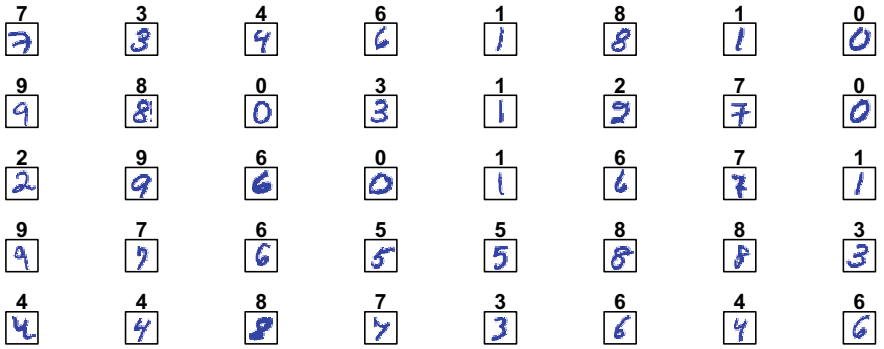

Fig. 6.10 Identification of different labels from the MNIST data set. Each label is 28 pixels by 28 pixels

```
    z <- z[, 28:1]
    image(1:28, 1:28, z, main = which.max(train_labels[i, ]) -
        1, col = grayscale(256), xlab = "", ylab = "")
}
```

The labels from the MNIST data is shown in Fig. 6.10.

6.4.2 *Define* `Placeholders`

Instead of running a single expensive operation independently with `R`, `TensorFlow` lets us describe a graph of interacting operations that run entirely outside `R`. To access the TensorFlow API, we reference the `tf` object exported by the `tensorflow` package.

In the following code, x and y is just a placeholder and does carry hold any value. We will assign a value to the placeholders when we run a computation. Specifically, we would want to input any number of the *MNIST* images, each flattened into a 784-dimensional vector. We represent this as a 2-dimensional tensor of floating-point numbers, with shape $(NULL, 784L)$, where $NULL$ implies a dimension of any length.

We also need the weights and biases for our model. In TensorFlow, `Variable` is a modifiable `tensor` in `TensorFlow` graph. It can be used and modified during computation. We create these variables by giving `tf$Variable` an initial value. In our case, we initialize both W and b as `tensors` with zeros.

In this case, W has shape $(784, 10)$ because we want to multiply the 784-dimensional image vectors by W to produce 10-dimensional vectors for the different classes. b has shape (10).

We thereafter implement our model with the function `tf$nn$softmax (tf$matmul(x, W) + b)` or the alternative `tfnnsoftmax(tf $add(tf$multiply(X, W), b))`.

```
# Declare placeholders (images are 55000 x 784 and labels are 55000 x 10)
x <- tf$placeholder(tf$float32, shape(NULL, 784L))

# Initialize the weight matrix (784 x 10)
W <- tf$Variable(tf$zeros(shape(784L, 10L)))
# Initialize the weight matrix (10 x 1)
b <- tf$Variable(tf$zeros(shape(10L)))

# We will be using a softmax output layer to predict y
y <- tf$nn$softmax(tf$matmul(x, W) + b)
```

6.4.3 *Training the Model*

We now need to define our *CE* loss function, which is $-\sum y' log(y)$, where *yhat* is the predicted probability distribution and y is the actual probability distribution. If you recall from Sect. 2.2, *CE* is how different is the the predicted probability distribution from the true distribution.

To implement *CE*, we need to first add a new placeholder to input the correct labels and then define the *CE* loss function. `tf$log` computes the log of each label y. We then multiply each element of *yhat* with the corresponding element of *tf*$*log*$(y)$. The `tf$reduce_sum` function adds the elements in the second dimension of y, as defined in the `reduction_indices=1L` parameter. Finally, `tf$reduce_mean` computes the mean over all examples in the batch. Note that `tensor` indices are 0-based in `TensorFlow` API, rather than 1-based in `R` vectors.

Having defined the complete model, `TensorFlow` now knows the entire graph of our computations and it can automatically implement the backpropagation algorithm. Then, it can apply our choice of optimization algorithm to modify the variables and reduce the loss.

```
# Input the correct labels
yhat <- tf$placeholder(tf$float32, shape(NULL, 10L))
# Define the CE loss function
cross_entropy <- tf$reduce_mean(-tf$reduce_sum(yhat * tf$log(y),
                                               reduction_indices = 1L))
# Minimize CE using gradient descent with a learning rate of 0.5.
optimizer <- tf$train$GradientDescentOptimizer(0.5)
train_step <- optimizer$minimize(cross_entropy)
```

We first create a `Session` and launch the model and initialize the variables. After this step, at each step of the `for` loop, we get a mini_batch of 100 random data points from the training set. We then run `train_step`, feeding in the mini_batches to replace the placeholders x and y.

6.4.4 *Instantiating a* `Session` *and Running the Model*

```
# Create a session before any computation.
sess = tf$InteractiveSession()

sess$run(tf$global_variables_initializer())
```

```
for (i in 1:1000) {
    batches <- mnist$train$next_batch(100L)
    batch_x <- batches[[1]]
    batch_y <- batches[[2]]
    sess$run(train_step, feed_dict = dict(x = batch_x, yhat = batch_y))
}
```

6.4.5 *Model Evaluation*

We would now want to evaluate our model. Let us first find out where we predicted the correct label. The `tf$argmax` function gives us the index of the highest entry in a `tensor` along some axis. For example, `tf$argmax(y, 1L)` is the label our model thinks is most likely for each input, while `tf$argmax(y_, 1L)` is the correct label. We use `tf$equal` to check if our prediction matches the truth.

The function `correct_prediction` returns a vector of booleans. To determine what fraction is correct, we cast it to floating-point numbers and then take the mean. For example, $(TRUE, FALSE, TRUE, TRUE, TRUE)$ would become (1.0, 0.0, 1.0, 1.0, 1.0) which becomes 0.80.

Also please note that since `tensors` in the TensorFlow API start from 0, we pass $1L$ to specify that `tf$argmax` should operate on the second dimension of the `tensor`.

Finally, seek the accuracy on our test data.

```
correct_prediction <- tf$equal(tf$argmax(y, 1L), tf$argmax(yhat, 1L))
accuracy <- tf$reduce_mean(tf$cast(correct_prediction, tf$float32))

# training accuracy
sess$run(accuracy, feed_dict=dict(x = mnist$train$images,
                                  yhat = mnist$train$labels))
```

```
[1] 0.9174727
```

```
# test accuracy
sess$run(accuracy, feed_dict=dict(x = mnist$test$images,
                                  yhat = mnist$test$labels))
```

```
[1] 0.92
```

The model returns an accuracy of 92%. Using ConvNets, we can get an accuracy of 97%.

6.5 Conclusion

We have learnt how to tweak some of the hyperparameters while working with deep layered neural networks. We have discussed the `TensorFlow` framework and successfully applied it to learn a deep neural network.

In the next chapter, we will discuss a new type of architecture for neural networks called convolutional neural networks.

Chapter 7
Convolutional Neural Networks (ConvNets)

The pooling operation used in convolutional neural networks is a big mistake, and the fact that it works so well is a disaster.
`Geoffrey Hinton`

Abstract In this chapter, we will discuss and understand the building blocks of a Convolutional Neural Network (ConvNet). In particular, we will learn

- How to build a convolutional neural network.
- How to apply convolutional neural networks on image data.
- How to apply convolutional neural networks for image recognition.
- Introduce the reader to neural style transfer to generate art.

7.1 Building Blocks of a Convolution Operation

7.1.1 What is a Convolution Operation?

Convolutional Networks work with something which is called **Sparse Connectivity**.

A Convolutional Neural Network (ConvNet or CNNs) exploit the *spatially local* correlation by enforcing a local connectivity between neurons of adjacent layers. This implies that the inputs of hidden units in layer ℓ belong to a subset of units in layer ℓ-1 which have "spatially contiguous" receptive fields.

Each unit does not respond to the variations which are outer to its receptive field with respect to the input. This ensures that the learnt "filters" result in the strongest response to a spatially local input pattern.

The convolution operation is the fundamental building block of a convolutional neural network (CNN or ConvNet).

Figure 7.1 and 7.2 shows a typical convolution operation, where a $5 \times 3 \times 3$ `filter` slides over a $32 \times 32 \times 3$ image and, along the way it computes the dot product between the `filter` (weights) and the chunks of the input image. As there

© Springer Nature Singapore Pte Ltd. 2019

A. Ghatak, *Deep Learning with R*,
https://doi.org/10.1007/978-981-13-5850-0_7

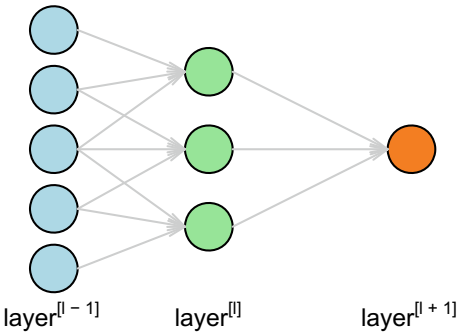

Fig. 7.1 The inputs of the hidden units in layer ℓ are from a subset of units in layer $\ell - 1$ that have spatially contiguous receptive fields

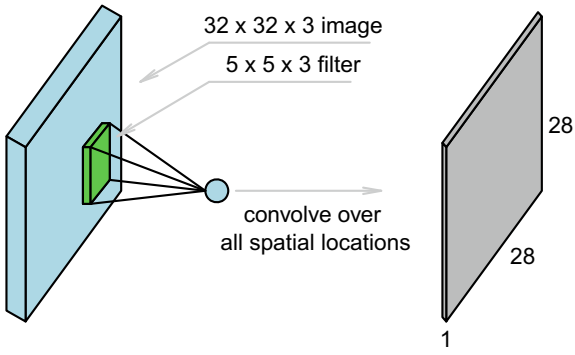

Fig. 7.2 We take a $5 \times 5 \times 3$ `filter` and slide it over the complete image and at each step, take the dot product between the `filter` and the respective chunks of the input image over one layer and, get an output dimension of $28 \times 28 \times 1$

are 28×28 unique positions where the `filter` can be put on the image, we get a $28 \times 28 \times 1$ dimensional output, for one layer of the image.

If you are facing difficulty in calculating the output dimension, do not bother; this will become clear as we move along to the next sections.

Let us try to understand this with an example. We will consider a 6×6 matrix and apply a 3×3 `filter` (also termed as *kernel*), which is a hyperparameter, over the matrix and interpret the results. We can think of the `filter` as an arrangement of weights.

1	3	5	2	6	1
7	0	2	4	1	3
5	1	3	6	2	7
2	4	3	0	0	1
1	7	3	9	2	1
3	6	2	4	1	0

*

1	0	−1
1	0	−1
1	0	−1

=

3	−8	1	1
6	−5	5	−1
−1	−3	5	6
−2	4	5	11

In the above figure, when we apply the `filter` to the first row and first column of the matrix, we multiply the parameters of the `filter` with the input and we get-

$$1 \times 1 + 7 \times 1 + 5 \times 1 + 3 \times 0 + 0 \times 0 + 1 \times 0 + 5 \times -1 + 2 \times -1 + 3 \times -1 = 3$$

The first row first column entry is 3, in the 4×4 output matrix. Similarly, when we scan across all the 4×4 sections of the 6×6 matrix with our `filter` we get the remaining values in the output matrix. We can think of the 4×4 output matrix as an image matrix.

Applying this `filter` on the input data is the **convolution operation**.

The convolution operation is also a very convenient way to specify edges in an image. Let us look at an example to detect edges using convolution. We will consider a $6 \times 6 \times 1$ grayscale image with a single channel (rgb images will have 3 channels images and, grayscale will have images have 1 channel).

7.1.2 Edge Detection

When we convolve the grayscale image with a 3×3 `filter` to get a 4×4 image matrix, we get a vertical edge detector as shown below (Fig. 7.3).

The image of the output matrix is plotted in Figure 7.4. The lighter regions in the middle are book-ended by the darker regions, with vertical edges down the middle. Since we are dealing with a 6×6 image matrix, the detected edge appears very thick, and that would not be the case if we had a large image matrix. The convolution operation still does a pretty good job of detecting the vertical edges of our image.

Let us try to detect horizontal edges with a horizontal edge detection filter as shown in Fig. 7.5-

There are different filters which are used in ConvNets and two of these are- *Sobel* `filter` and the *Scharr* `filter` depicted as

$$\text{Sobel filter} = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & 2 \\ 1 & 0 & -1 \end{bmatrix} \tag{7.1.1}$$

$$\text{Scharr filter} = \begin{bmatrix} 3 & 0 & -3 \\ 10 & 0 & -10 \\ 3 & 0 & -3 \end{bmatrix} \tag{7.1.2}$$

Different filters allow us to detect vertical (Fig. 7.6) and horizontal (Fig. 7.5) edges of an image in different ways. They are also used to detect edges at different angles.

Fig. 7.3 Edge detection using a vertical edge detection filter

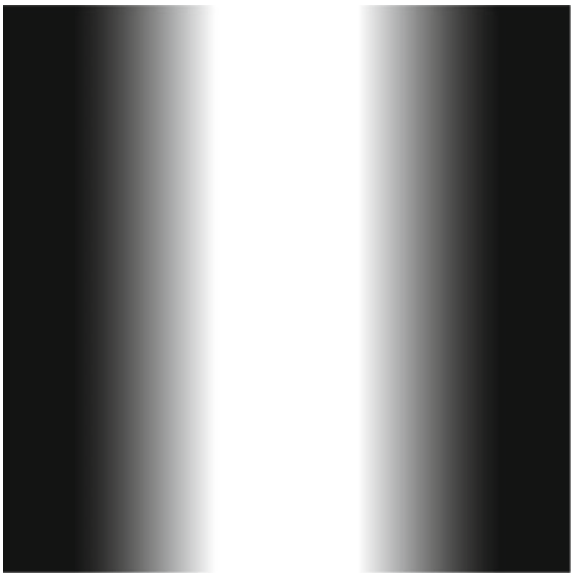

Fig. 7.4 The vertical edges of our $6 \times 6 \times 1$ image is detected by applying the convolution operation

20	20	20	0	0	0
20	20	20	0	0	0
20	20	20	0	0	0
0	0	0	20	20	20
0	0	0	20	20	20
0	0	0	20	20	20

*

1	1	1
0	0	0
−1	−1	−1

=

0	0	0	0
60	20	−20	−60
60	20	−20	−60
0	0	0	0

Fig. 7.5 Edge detection using a horizontal edge detection filter

If we have an $n \times n$ image matrix and an $f \times f$ `filter`, the general formula for the dimension of the output image matrix is $n - f + 1 \times n - f + 1$, but this will change as we will see in the next sections. By convention in computer vision, `f` is usually odd, because in an odd dimension filter will always have a central position and sometimes in computer vision its nice to have a central pixel, which we can refer to as the position of the filter. Another reason why `f` is generally odd is because of `padding`.

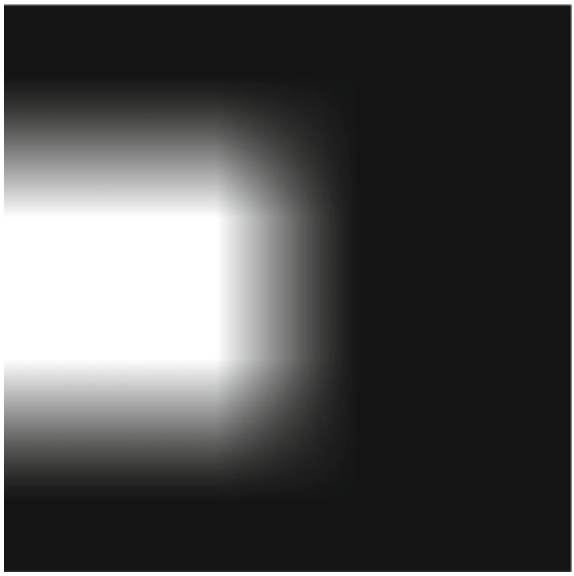

Fig. 7.6 The horizontal edges of our $6 \times 6 \times 1$ image is detected by applying the convolution operation

Technically this operation is sometimes called *cross-correlation* instead of convolution, but in deep learning literature by convention, we call this a convolutional operation.

We now have a fair understanding of what is a convolution operation and how it works. We will now look at some other building blocks of ConvNets.

7.1.3 Padding

In the above 6×6 image matrices and with a 3×3 `filter`, there are only 4×4 possible positions. This poses two problems first, we have a shrinking output and second, we are literally ignoring the information present in the edges of the image matrix. When we are dealing with really deep neural networks, we may not want the image to shrink on every step because at the final layer, we might end up with a very small image. We also may not want to lose information carried in the input data. We, therefore, introduce a hyperparameter called `padding`, represented by the letter `p`.

To apply padding to an image implies adding an additional border of one pixel all around the edges. When we apply a `padding p` $= 1$ on a 6×6 image, we are getting a 8×8 image matrix as shown in Figure 7.7. Now when we convolve this with a 3×3 `filter` we get back a 6×6 image matrix in the output. By convention, we apply `padding` with zeros.

0	0	0	0	0	0	0	0
0	1	3	5	2	6	1	0
0	7	0	2	4	1	3	0
0	5	1	3	6	2	7	0
0	2	4	3	0	0	1	0
0	1	7	3	9	2	1	0
0	3	6	2	4	1	0	0
0	0	0	0	0	0	0	0

Fig. 7.7 `padding` applied to an image matrix with one pixel. Notice the zeros in the outer rows and columns

If we apply a $\texttt{p} = 2$, i.e., apply two pixels of padding our 6×6 image becomes a 10×10 matrix and after convolving with a 3×3 `filter` we get a 7×7 image matrix. The general formula for the dimensions of the output matrix is

$$(n + 2p - f + 1) \times (n + 2p - f + 1)$$

By convention, in ConvNet parlance, there two names by which we address how much to pad

- Valid: No padding
- Same: Pad as much so that the output is the same size as the input

There is one last thing about `padding` and that is, if we have odd numbered `f` we will have asymmetric padding.

7.1.4 Strided Convolutions

Another building block of ConvNets is the stride. By stride we mean, how many steps do we take to "jump" over a block. This is also a hyperparameter and is represented by `s`.

Suppose we would like to convolve the following 9×9 image matrix by a 3×3 `filter`. We can hop over from one row / column to the other by a step of one ($\texttt{s} = 1$). In this example, after the convolution operation with $\texttt{s} = 1$, we get a 6×6 output.

1	3	5	2	6	1	5	6	2
7	0	2	4	1	3	2	5	1
5	1	3	6	2	7	4	3	8
2	4	3	0	0	1	9	1	4
1	7	3	9	2	1	3	9	1
3	6	2	4	1	0	1	3	5
2	5	7	0	4	8	1	2	7
1	0	5	2	5	1	7	9	2
6	3	7	9	2	5	3	4	1

We can also hop over from one row / column to the other by a step of two ($\mathtt{s} = 2$) and, after the convolution operation, we get a 3×3 output.
The dimensions of the output matrix with stride is

1	3	5	2	6	1	5	6	2
7	0	2	4	1	3	2	1	
5	1	3	6	2	7	4	3	8
2	4	3	0	0	1	9	1	4
1	7	3	9	2	1	3	9	1
3	6	2	4	1	0	1	3	5
2	5	7	0	4	8	1	2	7
1	0	5	2	5	1	7	9	2
6	3	7	9	2	5	3	4	1

$$\left(\frac{n+2p-f}{s}+1, \frac{n+2p-f}{s}+1\right)$$

If the fraction is not an integer, we use the `floor` function in `R`.

7.1.5 Convolutions over Volume

Applying convolutions to $6 \times 6 \times 3$ (3D) images involves dealing with color "channels" (Fig. 7.8), which is a stack of three rgb channels corresponding to red, green, and blue. To detect edges and other features in this image, we need a $3 \times 3 \times 3$ `filter`,
(a 3D `filter`) i.e., the number of channels in the `filter` matches the number of channels in the image.

The image is normally defined by its height, width, and the number of channels and so does the `filter`, which also has a height, a width, and a number of channels.

The output, however, in this case will be a $4 \times 4 \times 1$- dimensional image.

To compute the output of this convolutional operation, we will place the `filter` on the upper left most position of the image. Our `filter` now has $3 \times 3 \times 3 = 27$ parameters, and we take each of these parameters and carry out a *dotproduct* with the corresponding numbers from the red, green, and blue channels of the image (Fig. 7.9).

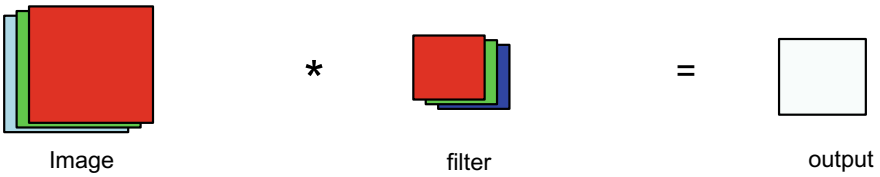

Fig. 7.8 Convoluting a $6 \times 6 \times 3$ image matrix with rgb channels

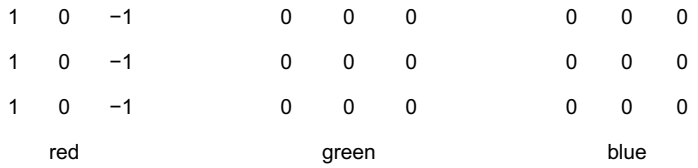

Fig. 7.9 To detect vertical edges in the red channel we need to set the green and blue channel in the `filter` to zero

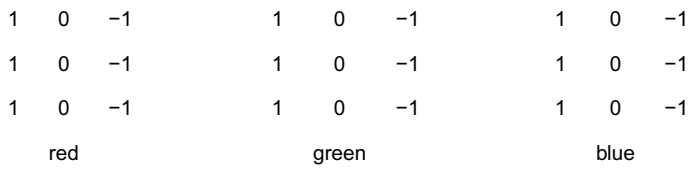

Fig. 7.10 To detect vertical edges irrespective of color we need to have the same values in each of the channels of the `filter`

We do this by taking the first nine numbers from the red channel of the `filter`, then the green channel and, the blue channel, and multiply it with the corresponding nine numbers from each of the individual three channels of the image. Adding up the *dotproduct* of all these numbers gives us the first number in the output. We then slide over the `filter` by one step and repeat the same procedure till we get 16 values of the output.

Now if we want to detect vertical edges (we can also detect horizontal or slanting edges, in which case we have to change the values in the filter and also consider using multiple filters) in the red channel of the image, then we need to set the green and blue filters to zero. Or if we do not care about the color of the edges, we can set the filter as is shown in Figure 7.10.

We can also consider using multiple filters (Fig. 7.11)

With different choices of the parameters in our `filter`, we can extract different types of features including horizontal and other angular edges.

There is another interesting exercise we can consider—using multiple filters. We can use the first `filter` as a vertical edge detector and the second `filter` as an horizontal edge detector. So, now we have two output matrices stacked together—the first output matrix defining the vertical edges and the second output layer defining the horizontal edges. We have the liberty and acumen to keep adding different `filters` to gather different types of edges.

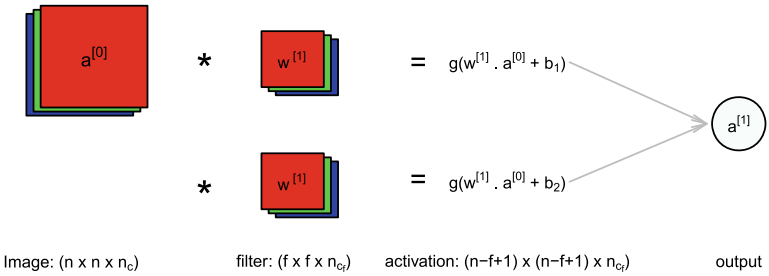

Fig. 7.11 Using multiple filters on the input to get a 2-D output

7.1.6 Pooling

The pooling (POOL) layer reduces the dimensions of the input and also therefore, reduces computation and make feature detectors more invariant (invariance implies that we can recognize an object even when its appearance varies) to its position in the input.

**Translation invariance* means that the system produces exactly the same response, regardless of how the input varies. For example, a dog-detector might report "dog-identified" for all input images.

The two types of pooling layers are

- Max-pooling layer: slides the filter over the input and stores the maximum of the values of the input (overlapping the the input) in the output.
- Average-pooling layer: slides the filter over the input and stores the average value of the input (overlapping the the input) in the output.

Pooling layers have no parameters for backpropagation to train. However, they have hyperparameters such as the window size `f`. This specifies the height and width of the $f \times f$ window to compute a maximum or average over.

Normally, max-pooling is used more than average-pooling. However, while working on a deep neural network, if we would want to collapse our output, we would like to use average-pooling.

Another important thing to note is that when we use max-pooling, we usually do not use `padding` and therefore the dimensions of the output is $\frac{n_H - f}{s} + 1 \times \frac{n_W - f}{s} + 1 \times n_C$

The hyperparameters for pooling are the following:

- filter size `f`
- stride `s`
- Max Pool or Average Pool.

Pooling is visually depicted below.

7.2 Single-Layer Convolutional Network

A simple convolutional network is depicted in Figs. 7.12 and 7.13 If we denote the height and width of the input matrix as n_h and n_w and, if we consider a layer ℓ which is a convolutional layer we can state

$$
\begin{aligned}
Input &= n_h^{[\ell-1]} \times n_w^{[\ell-1]} \times n_c^{[\ell-1]} \\
\text{number of filters} &= n_{c_f}^{[\ell]} \\
filter\ size &= f^{[\ell]} \\
stride &= s^{[\ell]} \\
padding &= p^{[\ell]} \begin{cases} = 0 \text{ if Valid} \\ \neq 0 \text{ if Same (to ensure same size as the input)} \end{cases} \\
Weights &= f^{[\ell]} \times f^{[\ell]} \times n_c^{[\ell-1]} \times n_{c_f}^{[\ell]} \\
bias &= n_{c_f}^{[\ell]} \\
n_h^{[\ell]} \text{ and } n_w^{[\ell]} &= [\frac{n^{[\ell-1]} + 2p^{[\ell]} - f^{[\ell]}}{s^{[\ell]}} + 1] \\
activations\ a^{[\ell]} &= n_h^{[\ell]} \times n_w^{[\ell]} \times n_c^{[\ell]} \\
Activations\ A^{[\ell]} &= m \times n_h^{[\ell]} \times n_w^{[\ell]} \times n_c^{[\ell]} \\
Output &= n_h^{[\ell]} \times n_w^{[\ell]} \times n_c^{[\ell]} \\
where,& \\
& n_c \text{ is the number of channels in the input,} \\
& n_{c_f}^{[\ell]} \text{ is number of filters in layer } \ell, \\
& m \text{ is the number of examples.}
\end{aligned}
$$

Before we proceed further, let us clear the conundrum of dimensions with an example—suppose we have a $64 \times 64 \times 20$ input volume, which we want to convolve with 50 filters having dimensions 9×9 each, and we are using a stride of 2 with no padding, the dimensions of the output volume is calculated with the formula

$$
\begin{aligned}
dim &= floor\left(\frac{(n + 2p - f)}{s}\right) + 1 \text{ (use floor if it is a fraction or else not)} \\
n &= 64 \\
p &= 0 \\
s &= 2 \\
f &= 9 \\
n_{c_f} &= 50 \\
floor\left(\frac{(64 + 0 - 9)}{2}\right) + 1 &= 28
\end{aligned}
$$

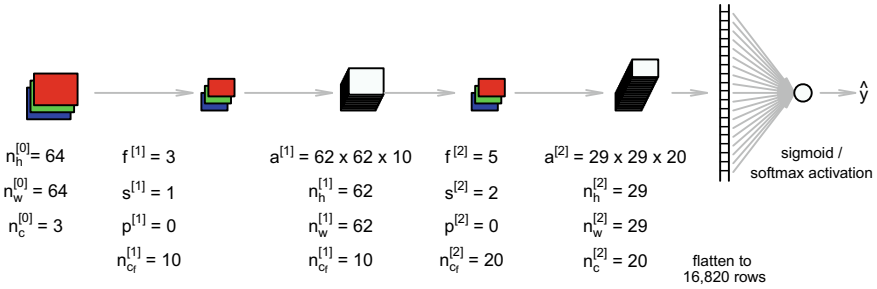

Fig. 7.12 A single-layer Convolution Network diagram

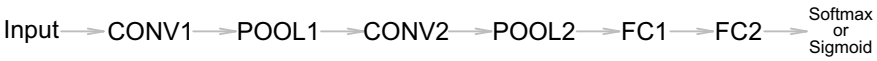

Fig. 7.13 A complete Convolution Neural Network diagram

The dimensions of the output volume is $28 \times 28 \times 50$.

Having defined the above, let us work through a single-layer ConvNet

7.2.1 Writing a ConvNet Application

In the `padd` function below, we add zeros around the border of the image.

```
padd <- function(A_prev, pad) {
    n <- dim(A_prev)
    new = array(0, dim = c(n[1] + 2 * pad, n[2] + 2 * pad, n[3]))
    seqr = seq(pad + 1, nrow(new) - pad)
    seqc = seq(pad + 1, ncol(new) - pad)
    new[seqr, seqc, 1:n[3]] = A_prev
    return(new)
}
```

In this function, we will implement a single step of convolution, where we apply the filter to a single position of the input. This will be used to build a convolutional unit, which, takes an input volume, applies a filter at every position of the input and outputs another volume.

```
conv_one_step <- function(a_slice_prev, W, b) {
    s = a_slice_prev * W
    Z = sum(s)
    Z = Z + b

    return(Z)
}
```

```
conv_fwd <- function(A_prev, W, b, hyperparams) {

    stride = as.numeric(hyperparams[["stride"]])
```

```
    pad = as.numeric(hyperparams[["pad"]])

    # stride = 1 pad = 0 n_H_prev = nrow(mat); n_W_prev =
    # ncol(mat) n_C = 2 W = fil; b = array(0, c(1, 1, 2))
    f = dim(W)[1]
    n_C = dim(W)[3]

    n_H = ((n_H_prev - f + 2 * pad)/stride) + 1
    n_H
    n_W = ((n_W_prev - f + 2 * pad)/stride) + 1
    n_W

    Z = array(0, dim = c(n_H, n_W, n_C))
    Z
    # A_prev = mat
    A_prev_pad = padd(A_prev, pad)

    a_prev_pad = A_prev_pad

    for (h in 1:n_H) {
        # loop over vertical axis of the output volume loop over
        # horizontal axis of the output volume loop over the channels
        # of the output volume
        for (w in 1:n_W) {
            for (c in 1:n_C) {
                vert_start = h * stride
                vert_end = vert_start + f - 1
                horiz_start = w * stride
                horiz_end = horiz_start + f - 1
                # Use the corners to define the (3D) slice of a_prev_pad
                a_slice_prev = a_prev_pad[vert_start:vert_end,
                  horiz_start:horiz_end, c]
                # print(a_slice_prev) Convolve the slice to get back one o/p
                # neuron
                Z[h, w, c] = conv_one_step(a_slice_prev, W[,
                  , c], b[, , c])
                Z[h, w, c] = conv_one_step(a_slice_prev, W[,
                  , c], b[, , c])
            }
        }
    }
    cache = list(A_prev = A_prev, W = W, b = b, hyperparams = hyperparams)

    return(list(Z = Z, cache = cache))
}
```

Now, we will implement MAX-POOL and AVG-POOL, in the same function.

```
pool_forward <- function(A_prev, hyperparams, mode = "max") {

    n_H_prev = dim(A_prev)[1]  # 4
    n_W_prev = dim(A_prev)[2]  # 4
    n_C_prev = dim(A_prev)[3]  # 2

    stride = as.numeric(hyperparams["stride"])
    f = as.numeric(hyperparams["f"])

    n_H = ((n_H_prev - f)/stride) + 1  # n_H
    n_W = ((n_W_prev - f)/stride) + 1  # n_W
    n_C = n_C_prev
```

```
    # Initialize output matrix A
    A = array(0, dim = c(n_H, n_W, n_C))

    for (h in 1:n_H) {
        for (w in 1:n_W) {
            for (c in 1:n_C) {
                vert_start = h * stride
                vert_end = vert_start + f - 1
                horiz_start = w * stride
                horiz_end = horiz_start + f - 1

                # Use the corners to define the (3D) slice of a_prev_pad
                a_prev_slice = A_prev[vert_start:vert_end, horiz_start:horiz_end,
                  c]

                if (mode == "max")
                  A[h, w, c] = max(a_prev_slice) else if (mode == "average")
                  A[h, w, c] = mean(a_prev_slice)
            }
        }
    }
    cache = list(A_prev = A_prev, hyperparams)

    return(list(A = A, cache = cache))
}
```

In modern deep learning frameworks, we only have to implement the forward pass, and the framework takes care of the backward pass, so most deep learning engineers do not need to bother with the details of the backward pass. The backward pass for convolutional networks is complicated.

Earlier we had implemented a simple (fully connected) neural network, where we used backpropagation to compute the derivatives with respect to the cost to update the parameters. Similarly, in ConvNets, we calculate the derivatives with respect to the cost in order to update the parameters.

The formula to compute dA with respect to the cost for a certain filter W_c and a given training example can be written as

$$dA = + \sum_{h=0}^{n_H} \sum_{w=0}^{n_W} W_c \times dZ_{hw}$$

```
conv_backward <- function(dZ, cache){

  A_prev = cache[['A_prev']]
  W = cache[['W']]
  b = cache[['b']]
  hparams = cache$hyperparams

  # Retrieve dimensions from A_prev's shape
  n_H_prev = dim(A_prev)[1] # 6
  n_W_prev = dim(A_prev)[2] # 6
  n_C_prev = dim(A_prev)[3] # 2

  # Retrieve dimensions from W's shape
  f = dim(W)[1]
  n_C = dim(W)[3]

  # Retrieve information from "hparameters"
  stride = as.numeric(hparams["stride"])
```

```
  pad = as.numeric(hparams["pad"])

  # Retrieve dimensions from dZ's shape
  # dZ = array(rnorm(36), dim = c(6, 6, 2)) # with pad = 1
  # dZ = array(rnorm(16), dim = c(4, 4, 2)) # with pad = 1
  n_H = dim(dZ)[1]
  n_W = dim(dZ)[2]
  n_C = dim(dZ)[3]

  dA_prev = array(0, dim = c(n_H_prev, n_W_prev, n_C_prev))
  dW = array(0, dim = c(f, f, n_C))
  db = array(0, dim = c(1, 1, n_C))

  # Pad A_prev and dA_prev
  A_prev_pad = padd(A_prev, pad)
  dA_prev_pad = padd(dA_prev, pad)

  for(h in 1:n_H){
    for(w in 1:n_W){
      for(c in 1:n_C){
        vert_start = h * stride
        vert_end = vert_start + f -1
        horiz_start = w * stride
        horiz_end = horiz_start + f -1

        # Use the corners to define the slice from a_prev_pad
        a_slice = a_prev_pad[vert_start:vert_end, horiz_start:horiz_end, c]

        # Update gradients for the window and the filter's parameters
        dA_prev_pad[vert_start:vert_end, horiz_start:horiz_end, c] =+ W[, , c] *
          dZ[h, w, c]
        dW[, , c] =+ a_slice * dZ[h, w, c]
        db[, , c] =+ dZ[h, w, c]
      }
    }
  }
  # Set the example's in dA_prev to the unpaded da_prev_pad
  seqr = seq(pad + 1, nrow(dA_prev_pad) - pad)
  seqc = seq(pad + 1, ncol(dA_prev_pad) - pad)
  new[seqr, seqc, 1:n[3]] = A_prev
  dA_prev = dA_prev_pad[seqr, seqc, ]

  return(list(dA_prev, dW, db))
}
```

Next, we will implement the backward pass for the pooling layer, starting with the MAX-POOL layer. Even though a pooling layer has no parameters for backprop to update, we will still need to backpropagate the gradient through the pooling layer in order to compute gradients for layers that came before the pooling layer.

Before getting into the backpropagation of the pooling layer, we will build a helper function called `create_mask_from_window` which creates a mask matrix to keep track of where the maximum of the matrix is located. A $True == 1$ indicates the position of the maximum in the matrix while the other entries are $False == 0$.

```
create_mask_from_window <- function(x) {
    mask = x == max(x)
    return(mask)
}
```

In max pooling, for each input window, all the "influence" on the output came from a single input value—the max. In average- pooling, every element of the input window has equal influence on the output. So to implement backprop, you will now implement a helper function that reflects this.

```
distribute_value <- function(dz, shape) {
    n_H = dim(shape)[1]
    n_W = dim(shape)[2]
    # Compute the value to distribute on the matrix
    average = dz/(n_H * n_W)
    # Create a matrix where every entry is the 'average' value
    a = matrix(rep(average, n_H * n_W), nrow = n_W)
    return(a)
}
```

```
pool_backward <- function(dA, cache, mode = "max"){
  A_prev = cache[['A_prev']]
  hparams = cache[[2]]
  stride = as.numeric(hparams["stride"])
  f = as.numeric(hparams["f"])

  # Retrieve dimensions from A_prev's shape
  n_H_prev = dim(A_prev)[1]
  n_W_prev = dim(A_prev)[2]
  n_C_prev = dim(A_prev)[3]

  n_H = dim(dA)[1]
  n_W = dim(dA)[2]
  n_C = dim(dA)[3]

  dA_prev = array(0, dim = c(n_H_prev, n_W_prev, n_C_prev))

  for(h in 1:n_H){
    for(w in 1:n_W){
      for(c in 1:n_C){
        vert_start = h * stride
        vert_end = vert_start + f -1
        horiz_start = w * stride
        horiz_end = horiz_start + f -1

        if(mode == "max"){
          a_prev_slice = A_prev[vert_start:vert_end, horiz_start:horiz_end, c]
          mask = create_mask_from_window(a_prev_slice)
          dA_prev[vert_start:vert_end, horiz_start:horiz_end, c] =+
            mask * dA[h, w, c]
        }
        else if(mode == "average"){
          da = dA[h, w, c]
          shape = matrix(0 ,nrow = f, ncol = f)
          dA_prev[vert_start:vert_end, horiz_start:horiz_end, c] =+
            distribute_value(da, shape)
        }
      }
    }
  }
  return(dA_prev)
}
```

7.3 Training a ConvNet on a Small DataSet Using `keras`

A "small" data set in neural networks is with respect to the size and depth of the network. We will use the "Dogs vs Cats" data set[1] which has 25,000 images in the training data set. We will break that up and create a subset of 1000 images for training, 500 images for validation and another 500 for testing.

```
data_dir <- "~/data"
new_dir <- "~/data/subset"
dir.create(new_dir)

train_dir_path <- file.path(new_dir, "train")
dir.create(train_dir_path)

validation_dir_path <- file.path(new_dir, "validation")
dir.create(validation_dir_path)

test_dir_path <- file.path(new_dir, "test")
dir.create(test_dir_path)

train_cats_dir <- file.path(train_dir_path, "cats")
dir.create(train_cats_dir)
train_dogs_dir <- file.path(train_dir_path, "dogs")
dir.create(tr_dogs_dir)

validation_cats_dir <- file.path(validation_dir_path, "cats")
dir.create(validation_cats_dir)
validation_dogs_dir <- file.path(validation_dir_path, "dogs")
dir.create(validation_dogs_dir)

test_cats_dir <- file.path(test_dir_path, "cats")
dir.create(test_cats_dir)
test_dogs_dir <- file.path(test_dir_path, "dogs")
dir.create(test_dogs_dir)

fnames <- paste0("cat.", 1:500, ".jpg")
file.copy(file.path(data_dir, fnames), file.path(train_cats_dir))
fnames <- paste0("dog.", 1:500, ".jpg")
file.copy(file.path(data_dir, fnames), file.path(train_dogs_dir))

fnames <- paste0("cat.", 501:750, ".jpg")
file.copy(file.path(data_dir, fnames), file.path(validation_cats_dir))
fnames <- paste0("dog.", 501:750, ".jpg")
file.copy(file.path(data_dir, fnames), file.path(validation_dogs_dir))

fnames <- paste0("dog.", 751:1000, ".jpg")
file.copy(file.path(data_dir, fnames), file.path(test_dogs_dir))
fnames <- paste0("cat.", 751:1000, ".jpg")
file.copy(file.path(data_dir, fnames), file.path(test_cats_dir))
```

```
library(keras)
is_keras_available()
img_width <- 150
img_height <- 150
batch_size <- 32
epochs <- 30
train_samples = 1000
validation_samples = 500
```

[1] Downloaded from https://www.kaggle.com/c/dogs-vs-cats/data on Apr 02, 2018, 07:40 IST.

We are now ready to do some data preprocessing in keras. In short, we need to read the images from a directory, rescale the pixel values between 0 and 255 to [0,1] interval, convert the images into RGB format and convert them to floating point tensors.

In the following function, we set the argument class_mode to "binary", as we will use the binary crossentropy loss.

```
train_generator <- flow_images_from_directory(
  train_dir_path, generator = image_data_generator(rescale = 1 / 255),
  target_size = c(img_height, img_width), color_mode = "rgb",
  class_mode = "binary", batch_size = batch_size,
  shuffle = TRUE, seed = 61)

validation_generator <- flow_images_from_directory(
  validation_dir_path, generator = image_data_generator(rescale = 1 / 255),
  target_size = c(img_width, img_height), color_mode = "rgb",
  classes = NULL, class_mode = "binary", batch_size = batch_size,
  shuffle = TRUE, seed = 61)
```

In our model, we start with a CONV layer having 32 filters of size 3×3, followed by relu activation and a POOL layer with max_pooling.

```
model <- keras_model_sequential()

model %>%
  layer_conv_2d(filter = 32, kernel_size = c(3,3),
                input_shape = c(img_height, img_width, 3)) %>%
  layer_activation("relu") %>%
  layer_max_pooling_2d(pool_size = c(2,2)) %>%

  layer_conv_2d(filter = 64, kernel_size = c(3,3)) %>%
  layer_activation("relu") %>%
  layer_max_pooling_2d(pool_size = c(2,2)) %>%

  layer_conv_2d(filter = 128, kernel_size = c(3,3)) %>%
  layer_activation("relu") %>%
  layer_max_pooling_2d(pool_size = c(2,2)) %>%

  layer_conv_2d(filter = 128, kernel_size = c(3,3)) %>%
  layer_activation("relu") %>%
  layer_max_pooling_2d(pool_size = c(2,2)) %>%

  layer_flatten() %>%
  layer_dense(units = 512, activation = "relu") %>%
  layer_dense(units = 1, activation = "sigmoid")

summary(model)
```

```
_________________________________________________________________________
Layer (type)                 Output Shape              Param #
=========================================================================
conv2d_1 (Conv2D)            (None, 148, 148, 32)      896
_________________________________________________________________________
activation_1 (Activation)    (None, 148, 148, 32)      0
_________________________________________________________________________
max_pooling2d_1 (MaxPooling2D)  (None, 74, 74, 32)     0
_________________________________________________________________________
conv2d_2 (Conv2D)            (None, 72, 72, 64)        18496
_________________________________________________________________________
```

```
activation_2 (Activation)        (None, 72, 72, 64)            0
_________________________________________________________________
max_pooling2d_2 (MaxPooling2D)   (None, 36, 36, 64)            0
_________________________________________________________________
conv2d_3 (Conv2D)                (None, 34, 34, 128)           73856
_________________________________________________________________
activation_3 (Activation)        (None, 34, 34, 128)           0
_________________________________________________________________
max_pooling2d_3 (MaxPooling2D)   (None, 17, 17, 128)           0
_________________________________________________________________
conv2d_4 (Conv2D)                (None, 15, 15, 128)           147584
_________________________________________________________________
activation_4 (Activation)        (None, 15, 15, 128)           0
_________________________________________________________________
max_pooling2d_4 (MaxPooling2D)   (None, 7, 7, 128)             0
_________________________________________________________________
flatten_1 (Flatten)              (None, 6272)                  0
_________________________________________________________________
dense_1 (Dense)                  (None, 512)                   3211776
_________________________________________________________________
dense_2 (Dense)                  (None, 1)                     513
=================================================================
Total params: 3,453,121
Trainable params: 3,453,121
Non-trainable params: 0
_________________________________________________________________
```

Let us compile and fit our model.

```
model %>% compile(loss = "binary_crossentropy", optimizer = optimizer_adam(lr = 0.001,
    decay = 1e-06), metrics = "accuracy")
```

```
history <- model %>% fit_generator(train_generator, steps_per_epoch = as.integer(train_samples/batch_size),
    epochs = epochs, validation_data = validation_generator,
    validation_steps = as.integer(validation_samples/batch_size),
    verbose = 2)
```

In the plot below (Fig. 7.14), we can see that the training accuracy keeps increasing and reaches an accuracy of 100% while the validation accuracy plateaus out after the ninth epoch. This is a clear sign of overfitting (Fig. 7.14).

```
plot(history)
```

```
evaluate_generator(model, validation_generator, validation_samples)
```

```
$loss
[1] 2.415147

$acc
[1] 0.6596493
```

It is always a good idea to save our models!

```
save_model_hdf5(model, paste0(model_path, 'basic_cnn.h5'),
                overwrite = TRUE)
save_model_weights_hdf5(model, paste0(model_path, 'basic_cnn_weights.h5'),
                        overwrite = TRUE)
```

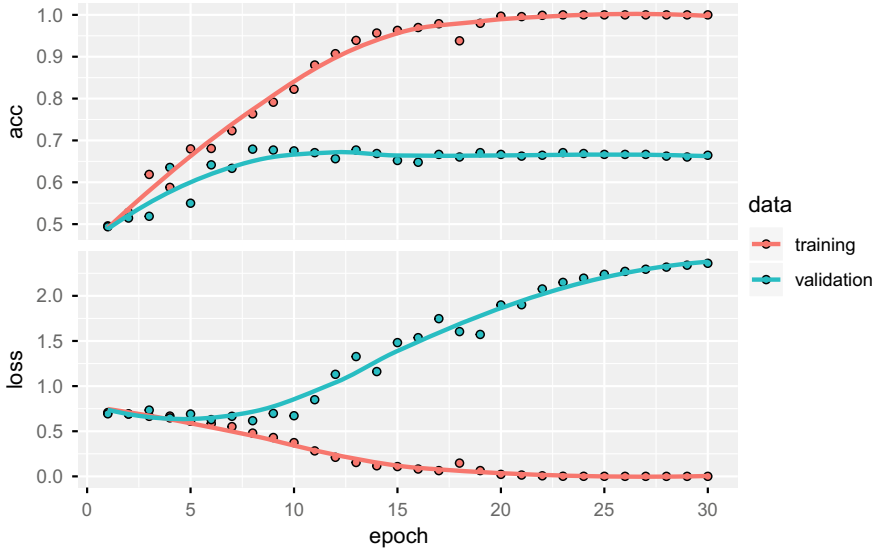

Fig. 7.14 The training set accuracy keeps increasing while the validation set accuracy plateaus out. Similarly, the training loss keeps decreasing and the validation loss keeps increasing

7.3.1 Data Augmentation

When we train a neural network model, we are actually finding the correct estimates of the parameters so that it can map a particular input (say, an image) to some output (a label). Our optimization goal is to find the sweet spot where the model loss is low.

In this case, we are training the neural network with only 1000 examples and around $3e{+}06$ parameters. With such a small data set, we run the risk of overfitting and would either need more data and if that is not available, we can augment our data set. Data augmentation is a process of creating more training data from existing training data.

> If we have a lot of parameters, we need to have a proportional number of examples to get good performance. The number of parameters required is proportional to the complexity of the task our model has to perform.

Some of the data augmentation methods are

(a) General

(i) Mirroring an image
(ii) Shearing an image
(iii) Randomly cropping an image
(iv) Rotating the image, etc.

(b) Color Shifting

(i) Shifting the RGB color range of the image.

(c) Advanced

(i) Apply PCA (Principal Component Analysis) for color augmentation

In `keras`, data augmentation is performed by the `image_data_generator` function. Some of the important arguments in this function are

- `rotation_range` is the value in degrees (0 to 180) within which to randomly rotate the image.
- `width_shift_range` and `height_shift_range` is the fraction of the total width / height within which to randomly shift the images horizontally or vertically.
- `shear_range` randomly applies shear transformation.
- `zoom_range` amount by which to zoom in.
- `horizontal_flip` to allow the images to be randomly flipped horizontally.
- `fill_mode` to fill in newly created pixels, which appear after a rotation or a vertical/horizontal shift, which are, "constant", "nearest", "reflect", or "wrap".
- `data_forma t` RGB "channels first", implying the depth of the image is at index 1 or "channels last", implying where it is 3. The default is the `image_data_format`.

We define the parameters of the augmented features in the next function.

```
augment <- image_data_generator(rescale = 1/255, rotation_range = 50,
    width_shift_range = 0.2, height_shift_range = 0.2, shear_range = 0.2,
    zoom_range = 0.2, horizontal_flip = TRUE, fill_mode = "nearest")
```

Preprocess the data with the augmented features.

```
train_generator_augmented <- flow_images_from_directory(
  train_dir_path, generator = augment,
  target_size = c(img_height, img_width),
  color_mode = "rgb", class_mode = "binary", batch_size = batch_size,
  shuffle = TRUE, seed = 61)

validation_generator <- flow_images_from_directory(
  validation_dir_path, generator = image_data_generator(rescale = 1 / 255),
  target_size = c(img_height, img_width), color_mode = "rgb",
  classes = NULL, class_mode = "binary", batch_size = batch_size,
  shuffle = TRUE, seed = 61)
```

The function texttt{generator_next} retrieves the results from a generator. To understand this function, let us have a look at the first nine results stored in `train_generator_augmented`.

```
batch <- generator_next(train_generator_augmented)
str(batch)
```

```
List of 2
 $ : num [1:32, 1:150, 1:150, 1:3] 0.835 0.676 0.663 0.733 0.885 ...
 $ : num [1:32(1d)] 1 1 0 0 0 0 0 1 1 1 ...
```

Fig. 7.15 Augmented images from the training data set

Let us plot some of our *augmented* images (Fig. 7.15).

```
sp <- par(mfrow = c(3, 3), pty = "s", mar = c(1, 0, 1, 0))
for (i in 1:9) {
    batch <- generator_next(train_generator_augmented)
    plot(as.raster(batch[[1]][1, , , ]))
}
```

```
par(sp)
```

To take care of overfitting, we add dropout regularization in the final layer (Fig. 7.16).

```
model <- keras_model_sequential()

model %>%
  layer_conv_2d(filter = 32, kernel_size = c(3,3),
                input_shape = c(img_height, img_width, 3)) %>%
  layer_activation("relu") %>%
  layer_max_pooling_2d(pool_size = c(2,2)) %>%

  layer_conv_2d(filter = 64, kernel_size = c(3,3)) %>% #32
  layer_activation("relu") %>%
  layer_max_pooling_2d(pool_size = c(2,2)) %>%

  layer_conv_2d(filter = 128, kernel_size = c(3,3)) %>% #64
```

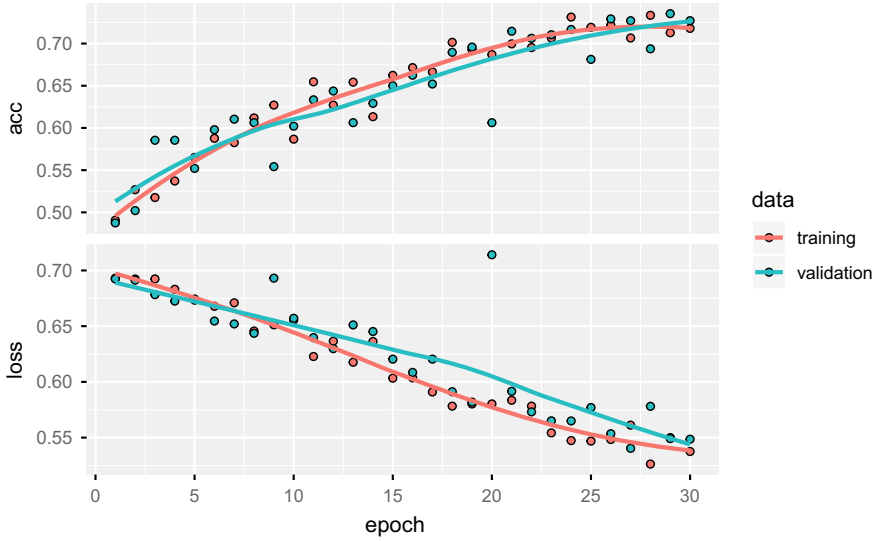

Fig. 7.16 The training and validation accuracies are moving together and the model reduces overfitting substantially

```
  layer_activation("relu") %>%
  layer_max_pooling_2d(pool_size = c(2,2)) %>%

  layer_conv_2d(filter = 128, kernel_size = c(3,3)) %>% #64
  layer_activation("relu") %>%
  layer_max_pooling_2d(pool_size = c(2,2)) %>%

  layer_flatten() %>%
  layer_dropout(rate = 0.5) %>%
  layer_dense(units = 512, activation = "relu") %>%
  layer_dense(units = 1, activation = "sigmoid")
```

```
model %>% compile(loss = "binary_crossentropy", optimizer = optimizer_adam(lr = 1e-04,
    decay = 1e-06), metrics = "accuracy")
```

```
history <- model %>% fit_generator(train_generator_augmented,
    steps_per_epoch = as.integer(train_samples/batch_size), epochs = epochs,
    validation_data = validation_generator, validation_steps = as.integer(validation_samples/batch_size),
    verbose = 2)
```

```
plot(history)
```

```
evaluate_generator(model, validation_generator, validation_samples)
```

```
$loss
[1] 0.5502918

$acc
[1] 0.723701
```

Using dropout regularization and data augmentation, we have substantially reduced overfitting. The validation accuracy has gone up to 74%, an increase of 9% from the previous model.

```
save_model_weights_hdf5(model, paste0(model_path, 'augmented_cnn_weights.h5'),
                overwrite = TRUE)
save_model_hdf5(model, paste0(model_path, 'augmented_cnn.h5'),
                overwrite = TRUE)
```

7.4 Specialized Neural Network Architectures

There are many pretrained specialized neural network models available— VGG-16, $ResNet$, $LeNet$-5, $AlexNet$, etc. Let us have a look at some of them.

7.4.1 LeNet-5

LeNet-5, is a 7-level ConvNet developed by [21] in 1998. using a 32×32 pixel grayscale image as input. The ability to process higher resolution images requires larger and more convolutional layers, so this technique is constrained by the availability of computing resources. The motivation of LeNet-5 was to recognize handwritten digits.

In the first step, LeNet-5 uses a set of six, 5×5 filters with a stride of one and no padding, ending up with a $28 \times 28 \times 6$ cube.

Then it applies average-pooling with a filter width two and a stride of two and that results in a $14 \times 14 \times 6$ cube.

Next, it applies another convolutional layer with a set of 16 filters of size 5×5 thereby ending up with 16 channels of size 10×10 to the next volume.

Then another average-pooling layer, which reduces the height and width by a factor of two, thereby ending up with $5 \times 5 \times 16$.

Multiplying these numbers we get 400 and the next layer is then a fully connected layer with 400 nodes with every one of 120 neurons, so there is a fully connected layer. LeNet then have two fully connected layers with 120 and 84 nodes, respectively. These 84 features are used for one final output, where $\hat{y}$ took on 10 possible values corresponding to each of the digits from 0 to 9.

In 1998, this neural network was small by modern standards, having approximately 60,000 parameters, compared to 10–100 million parameters, which we see in today's neural networks.

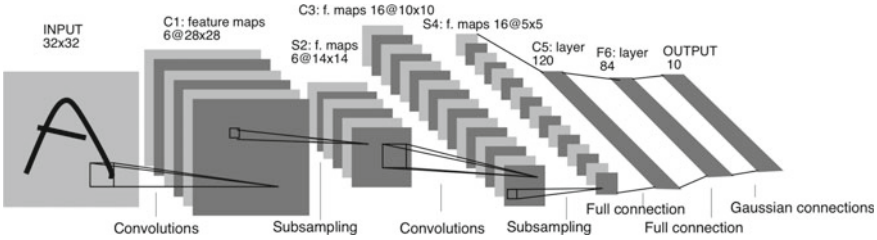

Fig. 7.17 LeNet-5 architecture [21]

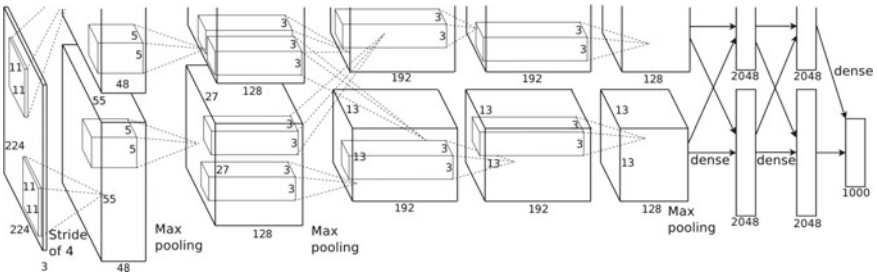

Fig. 7.18 AlexNet architecture—two GPUs run the layer parts of the architecture. [22]

7.4.2 AlexNet

In 2012, AlexNet was developed by [22], which has a very similar architecture as LeNet but much deeper, with more filters per layer, and with stacked convolutional layers. The ImageNet data set was used which is a data set consisting of over 15 million labeled images belonging to approximately 22,000 categories (Fig. 7.18 and Table 7.1).

AlexNet is quite similar to LeNet, but much larger with approximately 62.3 million parameters. The convolution layers, which accounts for 6% of the total number of parameters, consumes 95% of the computation.

It also uses dropout, data augmentation, `relu` activations in the fully connected layers and SGD with momentum = 0.9, a learning rate of 0.01 and weight decay of 0.0005.

7.4.3 VGG-16

VGG-16 is attributed to [23] and consists of 16 ConvNet layers and has a very uniform architecture. It is similar to AlexNet with only 3×3 convolutions with a stride of one and "same" padding; and 2×2 max-pooling layers with a stride of two.

The architecture of this network is depicted in Fig. 7.19.

Table 7.1 AlexNet architecture

Size operation	Filter	Depth	Stride	Padding	Number of parameters
$227 \times 227 \times 3$					
Conv1 + relu	11×11	96	4		$(11 \times 11 \times 3 + 1) \times 96 = 34,944$
$55 \times 55 \times 96$					
Max-pooling	3×3		2		
$27 \times 27 \times 96$					
Norm					
Conv2 + relu	5×5	256	1	2	$(5 \times 5 \times 96 + 1) \times 256 = 614,656$
$27 \times 27 \times 256$					
Max-pooling	3×3		2		
$13 \times 13 \times 256$					
Norm					
Conv3 + relu	3×3	384	1	1	$(3 \times 3 \times 256 + 1) \times 384 = 885,120$
$13 \times 13 \times 384$					
Conv4 + relu	3×3	384	1	1	$(3 \times 3 \times 384 + 1) \times 256 = 884,992$
$13 \times 13 \times 256$					
Max-pooling	3×3		2		
$6 \times 6 \times 256$					
Dropout (rate = 0.5)					
FC6 + relu					$6 \times 6 \times 256 \times 4096 = 37,748,736$
4096					
Dropout (rate = 0.5)					
FC7 + relu					$4096 \times 4096 = 16,7777,216$
4096					
FC8 + relu					$4096 \times 1000 = 4096000$
Overall					62, 369, 152
Conv versus FC					Conv:3.7 million, FC:58.6 million

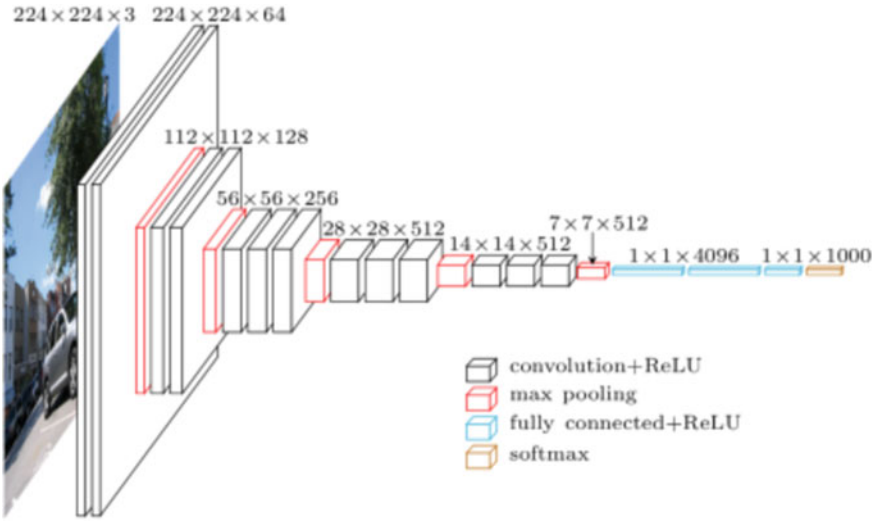

Fig. 7.19 VGG-16 architecture. [23]

The number 16 in the VGG-16 refers to the 16 layers in its architecture, that have weights and has approximately 138 million parameters.

7.4.4 GoogleNet

GoogleNet uses a ConvNet based on LeNet. This also includes a new element, which is termed as the "inception module". This model uses batch normalization, image distortions, and RMSprop and, some very small convolutions so as to reduce the number of parameters.

This architecture consists of a 22-layer deep neural network, but reduces the number of parameters from 60 million (in AlexNet) to approximately, 4 million.

This is a kind of a network, which is also called the **Inception Network**. Discussions on this subject is beyond the scope of this book. However, interested readers are encouraged to read the published papers about the subject.

7.4.5 Transfer Learning or Using Pretrained Models

If we already know the weights (structural parameters) of a model, which has a higher accuracy, we get a head start. This is also called **transfer learning**. Transfer learning is useful when we have a relatively small data set.

A pretrained network model is a model that has been trained on a large data set. If the data set on which the model has been trained was large enough and comparable to our data set, the features learned by the pretrained model can be used to train with a small data set.

It may be of interest to you to think how pretraining on a different task and a different data set can be transferred to a new task with a new data set, which is slightly perturbed from the other.

In `keras`, we have a handful of pretrained models in the ImageNet dataset (containing 1.2 million images with 1000 categories), and they are—$VGG16$, $VGG19$, $MobileNet$, $ResNet50$, etc.

As we are dealing with a data set of 1000 examples, let us experiment with $VGG16$ to extract some of the better features in our data set.

We will do this without data augmentation.

`application_vgg` is a function in `keras` which allows us to load the model. The arguments in this function are

- `include_top`—whether to include the 3 fully connected layers at the top of the network.
- `weights`—ImageNet weights, or the path to the weights file to be loaded.
- `input_shape`- Three inputs channels with a condition that width and height should not be smaller than 48.

```
library(keras)
conv_vgg <- application_vgg16(
  weights = "imagenet",
  include_top = F,
  input_shape = c(150, 150, 3)
)
conv_vgg
```

```
Model
________________________________________________________________________________
Layer (type)                        Output Shape                    Param #
================================================================================
input_1 (InputLayer)                (None, 150, 150, 3)             0
________________________________________________________________________________
block1_conv1 (Conv2D)               (None, 150, 150, 64)            1792
________________________________________________________________________________
block1_conv2 (Conv2D)               (None, 150, 150, 64)            36928
________________________________________________________________________________
block1_pool (MaxPooling2D)          (None, 75, 75, 64)              0
________________________________________________________________________________
block2_conv1 (Conv2D)               (None, 75, 75, 128)             73856
________________________________________________________________________________
block2_conv2 (Conv2D)               (None, 75, 75, 128)             147584
________________________________________________________________________________
block2_pool (MaxPooling2D)          (None, 37, 37, 128)             0
________________________________________________________________________________
block3_conv1 (Conv2D)               (None, 37, 37, 256)             295168
________________________________________________________________________________
block3_conv2 (Conv2D)               (None, 37, 37, 256)             590080
________________________________________________________________________________
block3_conv3 (Conv2D)               (None, 37, 37, 256)             590080
```

```
_________________________________________________________________________
block3_pool (MaxPooling2D)       (None, 18, 18, 256)           0
_________________________________________________________________________
block4_conv1 (Conv2D)            (None, 18, 18, 512)           1180160
_________________________________________________________________________
block4_conv2 (Conv2D)            (None, 18, 18, 512)           2359808
_________________________________________________________________________
block4_conv3 (Conv2D)            (None, 18, 18, 512)           2359808
_________________________________________________________________________
block4_pool (MaxPooling2D)       (None, 9, 9, 512)             0
_________________________________________________________________________
block5_conv1 (Conv2D)            (None, 9, 9, 512)             2359808
_________________________________________________________________________
block5_conv2 (Conv2D)            (None, 9, 9, 512)             2359808
_________________________________________________________________________
block5_conv3 (Conv2D)            (None, 9, 9, 512)             2359808
_________________________________________________________________________
block5_pool (MaxPooling2D)       (None, 4, 4, 512)             0
=========================================================================
Total params: 14,714,688
Trainable params: 14,714,688
Non-trainable params: 0
_________________________________________________________________________
```

7.4.6 *Feature Extraction*

When we use transfer learning, we would like to extract patterns with the highest appearance frequency. If those features are relevant to the data set, they have been trained, it is likely that they would be relevant to a similar data set.

Here, we will extract features learned from the VGG_16 model which was trained on the ImageNet data set (Fig. 7.20).

```
# base_dir <- "~/subset"
train_dir <- file.path(base_dir, "train")
validation_dir <- file.path(base_dir, "validation")
test_dir <- file.path(base_dir, "test")

data_gen <- image_data_generator(rescale = 1 / 255)
batch_size <- 20
```

```
extracted_features <- function(directory, sample){

  features <- array(0, dim = c(sample, 4, 4, 512))
  labels <- array(0, dim = c(sample))

  generator <- flow_images_from_directory(
    directory = directory,
    generator = data_gen,
    target_size = c(150, 150),
    batch_size = batch_size,
    class_mode = "binary"
  )

  i <- 0
  while(TRUE) {
```

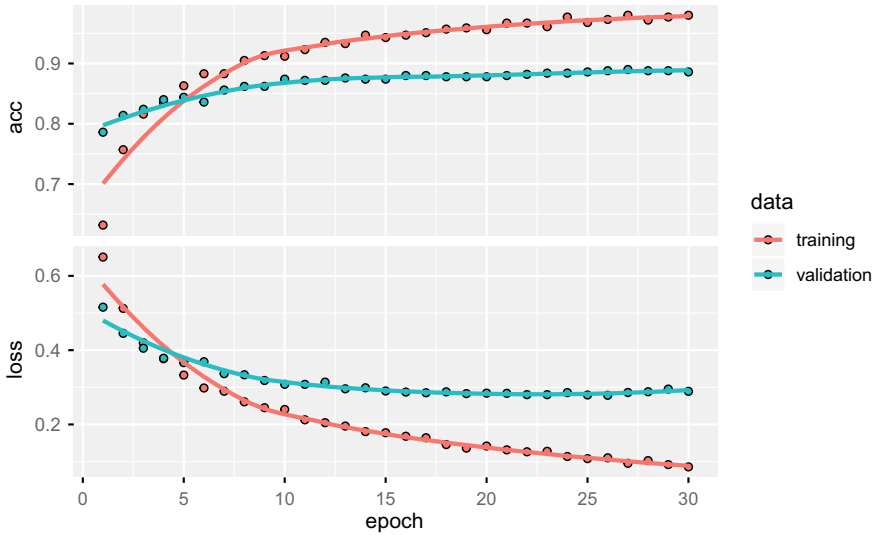

Fig. 7.20 With feature extraction, we have achieved a validation accuracy close to 90%

```
    batch <- generator_next(generator)
    input_batch <- batch[[1]]
    label_batch <- batch[[2]]
    feature_batch <- conv_vgg %>% predict(input_batch)

    index_range <- ((i * batch_size) + 1):((i + 1) * batch_size)
    features[index_range,,,] <- feature_batch
    labels[index_range] <- label_batch

    i <- i + 1
    if(i * batch_size >= sample) break
  }
  list(
    features = features,
    labels = labels
  )
}
```

```
train <- extracted_features(train_dir, 1000)
validation <- extracted_features(validation_dir, 500)
test <- extracted_features(test_dir, 500)


reshape_features <- function(features){
  array_reshape(features, dim = c(nrow(features), 4 * 4 * 512))
}

train$features <- reshape_features(train$features)
validation$features <- reshape_features(validation$features)
test$features <- reshape_features(test$features)
```

```
model <- keras_model_sequential() %>%
  layer_dense(units = 256, activation = "relu", input_shape = 4 * 4 * 512) %>%
```

```
  layer_dropout(rate = 0.5) %>%
  layer_dense(units = 1, activation = "sigmoid")

model %>% compile(
  optimizer = optimizer_rmsprop(lr = 2e-5),
  loss = "binary_crossentropy",
  metrics = "accuracy"
)

history <- model %>% fit(
  x = train$features, y = train$labels,
  epochs = 30,
  batch_size = 20,
  validation_data = list(validation$features, validation$labels)
)
```

```
plot(history)
```

```
evaluate(model, test$features, test$labels)
```

```
$loss
[1] 0.2554676

$acc
[1] 0.892
```

The model delivers a 90% accuracy on the unseen data; starting from an accuracy of 66% this is a remarkable jump, indeed!

Let us save this model.

```
save_model_weights_hdf5(model,
                        paste0(model_path,
                               'featureExtract_vgg16_weights.h5'),
                        overwrite = TRUE)
save_model_hdf5(model, paste0(model_path, 'featureExtract_vgg16.h5'),
                overwrite = TRUE)
```

7.5 What is the ConvNet Learning? A Visualization of Different Layers

What are deep ConvNets really learning? In this section, we will visualize what is going on in the deeper layers of a ConvNet. This will also help us to implement neural style transfer, which is discussed in the next section.

If we start with a hidden unit in layer 1 and scan through the training sets and find out what are the image patches that maximize that unit's activation, i.e., what is the image that maximizes that particular unit's activation.

A hidden unit in layer 1 will see a relatively small portion of the neural network. So, if we pick one hidden unit and find the five input images that maximizes that unit's activation, we might find five image patches.

```
library(keras)
model  <- load_model_hdf5("~/model_path/basic_cnn.h5")
```

```
summary(model)
```

```
_________________________________________________________________________
Layer (type)                 Output Shape              Param #
=========================================================================
conv2d_1 (Conv2D)            (None, 148, 148, 32)      896
_________________________________________________________________________
activation_1 (Activation)    (None, 148, 148, 32)      0
_________________________________________________________________________
max_pooling2d_1 (MaxPooling2D)  (None, 74, 74, 32)     0
_________________________________________________________________________
conv2d_2 (Conv2D)            (None, 72, 72, 64)        18496
_________________________________________________________________________
activation_2 (Activation)    (None, 72, 72, 64)        0
_________________________________________________________________________
max_pooling2d_2 (MaxPooling2D)  (None, 36, 36, 64)     0
_________________________________________________________________________
conv2d_3 (Conv2D)            (None, 34, 34, 128)       73856
_________________________________________________________________________
activation_3 (Activation)    (None, 34, 34, 128)       0
_________________________________________________________________________
max_pooling2d_3 (MaxPooling2D)  (None, 17, 17, 128)    0
_________________________________________________________________________
conv2d_4 (Conv2D)            (None, 15, 15, 128)       147584
_________________________________________________________________________
activation_4 (Activation)    (None, 15, 15, 128)       0
_________________________________________________________________________
max_pooling2d_4 (MaxPooling2D)  (None, 7, 7, 128)      0
_________________________________________________________________________
flatten_1 (Flatten)          (None, 6272)              0
_________________________________________________________________________
dense_1 (Dense)              (None, 512)               3211776
_________________________________________________________________________
dense_2 (Dense)              (None, 1)                 513
=========================================================================
Total params: 3,453,121
Trainable params: 3,453,121
Non-trainable params: 0
_________________________________________________________________________
```

As you can see, our model has 15 layers.

We will now preprocess the image into a 4D tensor.

```
img_path <- "~/path_to_image/vanGogh.png"

img <- image_load(img_path, target_size = c(150, 150))
img_tensor <- image_to_array(img)
img_tensor <- array_reshape(img_tensor, c(1, 150, 150, 3))

img_tensor <- img_tensor / 255

dim(img_tensor)
```

```
[1]   1 150 150   3
```

Fig. 7.21 Plot of the input image

The output shape of the first layer, as reflected in the model summary is $1 \times 148 \times 148 \times 32$.

Let us plot our input image (Fig. 7.21).

We will now extract the outputs of the 15 layers of our ConvNet model and, create a model `act_model`, that will contain the outputs, given the model input.

We then create a function `plot_channel`, which will take the channel number as the argument and plot the channel of the respective layer.

```
layer_outputs <- lapply(model$layers[1:15], function(layer) layer$output)

act_model <- keras_model(inputs = model$input, outputs = layer_outputs)

activations <- act_model %>% predict(img_tensor)
first_layer_act <- activations[[1]]
tenth_layer_act <- activations[[10]]

dim(first_layer_act)
```

```
[1]   1 148 148  32
```

```
dim(tenth_layer_act)
```

```
[1]   1  15  15 128
```

```
plot_channel <- function(channel) {
    rotate <- function(x) t(apply(x, 2, rev))
    image(rotate(channel), axes = FALSE, asp = 1, col = terrain.colors(12))
}
```

We will now use the `plot_channel` function to plot the 1*st* and 32*nd* channels of the first layer.

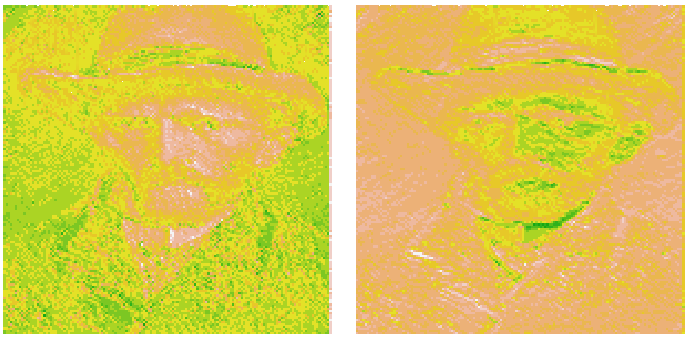

Fig. 7.22 The 1st and 32nd channel activation of the first layer

Fig. 7.23 Plot of three random channels of the tenth layer. The higher layers represent more global and abstract information

The visualisation of the sixth, tenth and thirty-second channels respectively of the tenth layer is shown in Fig. 7.23.

```
par(mfrow = c(1, 2))
plot_channel(first_layer_act[1, , , 1])
plot_channel(first_layer_act[1, , , 32])
```

The first layer, as it appears in Fig. 7.22 is exploring the edges of the face. It is retaining most of the information present in the input image.

```
par(mfrow = c(1, 3))
plot_channel(tenth_layer_act[1, , , 6])
plot_channel(tenth_layer_act[1, , , 10])
plot_channel(tenth_layer_act[1, , , 32])
```

7.6 Introduction to Neural Style Transfer

Neural style transfer [24] is a process by which we apply the style of a reference image to a target image, while preserving the content of the target image. Style is essentially the texture, color, and visual patterns of the reference image and the content is the higher level structure of the target image. Neural style transfer is more akin to signal processing rather than AI.

In neural style transfer, we want to conserve the content of the target image, while applying the style of the reference image. If we can define content and style mathematically, we can define the loss using the distance function (the ℓ_2 norm), which we would like to minimize as follows

loss = distance(style(reference_image) − style(generated_image)) + distance(content(original_image) − content(generated_image))

7.6.1 Content Loss

We know from the previous section that activations from the initial layers represent more local information and the activations from the higher layers contain more abstract information of the image. We, therefore, expect the abstract representations of the image's content to be captured by the later layers of the ConvNet.

If we can think that the final layer of the ConvNet actually captures the content of the target image, we can use a pretrained ConvNet and define the content loss as the ℓ_2 norm between the activations of the final layer, computed over the target image and, the activations of the same layer computed over the generated image.

7.6.2 Style Loss

The style loss aims at capturing the appearance and the style of the reference image. In the paper by [24], the authors use a "Gram matrix" (the inner product between the feature maps of a given layer) of the layer's activations, which represents a mapping of the correlations between the features of a layer. These correlations capture the patterns of the spatial scales, which in turn, corresponds to the appearance of the textures of the image, present at the particular scale.

Style loss, therefore, aims at preserving the internal correlations within the activations of different layers, across the style of the reference image and the generated image.

7.6.3 Generating Art Using Neural Style Transfer

We will now run the gradient-ascent process using the $L\text{-}BFGS$ algorithm, plotting the generated image at each iteration of the algorithm. We will be using two images of *Van Gogh*- "siesta" as the target image and "irises", as the style image (Fig. 7.24). The generated image is plotted in (Figs. 7.25).

Fig. 7.24 On the left is the style image and on the right is the target image

Fig. 7.25 VGG19 network was used with L-BFGS optimization with 20 iterations to obtain the generated image from the style and target images from above

```
library(imager)
img_path <- "~/data"

im1 <- load.image(paste0(img_path, "/irises.png"))
im2 <- load.image(paste0(img_path, "/siesta.png"))
par(mfrow = c(1, 2))
plot(im1, axes = F)
plot(im2, axes = F)
```

```
library(imager)
img_path <- "~/data"

im3 <- load.image(paste0(img_path, "/iris_siesta.png"))
par(mfrow = c(1, 1))
plot(im3, axes = F)
```

7.7 Conclusion

We have explored the architecture of convolutional neural networks. We have discussed transfer learning and used pretrained models for feature extraction. We have also discussed some of the well known pretrained networks and applied them using neural style transfer to create new images, keeping in mind the content loss and style loss attributes of transfer learning.

In the next chapter, we will discuss about sequence models and how we can create character-level language models using LSTMs.

Chapter 8
Recurrent Neural Networks (RNN) or Sequence Models

Machine intelligence is the last invention that humanity will ever need to make.

Nick Bostrom

Abstract In this chapter, we will explore and discuss the basic architecture of sequence models (Recurrent Neural Networks). In particular, we will

- Build and train sequence models, and a commonly used variant known as Long Short-Term Memory Models (LSTMs).
- Apply sequence models to Natural Language Processing (NLP) problems, including text synthesis.

This is the last chapter of the book and, the reader is expected to have a very good understanding of neural networks, including convolutional networks.

8.1 Sequence Models or RNNs

ConvNets generally do not perform well when the input data is interdependent. ConvNets do not have any correlation between the previous input and the next input. Therefore, all the outputs are self-dependent. In ConvNets, if we run 100 different inputs, none of them would be biased by the previous output.

However, for sentence generation or text translation, all the words generated are dependent on the words generated apriori (sometimes it is dependent on words coming after as well). Therefore, we need to have some bias, based on our previous output. This is where sequence models (as the name suggests), come in. Sequence models have a sense of memorizing, what happened earlier in the sequence of data. This helps the system to gain context.

Sequence models make use of sequential information. In our previous neural network models, all inputs (and outputs) are independent of each other. However, in dealing with some real-life problems, this may not be an appropriate approach. If we would want to predict the next word in a sentence we would be better off with the knowledge of the previous sequence of words. Sequence models remember all these

© Springer Nature Singapore Pte Ltd. 2019

A. Ghatak, *Deep Learning with R*,
https://doi.org/10.1007/978-981-13-5850-0_8

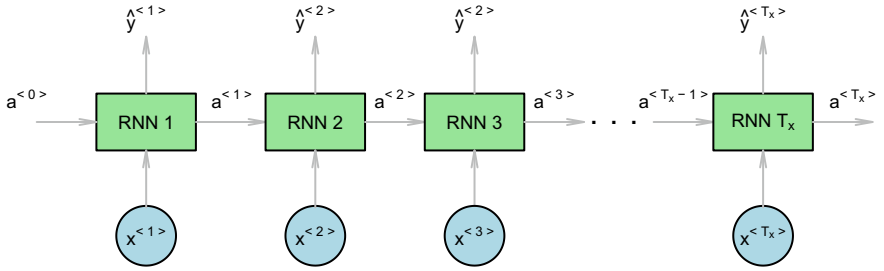

Fig. 8.1 Forward propagation for a basic RNN model

relations while training itself. In these neural network architectures, the output of a particular unit depends on the previous computations.

Sequence models are recurrent because they perform the same task for every element of a sequence. They also exhibit "memory", which captures information about what has been computed up to a time-step. Unlike feedforward neural networks, sequence models can use their internal state (memory) to process sequences of inputs.

We will be using the following notation to describe RNN objects:

- As before, the superscript $[l]$ denotes an object associated with the lth layer.
- Superscript (i) implies that the object is associated with the ith observation.
- Superscript $\langle t \rangle$ denotes an object at the tth time-step.
- Subscript i denotes the ith entry of a vector.

$a_i^{[l]}$ denotes the ith entry of the activations in layer l.
$\hat{y}^{\langle t \rangle (i)}$ is the predicted output at the tth time-step for the ith $observation$.

All sequence models have the form of a chain of repeating modules of neural network. In standard sequence models, this repeating module will have a very simple structure, such as a single `tanh` layer. A typical sequence model chain looks like as shown in Fig. 8.1.

Figure 8.2 is a blown up schematic of a basic RNN cell.

A basic sequence model cell, first takes the $x^{\langle t \rangle}$ from the sequence of input and then it outputs $\hat{y}^{\langle t \rangle}$, which together with $x^{\langle t+1 \rangle}$ is the input for the next step. This way, it keeps remembering the context while training.

After comparing $\hat{y}^{\langle t \rangle}$ to the true labels, we will get the error rate. The error rate is then used to backpropagate using a technique called Backpropagation through Time (`BPTT`). `BPTT` travels back through the network and adjusts the weights based on the error rate.

Theoretically, sequence models can handle context from the beginning of the sentence which will allow more accurate predictions of a word at the end of a sentence. In practice, this is not necessarily true for plain vanilla models and has given rise to Long Short-Term Memory (`LSTM`) unit inside the Neural Network.

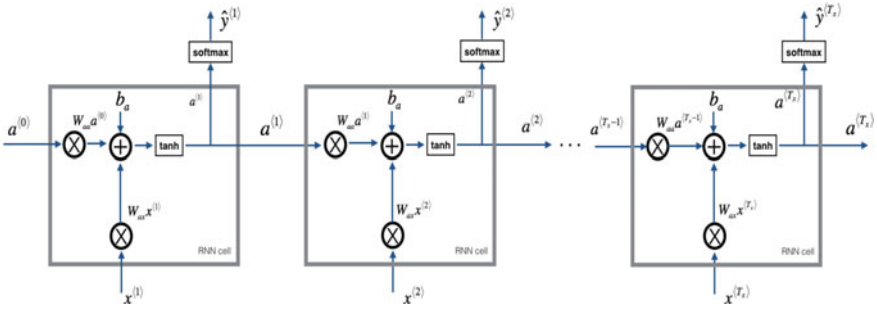

Fig. 8.2 The basic RNN cell takes input $x^{\langle t \rangle}$ (at time-step t) and $a^{\langle t-1 \rangle}$ (previous hidden state containing information from the past), and outputs $a^{\langle t \rangle}$ which is the input to the next RNN cell and, also predicts $y^{\langle t \rangle}$ [**Andrew Ng**]

8.2 Applications of Sequence Models

Some examples where sequence models are used are the following:

- Speech recognition—Predicting phonetic segments based on input sound waves, thus formulating a word.
- Music generation—Only the output Y is a sequence, the input can be an empty set, or it can be a single integer say, referring to the genre of music we would want to generate or maybe the first few notes of the piece of the composition. Here X can be nothing or maybe just an integer and output Y is a sequence.
- DNA sequence analysis—DNAs are represented by four alphabets, namely—A, C, G, and T. Given a DNA sequence label which is part of this DNA sequence corresponds to, say a protein.
- Sentiment classification—The input X is a sequence; therefore, given an input phrase determine the sentiment, say how many stars does the review fetch?
- Machine translation—translate a sentence from one language to the other.
- Video activity recognition—Automatically creating the subtitles of a video for each frame.
- Named Entity recognition—Given a sentence, identify the people in that sentence.

8.3 Sequence Model Architectures

Reference [30] states that "recurrent networks allow us to operate over sequences of vectors" and in Fig. 8.3, he illustrates this by describing each rectangle as a vector and the arrows as functions.

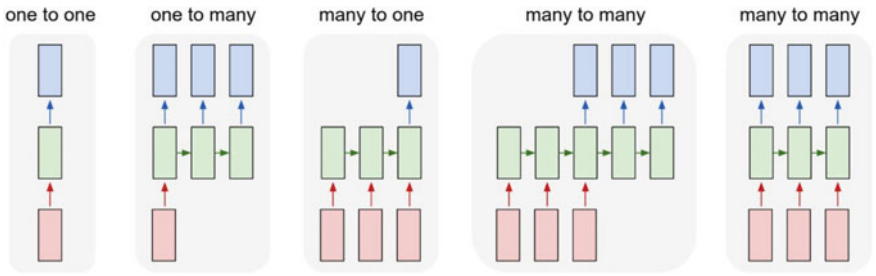

Fig. 8.3 From left to right: (1) Processing without RNN, from fixed-sized input to fixed-sized output (i.e., image classification). (2) Sequence output (i.e., image captioning takes an image and outputs a sentence of words). (3) Sequence input (i.e., sentiment analysis where a given sentence is classified as expressing positive or negative sentiment). (4) Sequence input and sequence output (i.e., Machine Translation: an RNN reads a sentence in English and then outputs a sentence in French). (5) Synced sequence input and output (i.e., video classification where we wish to label each frame of the video). In every case, there are no constraints on the lengths of the sequences because the recurrent transformation (in green) is fixed, and can be applied as many times as we like [30]

8.4 Writing the Basic Sequence Model Architecture

We will code the basic sequence model architecture as follows:

- Compute the hidden state with tanh activation: $a^{\langle t \rangle} = \tanh(W_{aa}a^{\langle t-1 \rangle} + W_{ax}x^{\langle t \rangle} + b_a)$.
- Use the hidden state $a^{\langle t \rangle}$, to compute the prediction $\hat{y}^{\langle t \rangle} = softmax(W_{ya}a^{\langle t \rangle} + b_y)$. We will use the `softmax` activation.
- Store $(a^{\langle t \rangle}, a^{\langle t-1 \rangle}, x^{\langle t \rangle}, parameters)$ in cache
- Return $a^{\langle t \rangle}$, $y^{\langle t \rangle}$ and cache

$x^{\langle t \rangle}$ will have dimension (n_x, m), and $a^{\langle t \rangle}$ will have dimension (n_a, m).

```
softmax <- function(Z) {
  exp_scores = exp(t(Z))
  A = exp_scores / rowSums(exp_scores)
  return(A)
}

sigmoid <- function(x){
  1 / (1 + exp(-x))
  }
```

In this example, $T_x = T_y$.

```
rnn_cell_forward <- function(xt, a_prev, parameters){

  Wax = (parameters[["Wax"]])
  Waa = parameters[["Waa"]]
  Wya = parameters[["Wya"]]
  ba = parameters[["ba"]]
```

```
  by = parameters[["by"]]

  a_next = sinh((Waa %*% a_prev) + t(apply(Wax %*% xt, 1 , function(x) x + ba))) /
    cosh((Waa %*% a_prev) + t(apply(Wax %*% xt, 1 , function(x) x + ba)))
  yt_pred = t(softmax((Wya %*% a_next) + by))
  cache = list('a_next' = a_next,
               'a_prev' = a_prev,
               'xt' = xt,
               'params' = parameters)

  return(list('a_next' = a_next, "yt_pred" = yt_pred, "cache" = cache))
}
```

An sequence model is the repetition of the cell shown in Fig. 8.2. Each cell takes as input the hidden state from the previous cell ($a^{\langle t-1 \rangle}$) and the current time-step's input data ($x^{\langle t \rangle}$). It outputs a hidden state ($a^{\langle t \rangle}$) and a prediction ($y^{\langle t-1 \rangle}$) for this time-step.

```
rnn_forward <- function(x, a0, parameters) {

    caches = NULL

    n_x = dim(x)[1]
    m = dim(x)[2]
    T_x = dim(x)[3]
    n_y = dim(parameters[["Wya"]])[1]
    n_a = dim(parameters[["Wya"]])[2]

    a = array(0, c(n_a, m, T_x))
    y_pred = array(0, c(n_y, m, T_x))

    a_next = a0

    for (t in 1:T_x) {
        a_next = rnn_cell_forward(x[, , t], a_next, parameters)[["a_next"]]
        yt_pred = rnn_cell_forward(x[, , t], a_next, parameters)[["yt_pred"]]
        cache = rnn_cell_forward(x[, , t], a_next, parameters)
        a[, , t] = a_next
        y_pred[, , t] = yt_pred

        caches = append(caches, list(cache))
    }
    return(list(a = a, y_pred = y_pred, caches = caches))
}
```

```
set.seed(1)
x = array(rnorm(3*10*4), dim = c(3,10,4))
a0 = matrix(rnorm(50), nrow =5)
Waa = matrix(rnorm(25), nrow =5)
Wax = matrix(rnorm(15), nrow =5)
Wya = matrix(rnorm(10), nrow =2)
ba = rnorm(5)
by = rnorm(2)
parameters = list("Waa"= Waa, "Wax"= Wax, "Wya"= Wya, "ba"= ba, "by"= by)

a  = rnn_forward(x, a0, parameters)[["a"]]; dim(a)
```

```
[1]  5 10  4
```

```
y_pred = rnn_forward(x, a0, parameters)[["y_pred"]]; dim(y_pred)
```

```
[1]  2 10  4
```

```
caches = rnn_forward(x, a0, parameters)[["caches"]]; length(caches)
```

```
[1] 4
```

8.4.1 Backpropagation in Basic RNN

```
rnn_cell_backward <- function(da_next, cache){

  a_next = cache[["a_next"]]
  a_prev = cache[["a_prev"]]
  xt = cache[["xt"]]
  parameters = cache[["params"]]

  # Retrieve values from parameters
  Wax = parameters[["Wax"]]
  Waa = parameters[["Waa"]]
  Wya = parameters[["Wya"]]
  ba = parameters[["ba"]]
  by = parameters[["by"]]

  dtanh = (1- a_next^2) * da_next

  dxt = t(Wax) %*% dtanh
  dWax = dtanh %*% t(xt)

  da_prev = t(Waa) %*% dtanh
  dWaa = dtanh %*% t(a_prev)

  dba = rowSums(dtanh)

  gradients = list("dxt" = dxt,
                   "da_prev" = da_prev,
                   "dWax" = dWax,
                   "dWaa" = dWaa,
                   "dba" = dba)

  return(gradients)
}
```

```
set.seed(1)
xt = matrix(rnorm(30), nrow = 3)
a_prev = matrix(rnorm(50), nrow = 5)
Wax = matrix(rnorm(15), nrow = 5)
Waa = matrix(rnorm(25), nrow = 5)
Wya = matrix(rnorm(10), nrow = 2)
ba = rnorm(5)
by = rnorm(2)
parameters = list("Wax" = Wax, "Waa" = Waa, "Wya" = Wya, "ba" = ba, "by" = by)

a_next = rnn_cell_forward(xt, a_prev, parameters)[["a_next"]]
```

```
yt = rnn_cell_forward(xt, a_prev, parameters)[["yt_pred"]]
cache = rnn_cell_forward(xt, a_prev, parameters)[["cache"]]

da_next = matrix(rnorm(50), nrow = 5)
gradients = rnn_cell_backward(da_next, cache)

dim(gradients[["dxt"]])
```

```
[1]  3 10
```

```
dim(gradients[["da_prev"]])
```

```
[1]  5 10
```

```
dim(gradients[["dWax"]])
```

```
[1] 5 3
```

```
dim(gradients[["dWaa"]])
```

```
[1] 5 5
```

```
length(gradients[["dba"]])
```

```
[1] 5
```

```
rnn_backward <- function(da, cache) {

  caches = cache[[1]]
  caches = caches$cache
  x = cache[[2]]

  a1 = caches$a_next
  a0 = caches$a_prev
  x1 = caches$xt
  parameters = caches$params

  n_a = dim(da)[1]
  m = dim(da)[2]
  T_x = dim(da)[3]

  n_x = dim(x1)[1]
  m = dim(x1)[2]

  dx = array(0, dim = c(n_x, m, T_x))
  dWax = matrix(0, nrow = n_a, ncol = n_x)
  dWaa = matrix(0, nrow = n_a, ncol = n_a)
  dba = rep(0, n_a)
  da0 = matrix(0, nrow = n_a, ncol = m)
  da_prevt = matrix(0, nrow = n_a, ncol = m)

  for(t in T_x:1){
    gradients = rnn_cell_backward(da[, , t] + da_prevt, caches)
    dxt  = gradients[["dxt"]]
    da_prevt = gradients[["da_prev"]]
    dWaxt = gradients[["dWax"]]
```

```
    dWaat = gradients[["dWaa"]]
    dbat = gradients[["dba"]]

    dx[, , t] = dxt
    dWax =+ dWaxt
    dWaa =+ dWaat
    dba =+ dbat

    da0 = da_prevt

    gradients = list("dx" = dx,
                     "da0" = da0,
                     "dWax" = dWax,
                     "dWaa" = dWaa,
                     "dba" = dba)

    return(gradients)
  }
}
```

```
set.seed(1)
x = array(rnorm(3*10*4), dim = c(3, 10, 4))
a0 = matrix(rnorm(50), nrow = 5)
Wax = matrix(rnorm(15), nrow = 5)
Waa = matrix(rnorm(25), nrow = 5)
Wya = matrix(rnorm(10), nrow = 2)
ba = rnorm(5)
by = rnorm(2)
parameters = list("Wax" = Wax, "Waa" = Waa, "Wya" = Wya, "ba" = ba, "by" = by)
a = rnn_forward(x, a0, parameters)[["a"]]
y = rnn_forward(x, a0, parameters)[["y_pred"]]
cache = rnn_forward(x, a0, parameters)[["caches"]]
da = array(rnorm(5*10*4), dim = c(5, 10, 4))
gradients = rnn_backward(da, cache)

dim(gradients$dx)
```

```
[1]  3 10  4
```

```
dim(gradients$da0)
```

```
[1]  5 10
```

```
dim(gradients$dWax)
```

```
[1] 5 3
```

```
dim(gradients$dWaa)
```

```
[1] 5 5
```

```
length(gradients$dba)
```

```
[1] 5
```

8.5 Long Short-Term Memory (LSTM) Models

Long Short-Term Memory networks (LSTMs) are a special type of recurrent networks which are capable of learning long-term dependencies and was introduced by [29].

8.5.1 The Problem with Sequence Models

Vanishing/Exploding Gradients

Plain vanilla sequence models are not quite used very often in practice. The main reason behind is the vanishing gradient problem. For these models, ideally, we would want to have long memories, so the network can connect data relationships over distances in time and thus make real progress in understanding the context of the language, etc. However, in plain vanilla sequence models, the more time-steps we have, the more is the chance we have of backpropagation gradients either exploding or vanishing. To overcome this problem, we have LSTMs.

The Problem of Long-Term Dependencies

The appeal of sequence model was that they could be able to connect previous information to the present task. Sometimes, we only need to look at recent information to perform the present task. For example, consider a language model trying to predict the next word based on the previous ones. If we are trying to predict the last word in "aeroplanes fly in the sky," we are pretty sure that last word is sky. In such cases, where the gap between the relevant information and the time-step where it is required is small, the model can learn to use the past information.

But when the gap between the relevant information and the time-step where it is required is large, RNNs are unable to connect the information. Consider predicting the last word in the sentence "I grew up in China which is a country of different cultures, … and therefore I speak fluent Chinese." The most recent information suggests that the last word is probably the name of a language. If our sequence model needs to predict which language, it needs the context of the country, which could be way further back in time-steps. It is possible for the gap between the relevant information and the time-step, where it is needed to become very large. And as that gap grows, the model becomes unable to learn to connect the information.

This problem of sequence models was explored by [29] in their paper in 1994, wherein they found some fundamental reasons, for why it might be difficult.

Thankfully, LSTMs help us to surmount the above two problems.

LSTMs are a special kind of sequence models (RNNs), which are capable of learning long-term dependencies. They were introduced by [29] in 1997, which was successively improved by many other researchers. Adding the LSTM to the network is like adding a memory unit that can remember context from the very beginning of the input. LSTMs are, therefore, designed to avoid the long-term dependency problem and, remembering information for long time- steps comes quite naturally to

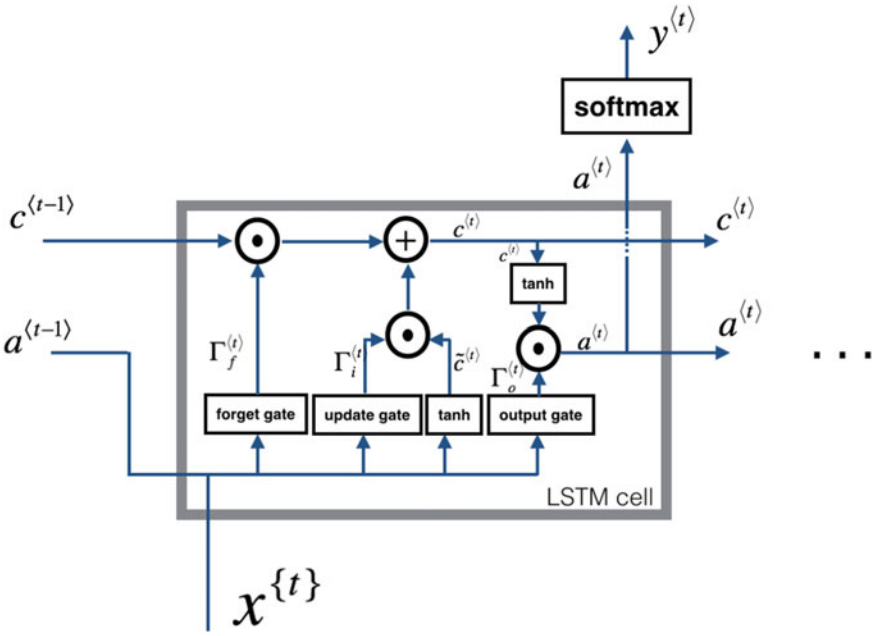

Fig. 8.4 An LSTM cell [**Andrew Ng**]

them. LSTMs work very well on a large variety of problems, and is presently very widely used.

Figure 8.4 describes an LSTM cell. The horizontal line running through the top of the diagram runs straight down the entire chain, with only some minor linear interactions. Information flows along this line unchanged.

Gates are positioned to optionally let information through. They are composed out of a sigmoid neural net layer and a pointwise multiplication operation. The sigmoid layer outputs numbers between zero and one, describing how much of each component should be let through. A value of zero implies that nothing is let through and a value of one implies that all information is let through.

8.5.2 Walking Through LSTM

An LSTM has three of these gates, to protect and control the cell state.

The first step in LSTM is to decide what information will be thrown away from the cell state and is made by the "forget gate layer" having a `sigmoid` activation. It looks at $y^{\langle t \rangle}$ and $y^{\langle t \rangle}$, and outputs a number between 0 and 1, for each number in the cell state $c^{\langle t-1 \rangle}$. A value of 1 signifies that all information is kept and a value of 0 signifies all information is thrown away.

$$\Gamma_u = \sigma(W_n[a^{\langle t-1\rangle}, x^{\langle t\rangle}] + b_u) \tag{8.5.1}$$

The next step is to decide what new information is required to be updated in the cell state. This has two parts. First, a `sigmoid` layer called the "update gate layer" decides which values are to be updated. Next, a `tanh` layer creates a vector of new candidate values, $\tilde{c}^{\langle t\rangle}$, that could be added to the state. These two layers are then combined to create an update to the state.

$$\begin{aligned} \Gamma_f &= \sigma(W_f[a^{\langle t\rangle}, x^{\langle t\rangle}] + b_f) \\ \tilde{c}^{\langle t\rangle} &= tanh(W_c[\Gamma_a^{\langle t-1\rangle}, x^{\langle t\rangle}] + b_c) \end{aligned} \tag{8.5.2}$$

It is now time to update the old cell state, $c^{\langle t-1\rangle}$, into the new cell state $c^{\langle t\rangle}$. We multiply the old state by Γ_f, (so as to forget the things we decided to forget earlier) and then add it to $\Gamma_u * \tilde{c}^{\langle t\rangle}$. This is the new candidate values, scaled by how much we decided to update each state value.

$$c^{\langle t\rangle} = \Gamma_u * \tilde{c}^{\langle t\rangle} + \Gamma_f * c^{\langle t-1\rangle} \tag{8.5.3}$$

Finally, we now need to decide what we are going to output. This output will be based on our cell state, but will be a filtered version. First, we run a sigmoid layer which decides what parts of the cell state we are going to output. Then, we put the cell state through `tanh` activation, (pushing the values to lie between -1 and 1), and multiply it by the output of the `sigmoid` gate, so that we only output the parts we decided to.

$$\begin{aligned} \Gamma_o &= \sigma(W_o[a^{\langle t-1\rangle}, x^{\langle t\rangle}] + b_o) \\ a^{\langle t\rangle} &= \Gamma_o * c^{\langle t\rangle} \end{aligned} \tag{8.5.4}$$

A variation of the LSTM is the **Gated Recurrent Unit** (GRU), introduced by [27] in 2014. It combines the forget and input gates into a single "update gate" and also merges the cell state and hidden state, and makes some other changes. The resulting model is simpler than standard LSTM models, and is quite popular.

We will not discuss GRUs here, but the reader is advised to look up on relevant papers on the subject.

8.6 Writing the LSTM Architecture

```
lstm_cell_forward <- function(xt, a_prev, c_prev, parameters){


  Wf = parameters[["Wf"]]
  bf = parameters[["bf"]]
```

```
  Wi = parameters[["Wi"]]
  bi = parameters[["bi"]]
  Wc = parameters[["Wc"]]
  bc = parameters[["bc"]]
  Wo = parameters[["Wo"]]
  bo = parameters[["bo"]]
  Wy = parameters[["Wy"]]
  by = parameters[["by"]]

  n_x = dim(xt)[1]; n_x
  m = dim(xt)[2]; m
  n_y = dim(Wy)[1]; n_y
  n_a = dim(Wy)[2]; n_a

  concat = matrix(0, nrow = n_a + n_x, ncol = m); concat
  concat[1:n_a, 1:m] = a_prev
  concat[(n_a + 1):(n_a + n_x), 1:m] = xt

  ft = sigmoid((Wf %*% concat) + bf)
  it = sigmoid((Wi %*% concat) + bi)
  cct = sinh((Wc %*% concat) + bc) / cosh((Wc %*% concat) + bc)
  c_next = ft * c_prev + it * cct
  ot = sigmoid((Wo %*% concat) + bo)
  a_next = ot * (sinh(c_next) / cosh(c_next))

  yt_pred = t(softmax((Wy %*% a_next) + by))

  cache = list("a_next" = a_next,
               "c_next" = c_next,
               "a_prev" = a_prev,
               "c_prev" = c_prev,
               "ft" = ft,
               "it" = it,
               "cct" = cct,
               "ot" = ot,
               "xt" = xt,
               "parameters" = parameters)

  return(list("a_next" = a_next,
              "c_next" = c_next,
              "yt_pred" = yt_pred,
              "cache" = cache))
}
```

```
set.seed(1)
xt = matrix(rnorm(30), nrow = 3)
a_prev = matrix(rnorm(50), nrow = 5)
c_prev = matrix(rnorm(50), nrow = 5)
Wf = matrix(rnorm(5*8), nrow = 5)
bf = rnorm(5)
Wi = matrix(rnorm(5*8), nrow = 5)
bi = rnorm(5)
Wo = matrix(rnorm(5*8), nrow = 5)
bo = rnorm(5)
Wc = matrix(rnorm(5*8), nrow = 5)
bc = rnorm(5)
Wy = matrix(rnorm(10), nrow = 2)
by = rnorm(2)
parameters = list("Wf" =  Wf, "Wi" =  Wi, "Wo" =  Wo, "Wc" =  Wc, "Wy" =  Wy,
                  "bf" = bf, "bi" = bi, "bo" = bo, "bc" = bc, "by" = by)
```

```
a_next  = lstm_cell_forward(xt, a_prev, c_prev, parameters)[["a_next"]]
dim(a_next)
```

```
[1]  5 10
```

```
c_next  = lstm_cell_forward(xt, a_prev, c_prev, parameters)[["c_next"]]
dim(c_next)
```

```
[1]  5 10
```

```
yt_pred  = lstm_cell_forward(xt, a_prev, c_prev, parameters)[["yt_pred"]]
dim(yt_pred)
```

```
[1]  2 10
```

```
cache = lstm_cell_forward(xt, a_prev, c_prev, parameters)[["cache"]]
length(cache)
```

```
[1] 10
```

```
lstm_forward <- function(x, a0, parameters){

  caches = NULL

  n_x = dim(x)[1]
  m = dim(x)[2]
  T_x = dim(x)[3]
  n_y = dim(parameters[["Wy"]])[1]
  n_a = dim(parameters[["Wy"]])[2]

  a = array(0, dim = c(n_a, m, T_x))
  c = a
  y = array(0, dim = c(n_y, m, T_x))

  a_next = a0
  c_next = matrix(0, nrow = dim(a_next)[1], ncol = dim(a_next)[2])

  for(t in 1:T_x) {
    a_next = lstm_cell_forward(x[ , , t], a_next, c_next, parameters)[["a_next"]]
    c_next = lstm_cell_forward(x[ , , t], a_next, c_next, parameters)[["c_next"]]
    yt = lstm_cell_forward(x[ , , t], a_next, c_next, parameters)[["yt_pred"]]
    cache = lstm_cell_forward(x[ , , t], a_next, c_next, parameters)[["cache"]]

    a[, , t] = a_next
    y[, , t] = yt
    c[, , t]  = c_next

    caches = append(caches, list(cache))
  }
  return(list("a" = a, "y" = y, "c" = c, "caches" = caches))
}
```

```
set.seed(1)
x = array(3*10*7, dim = c(3,10,7))
a0 = matrix(rnorm(50), nrow = 5)
Wf = matrix(rnorm(5*8), nrow = 5)
bf = rnorm(5)
```

```
Wi = matrix(rnorm(5*8), nrow = 5)
bi = rnorm(5)
Wo = matrix(rnorm(5*8), nrow = 5)
bo = rnorm(5)
Wc = matrix(rnorm(5*8), nrow = 5)
bc = rnorm(5)
Wy = matrix(rnorm(2*5), nrow = 2)
by = rnorm(2)
parameters = list("Wf" =  Wf, "Wi" =  Wi, "Wo" =  Wo, "Wc" =  Wc, "Wy" =  Wy,
                  "bf" = bf, "bi" = bi, "bo" = bo, "bc" = bc, "by" = by)

a = lstm_forward(x, a0, parameters)[["a"]]
dim(a)
```

```
[1]  5 10  7
```

```
y = lstm_forward(x, a0, parameters)[["y"]]
dim(y)
```

```
[1]  2 10  7
```

```
c = lstm_forward(x, a0, parameters)[["c"]]
dim(c)
```

```
[1]  5 10  7
```

```
caches = lstm_forward(x, a0, parameters)[["caches"]]
length(caches)
```

```
[1] 7
```

```
lstm_cell_backward <- function(da_next, dc_next, cache){

  a_next = cache[["a_next"]]
  c_next = cache[["c_next"]]
  a_prev = cache[["a_prev"]]
  c_prev = cache[["c_prev"]]
  ft = cache[["ft"]]
  it = cache[["it"]]
  cct = cache[["cct"]]
  ot = cache[["ot"]]
  xt = cache[["xt"]]
  parameters = cache[["parameters"]]

  n_x = dim(xt)[1]
  m = dim(xt)[2]
  n_a = dim(a_next)[1]
  m = dim(a_next)[2]

  dot = da_next * (sinh(c_next) / cosh(c_next)) * ot * (1 - ot)
  dcct = (dc_next * it + ot *
            (1 - (sinh(c_next) / cosh(c_next))^2 *
               it * da_next) *
            (1 - (cct)^2))
  dit = (dc_next * cct + ot *
           (1 - (sinh(c_next) / cosh(c_next))^2 *
              cct * da_next) *
```

```
          it * (1 - it))
  dft = (dc_next * c_prev + ot *
          (1 - (sinh(c_next) / cosh(c_next))^2) *
          c_prev * da_next) * ft * (1 - ft)

  concat = rbind(a_prev, xt)

  dWf = dft %*% t(concat)
  dWi = dit %*% t(concat)
  dWc = dcct %*% t(concat)
  dWo = dot %*% t(concat)
  dbf = rowSums(dft)
  dbi = rowSums(dit)
  dbc = rowSums(dcct)
  dbo = rowSums(dot)

  na_end = dim(parameters[['Wf']])[2]

  da_prev = (t(parameters[['Wf']][, 1:n_a]) %*% dft) +
    (t(parameters[['Wi']][, 1:n_a]) %*% dit) +
    (t(parameters[['Wc']][, 1:n_a]) %*% dcct) +
    (t(parameters[['Wo']][, 1:n_a]) %*% dot)

  dc_prev = dc_next *
    (ft + ot) *
    (1 - sinh(c_next) / cosh(c_next))^2 *
    ft *
    da_next

  dxt = t(parameters[['Wf']][, (n_a + 1):na_end]) %*% dft +
    t(parameters[['Wi']][, (n_a + 1):na_end]) %*% dit +
    t(parameters[['Wc']][, (n_a + 1):na_end]) %*% dcct +
    t(parameters[['Wo']][, (n_a + 1):na_end]) %*% dot

  gradients = list("dxt" = dxt,
                   "da_prev" = da_prev,
                   "dc_prev" = dc_prev,
                   "dWf" = dWf,
                   "dbf" = dbf,
                   "dWi" = dWi,
                   "dbi" = dbi,
                   "dWc" = dWc,
                   "dbc" = dbc,
                   "dWo" = dWo,
                   "dbo" = dbo)

  return(gradients)
}
```

```
set.seed(1)
xt = matrix(rnorm(30), nrow = 3)
a_prev = matrix(rnorm(50), nrow = 5)
c_prev = matrix(rnorm(50), nrow = 5)
Wf = matrix(rnorm(40), nrow = 5, ncol = 5 + 3)
bf = rnorm(5)
Wi = matrix(rnorm(40), nrow = 5, ncol = 5 + 3)
bi = rnorm(5)
Wo = matrix(rnorm(40), nrow = 5, ncol = 5 + 3)
bo = rnorm(5)
Wc = matrix(rnorm(40), nrow = 5, ncol = 5 + 3)
```

```
bc = rnorm(5)
Wy = matrix(rnorm(10), nrow = 2)
by = rnorm(2)

parameters = list("Wf" = Wf,
                  "Wi" = Wi,
                  "Wo" = Wo,
                  "Wc" = Wc,
                  "Wy" = Wy,
                  "bf" = bf,
                  "bi" = bi,
                  "bo" = bo,
                  "bc" = bc,
                  "by" = by)

a_next = lstm_cell_forward(xt, a_prev, c_prev, parameters)[["a_next"]]

c_next = lstm_cell_forward(xt, a_prev, c_prev, parameters)[["c_next"]]

yt = lstm_cell_forward(xt, a_prev, c_prev, parameters)[["yt_pred"]]

cache = lstm_cell_forward(xt, a_prev, c_prev, parameters)[["cache"]]


da_next = matrix(rnorm(50), nrow = 5)

dc_next = matrix(rnorm(50), nrow = 5)

gradients = lstm_cell_backward(da_next, dc_next, cache)

dim(gradients[["dxt"]])
```

```
[1]  3 10
```

```
dim(gradients[["da_prev"]])
```

```
[1]  5 10
```

```
dim(gradients[["dc_prev"]])
```

```
[1]  5 10
```

```
dim(gradients[["dWf"]])
```

```
[1] 5 8
```

```
dim(gradients[["dWc"]])
```

```
[1] 5 8
```

```
length(gradients[["dbf"]])
```

```
[1] 5
```

```
length(gradients[["dbi"]])
```

```
[1] 5
```

```
length(gradients[["dbo"]])
```

```
[1] 5
```

```
lstm_backward <- function(da, caches){

  a1 = caches[[1]][["a_next"]]
  c1 = caches[[1]][["c_next"]]
  a0 = caches[[1]][["a_prev"]]
  c0 = caches[[1]][["c_prev"]]
  f1 = caches[[1]][["ft"]]
  i1 = caches[[1]][["it"]]
  cc1 = caches[[1]][["cct"]]
  o1 = caches[[1]][["ot"]]
  x1 = caches[[1]][["xt"]]
  parameters = caches[[1]][["parameters"]]

  n_a = dim(da)[1]
  m = dim(da)[2]
  T_x = dim(da)[3]
  n_x = dim(x1)[1]
  m = dim(x1)[2]

  dx = array(0, dim = c(n_x, m, T_x))
  da0 = matrix(0, nrow = n_a)
  da_prevt = rep(0, length(da0))
  dc_prevt = rep(0, length(da0))
  dWf = matrix(0, nrow = n_a, ncol = n_a + n_x)
  dWi = rep(0, length(dWf))
  dWc = rep(0, length(dWf))
  dWo = rep(0, length(dWf))
  dbf = rnorm(n_a)
  dbi = rep(0, length(dbf))
  dbc = rep(0, length(dbf))
  dbo = rep(0, length(dbf))

  for(t in T_x:1){
    gradients = lstm_cell_backward(da[, , t], dc_prevt, caches[[t]])
    dx[, ,t] = gradients[["dxt"]]
    dWf =+ gradients[["dWf"]]
    dWi =+ gradients[["dWi"]]
    dWc =+ gradients[["dWc"]]
    dWo =+ gradients[["dWo"]]
    dbf =+ gradients[["dbf"]]
    dbi =+ gradients[["dbi"]]
    dbc =+ gradients[["dbc"]]
    dbo =+ gradients[["dbo"]]
  }

  da0 = gradients[["da_prev"]]

  gradients = list("dx" = dx,
                   "da0" = da0,
                   "dWf" = dWf,
                   "dbf" = dbf,
```

```
                    "dWi" = dWi,
                    "dbi" = dbi,
                    "dWc" = dWc,
                    "dbc" = dbc,
                    "dWo" = dWo,
                    "dbo" = dbo)

  return(gradients)
}
```

```
set.seed(1)
x = array(rnorm(3*10*7), dim = c(3, 10, 7))
a0 = matrix(rnorm(50), nrow = 5)
Wf = matrix(rnorm(40), nrow = 5, ncol = 5 + 3)
bf = rnorm(5)
Wi = matrix(rnorm(40), nrow = 5, ncol = 5 + 3)
bi = rnorm(5)
Wo = matrix(rnorm(40), nrow = 5, ncol = 5 + 3)
bo = rnorm(5)
Wc = matrix(rnorm(40), nrow = 5, ncol = 5 + 3)
bc = rnorm(5)
Wy = matrix(rnorm(10), nrow = 2)
by = rnorm(2)

parameters = list("Wf" = Wf,
                  "Wi" = Wi,
                  "Wo" = Wo,
                  "Wc" = Wc,
                  "Wy" = Wy,
                  "bf" = bf,
                  "bi" = bi,
                  "bo" = bo,
                  "bc" = bc,
                  "by" = by)


a = lstm_forward(x, a0, parameters)[["a"]]
y = lstm_forward(x, a0, parameters)[["y"]]
c = lstm_forward(x, a0, parameters)[["c"]]
caches = lstm_forward(x, a0, parameters)[["caches"]]

da = array(rnorm(5*10*4), dim = c(5, 10, 4))
gradients = lstm_backward(da, caches)
dim(gradients[["dx"]])
```

```
[1]  3 10  4
```

```
dim(gradients[["da0"]])
```

```
[1]  5 10
```

```
dim(gradients[["dWf"]])
```

```
[1] 5 8
```

```
dim(gradients[["dWi"]])
```

```
[1] 5 8
```

```
dim(gradients[["dWc"]])
```

```
[1] 5 8
```

```
dim(gradients[["dWo"]])
```

```
[1] 5 8
```

```
length(gradients[["dbf"]])
```

```
[1] 5
```

```
length(gradients[["dbi"]])
```

```
[1] 5
```

```
length(gradients[["dbc"]])
```

```
[1] 5
```

```
length(gradients[["dbo"]])
```

```
[1] 5
```

8.7 Text Generation with LSTM

In this section, we will learn how RNNs can be used to generate sequence text data. Nowadays, there is a huge amount of data that can be categorized as sequential, namely—data in the form of audio, video, text, time series, sensor data, etc. Examples of text generation include machines writing entire chapters of popular novels like "Game of Thrones" and "Harry Potter", with varying degrees of success.

Working with text data and text generation is challenging which we will gradually explore. We will experiment with some of the novels written by "Shakespeare", to create new literature albeit, restricted to a few hundred words.

8.7.1 Working with Text Data

Text can be considered a form of sequence data, i.e., a sequence of characters or a sequence of words. Applying deep learning to NLP endeavors to identify patterns in the same way it does with images, i.e., pixels of the image. As much as we vectorized the input image to pixels, with text data, we need to *vectorize* the input text in a similar manner, so that we can use it as an input for our deep learning models.

This can be done by

- Transform each word to a vector.
- Transform each character to a vector.
- Extract n-grams of words or characters and transform them to a vector.

Text generators can be extremely tricky. With text data, a model may be trained to make accurate predictions using the sequences that have occurred previously; however, a wrong prediction can make the entire sentence meaningless. However, with numerical sequence data, a wrong prediction could still be considered a valid prediction without any visual (*perceptual*) distinction.

Text generation usually involves the following steps

- Loading of Data
- Creating Character/Word mappings
- Data Preprocessing
- Modeling
- Generating text.

8.7.2 Generating Sequence Data

When we deal with text data, *tokens* signify **words** or **characters**. A *language model* is one which can predict the next *token* with some probability, given the previous *tokens*. We generate sequence data by training a sequence model (or a ConvNet) to predict the next *token* or next few *tokens* in a sequence, using the previous tokens as input.

A text generation model is supposed to capture the *latent space* of the language. Latent means hidden—we observe some data which is in the space that is observable and, map it to a latent space where similar data points are closer together. A neural network extracts the features through many layers (convolutional, recurrent, pooling etc.). In other words, the *latent space* is the space where our features lie.

8.7.3 Sampling Strategy and the Importance of Softmax Diversity

There are two ways in which we can choose the next character, which are as follows:

- Greedy Sampling—consists of choosing the most likely character.
- Stochastic Sampling—introduces randomness in the sampling process.

In stochastic sampling, if the *token* "a" has a 0.45% probability of becoming the next character, the model will choose that character 45% of the time.

We have seen earlier in Sect. 1.11.5 that randomness is measured by the entropy of the probability distribution. The standard `softmax` output often does not offer any strategy to ***control*** the amount of randomness in choosing the next *token*.

`Temperature` (or `diversity`) is a hyperparameter of LSTMs (neural networks, in general), which is used to control the randomness of predictions by scaling the logits before applying softmax. When the temperature is 1, we compute the softmax directly on the logits.

Using a higher `temperature` produces a *softer* probability distribution and makes the sequence model more "excited", resulting in more diversity and also prone to making more mistakes. In contrast, a lower `temperature` produces a conservative output.

Neural networks produce class probabilities with logit vector z, where $z = (z_1, z_2, \ldots, z_n)$, resulting in the following `softmax` output

$$\frac{exp^{\frac{z_i}{T}}}{\sum_j exp^{\frac{z_j}{T}}} \tag{8.7.1}$$

where T is the temperature.

8.7.4 Implementing LSTM Text Generation (Character-Level Neural Language Model)

In this example, we will utilize the first 20 stories written by Shakespeare. We will then take an LSTM layer, feed the layer with strings of n characters, which have been extracted from a text corpus and train it predicts n+1 characters. The output is a `softmax` activation over all possible characters.

A very simple method to represent a word in the vector format is by using a **one-hot encoded** vector, where the digit 1 stands for the position where the word exists and the digit 0 implies everywhere else. The vectorized representation of a word could be written as [0, 0, 0, 0, 0, 1], where 1 is the representation of the word (or token) in the sentence (or string of characters).

```
library(stringr)
library(tokenizers)
library(readr)

# define length of extracted character sequences
maxlen <- 60

textFile <- keras::get_file("pg100.txt",
                    origin =
                    "http://www.gutenberg.org/cache/epub/100/pg100.txt")
text = readLines(textFile)
length(text)
text = text[-(1:173)]
text = text[-(124195:length(text))]
text = paste(text, collapse = " ")
text = strsplit(text, "<<[^>]*>>")[[1]]

dramatis_personae <- grep("Dramatis Personae", text, ignore.case = TRUE)
```

```
  text = text[-dramatis_personae]
  text <- text[-(20:182)]
  text = str_to_lower(text) %>%
  str_c(collapse = "\n")

  text_indexes <- seq(1, nchar(text) - maxlen, by = 3)
  sentences <- str_sub(text, text_indexes, text_indexes + maxlen - 1)

  next_chars <- str_sub(text, text_indexes + maxlen, text_indexes + maxlen)

  chars <- unique(sort(strsplit(text, "")[[1]]))
  cat("Number of unique characters:", length(chars), "\n")

  # map unique characters to their index in 'chars'
  char_indices <- 1:length(chars)
  names(char_indices) <- chars

  # one-hot encode the characters
  x <- array(0L, dim = c(length(sentences), maxlen, length(chars)))
  y <- array(0L, dim = c(length(sentences), length(chars)))

   # Vectorize
   for (i in 1:length(sentences)) {
    sentence <- strsplit(sentences[[i]], "")[[1]]
    for (t in 1:length(sentence)) {
    char <- sentence[[t]]
    x[i, t, char_indices[[char]]] <- 1
    }

    next_char <- next_chars[[i]]
    y[i, char_indices[[next_char]]] <- 1
}
```

The following code is from [26].

```
library(keras)
model <- keras_model_sequential() %>% layer_lstm(units = 128,
    input_shape = c(maxlen, length(chars))) %>% layer_dense(units = length(chars),
    activation = "softmax")

optimizer <- optimizer_rmsprop(lr = 0.01)

model %>% compile(loss = "categorical_crossentropy", optimizer = optimizer)

sample_next_char <- function(preds, diversity = 1) {
    preds <- as.numeric(preds)
    preds <- log(preds)/diversity
    exp_preds <- exp(preds)
    preds <- exp_preds/sum(exp_preds)
    which.max(t(rmultinom(1, 1, preds)))
}

for (epoch in 1:20) {
    cat("epoch", epoch, "\n")

    model %>% fit(x, y, batch_size = 128, epochs = 1)

    # Select a text seed at random
    start_index <- sample(1:(nchar(text) - maxlen - 1), 1)
    seed_text <- str_sub(text, start_index, start_index + maxlen -
        1)
```

```
    cat("Generating with seed:", seed_text, "\n")
    cat("******************************\n")

    for (diversity in c(0.2, 0.6, 1, 1.3)) {

        cat("softmax distribution diversity = ", diversity, "\n")
        cat(seed_text, "\n")

        generated_text <- seed_text

        # Generate 500 characters
        for (i in 1:500) {

            sampled <- array(0, dim = c(1, maxlen, length(chars)))
            generated_chars <- strsplit(generated_text, "")[[1]]
            for (t in 1:length(generated_chars)) {
                char <- generated_chars[[t]]
                sampled[1, t, char_indices[[char]]] <- 1
            }

            preds <- model %>% predict(sampled, verbose = 0)
            next_index <- sample_next_char(preds[1, ], diversity)
            next_char <- chars[[next_index]]

            generated_text <- paste0(generated_text, next_char)
            generated_text <- substring(generated_text, 2)

            cat(next_char)
        }
        cat("\n\n")
    }
}
```

The output presented below is a series of characters predicted by the model. You may appreciate that the predicted characters (which also include white spaces and punctuations), combine together in a sequence to represent, correct interpretations (with a measure of success).

After 20 iterations and using a random text seed, "t; light is an effect of fire, and fire will burn; ergo, l," we get the following:

diversity: 0.1

et me she shall be the the should be promise the world to me the man all the seems that the man that the world that the man is the man is the the stand that the man that i will be the that the man is the the should shall be the man in the spring to the man that the man and the that the man that the man that the man that the self the the thing the man that the man that the world that i will be the the man that the man that the man in the man that with the seems the

diversity: 0.5

et the thing he will the best meason the show my heart, the heart in every contented scanush that strange so kind the great in a hand of this court; the stands of pity in a man with the thoughts prove the point things me to this of ring of the thoughts of it of the forest of all the brand hear the tell profess she gave a shall seem to should

have taked the plain stands of mine own field and she see the brother man that the man that and the heart. where i have

diversity: 1

ikines him, poor some like livies. the is your dauld hold indeed. messenger. if they readye enderting stosilles. parolles. kente; and dromio of syracuse tencus! warned and a a messenger antony musing as you'll littu'st thou my dear rection to cluse and fair alone cage, patch on this gentle. gentle withould would have return service is evils woman and traitorable, that truanter. but i know that peace. duke senior. the honour'ds it. so lusgi'st upon him. dulen. which i

diversity: 1.5

ook werts himery; and, values; route? sodd. saliu; order love, honour, play fors gave now i hoain mode thausaysy aad: a hel't gor bys toiot. helena. prioft to't:est le cal! rooass. nairr to be parop he, may t cleopatra. countrollsdinglh, i s, reasparo! any absenish withal? call it r apfoalacrenh. the flobmime. clowin. go was'ibles neitoxtres dilaat oathsech, civis. by yight. 'tis cutart: ther; bythdy woo inte'ed prisiggssge for willy; hold, hear nese-unrinler. pirioce. ! luci

From the text generated, we can see that at lower temperatures, the text is repetitive and the text consists of real English words; however with higher temperatures, the text produced is a trifle adventurous and it produces new words like "traitorable". The words also appear to be random strings of characters.

8.8 Natural Language Processing

Natural Language Processing (NLP), is broadly defined as the automatic manipulation of natural language, like speech and text. NLP can be used to interpret text and make it analyzable.

Since deep learning architectures are incapable of processing characters or words in their raw form, the text needs to be converted to numbers as inputs.

8.8.1 Word Embeddings

Word embeddings are the texts converted into numbers. A `vocabulary` is a list of all unique words in the English language.

We have seen earlier that a one-hot encoded vector can be used to represent a vectorized representation of word/character (*token*).

Vectorized representation of words through one-hot encoding result in binary (present/not present), sparse (most of the arrays are 0s) and high-dimensional vectors. Imagine, if we have 15,000 word tokens, one-hot encoding could output more than

a 15,000- dimensional vector. Moreover, it just tells us the presence of a token. A way to overcome this problem is by using word embeddings.

Word embeddings learn from data and, they map the words into a geometric space. The distance between two words in a geometric space could be the ℓ_2 norm (or any other distance measurement), between the two word vectors and, the distances relate to the semantic distance between the associated words. In addition, specific directions in the embedded space also carry information of associativity.

Word-embedding spaces are domain specific as the semantic relationship between words differ with different tasks, i.e., a word-embedding space for a model to predict presence of breast cancer will be completely different from that of a restaurant-review sentiment classification model. We should, therefore, learn a new embedding space which is specific to the task at hand. In `keras`, this is quite simple, it is about learning the weights of a layer using the function `layer_embedding()`.

8.8.2 Transfer Learning and Word Embedding

Sometimes, we may not have enough training data in hand, which could make it difficult to learn a task-specific embedding of our vocabulary. In this situation, as we learnt in the section on transfer learning in the previous chapter, we can load embedding vectors from a precomputed embedding space, which captures the semantic aspects of the language structure and reuse the features learned, on a different problem.

Using dense, low-dimensional embedding space was first explored by Bengio [33] in his paper titled *"A Neural Probabilistic Language Mode"*, 2003. The two most common precomputed word embeddings are Word2Vec algorithm, developed by [31], 2013 and, Global Vectors for Word Representation (GloVe) developed by [35]. GloVe is an embedding technique based on factorizing a matrix of word co-occurrence statistics. It is available at https://nlp.stanford.edu/projects/glove, which is a zip file called glove.6B.zip having 100-dimensional embedding vectors for 400,000 words.

The transfer learning process for word embeddings involve the following:

- Learn word embeddings from a large text corpus by downloading a pretrained embedding.
- Transfer embedding to a new task with a smaller training set.
- You may further fine tune the word embeddings with the new data.

Let us use the downloaded embedding vectors to build an index that maps the words to their vector representation.

```
glove_dir = "~/Downloads/glove"

lines <- readLines(file.path(glove_dir, "glove.6B.100d.txt"))

words <- NULL
```

```
embeddings_index <- new.env(hash = TRUE, parent = emptyenv())
for (i in 1:length(lines)) {
    line <- lines[[i]]
    values <- strsplit(line, " ")[[1]]
    word <- values[[1]]
    embeddings_index[[word]] <- as.double(values[-1])
    words[i] <- word
}

cat("Found", length(words), "word vectors.\n")
```

```
Found 400000 word vectors.
```

8.8.3 Analyzing Word Similarity Using Word Vectors

Let us play around with some of the words represented by their vectors. We will use the respective word vectors to find similarities between similar and dissimilar words. For this we will use the cosine similarity function, which is defined as

$$sim_{cosine}(u, v) = \frac{u^T v}{\|u\|_2 \, \|v\|_2} = cosine(\theta)$$

where θ is the angle between the two vectors. For similar words, the angle between the vectors will be small resulting in a large value of $cosine(\theta)$.

To calculate the ℓ_2 norm, we will use the `vector.naorm()` function from the `InspectChangepoint` library.

```
cosine_sim <- function(u, v) {
    library(InspectChangepoint)
    num = sum(u * v)
    norm_u = vector.norm(u)
    norm_v = vector.norm(v)
    den = norm_u * norm_v

    cos_sim <- num/den
    return(cos_sim)
}
```

```
e_man <- as.numeric(embeddings_index[["man"]])
e_woman <- as.numeric(embeddings_index[["woman"]])

e_apple <- as.numeric(embeddings_index[["apple"]])
e_hippopotamus <- as.numeric(embeddings_index[["hippopotamus"]])

e_france <- as.numeric(embeddings_index[["france"]])
e_paris <- as.numeric(embeddings_index[["paris"]])
e_india <- as.numeric(embeddings_index[["india"]])
e_delhi <- as.numeric(embeddings_index[["delhi"]])


cat("cosine similarity(apple, hippopotamus)", cosine_sim(e_apple,
    e_hippopotamus))
```

```
cosine similarity(apple, hippopotamus) -0.09480483
```

```
cat("cosine similarity(man, woman)", cosine_sim(e_man, e_woman))
```

```
cosine similarity(man, woman) 0.8323494
```

```
cat("cosine similarity(france-paris, india-delhi)", cosine_sim((e_france -
    e_paris), (e_india - e_delhi)))
```

```
cosine similarity(france-paris, india-delhi) 0.6974226
```

8.8.4 Analyzing Word Analogies Using Word Vectors

We can also use the word vectors to discover analogies between words, i.e., find which word is the most similar to a given word. The following simple code highlights how we can do that.

```
word_analogy <- function(word_a, word_b, word_c, words) {
    e_a = as.numeric(embeddings_index[[word_a]])
    e_b = as.numeric(embeddings_index[[word_b]])
    e_c = as.numeric(embeddings_index[[word_c]])
    max_cos_sim = -100
    best_word = NULL
    for (w in words) {
        if (!(w %in% c(word_c, word_a, word_b))) {

            e_w = as.numeric(embeddings_index[[w]])
            cos_sim <- cosine_sim((e_b - e_a), (e_w - e_c))

            if (cos_sim > max_cos_sim) {
                max_cos_sim = cos_sim
                best_word = w
            }
        }
    }
    analogy <- paste0(word_a, " -> ", word_b, " : ", word_c,
        " -> ", best_word, sep = "")

    return(analogy)
}
```

```
word_analogy("italy", "italian", "spain", words)
```

```
[1] "italy -> italian : spain -> spanish"
```

```
word_analogy("small", "smaller", "large", words)
```

```
[1] "small -> smaller : large -> larger"
```

```
word_analogy("india", "delhi", "china", words)
```

```
[1] "india -> delhi : china -> beijing"
```

8.8.5 *Debiasing Word Vectors*

Lets first see how the GloVe word embeddings relate to gender. The resulting vector approximately encodes the concept of "gender".

```
g = as.numeric(embeddings_index[["woman"]]) - as.numeric(embeddings_index[["man"]])
g
```

```
  [1]  0.2207500  0.0632200 -0.1176600  0.7332440  0.1124228  0.3521500
  [7] -0.1095900  0.2969030  0.6333900  0.0813400 -0.5658500  0.0360900
 [13]  0.2707400  0.2881100 -0.1544100 -0.1187690 -0.1046800 -0.0312400
 [19]  0.0599500 -0.0622400 -0.7541500 -0.0424780  0.2065190  0.7089200
 [25] -0.1709100 -0.1067500  0.0946500 -0.9140200  0.0144600 -0.1311700
 [31]  0.2762930  0.0438400  0.6295340 -0.1956900  0.0590300 -0.2443100
 [37]  0.0977600 -0.2959600  0.1919100  0.4377150 -0.3925200  0.1424570
 [43] -0.7447100 -0.0361200  0.1059100 -0.0611300 -0.1433850  0.0986680
 [49] -0.0621100 -0.0174600  0.0690700  0.0023600  0.1576600 -0.1122000
 [55] -0.0309300  0.2501000  0.0715700 -0.2485000 -0.0214000 -0.0263000
 [61]  0.0294400 -0.1174000  0.4505370  0.1447440 -0.0188800  0.2010770
 [67] -0.0616200 -0.2057700  0.0748700 -0.2081480 -0.0233400  0.2231390
 [73] -0.2148500  0.8334400  0.1910900  0.7920510 -0.0315700 -0.0403800
 [79] -0.1284900 -0.7932300 -0.8311650 -0.0236300  0.5066970  0.3183280
 [85]  0.0554000 -0.6549000  0.6084610  0.4950300  0.1802440 -0.2314800
 [91]  0.5914400  0.1549300  0.3665900 -0.1381820 -0.2126600  0.4200540
 [97] -0.1771600  0.3906300  0.1124820  0.2315700
```

```
name_list = c("john", "alejandra", "paul", "hari", "marco", "akbar",
    "ivanka", "nusrat", "simran", "elizabeth", "victoria")
for (w in name_list) {
    e_w = as.numeric(embeddings_index[[w]])
    cos_sim <- cosine_sim(e_w, g)
    cat(w, cos_sim, "\n")
}
```

```
john -0.2283502
alejandra 0.2040976
paul -0.2612691
hari -0.1162951
marco -0.2273328
akbar -0.155323
ivanka 0.08449648
nusrat 0.1118457
simran 0.1675252
elizabeth 0.202231
victoria 0.0955887
```

Comparing first names with the vector g, we can see that female first names tend to have a positive cosine similarity, while male first names tend to have a negative cosine similarity.

Let us try an array of different words and let us see how our word vector attributes them in relation to gender.

```
word_list = c("fishing", "science", "arts", "army", "lawyer",
    "engineer", "pilot", "computer", "technology", "receptionist",
    "fashion", "teacher", "singer", "mascara", "literature")

for (w in word_list) {
    e_w = as.numeric(embeddings_index[[w]])
    cos_sim <- cosine_sim(e_w, g)
    cat(w, cos_sim, "\n")
}
```

```
fishing -0.05767286
science -0.02147577
arts 0.01484675
army -0.09906478
lawyer 0.01669994
engineer -0.1230001
pilot -0.04113394
computer -0.1154572
technology -0.1447453
receptionist 0.2806876
fashion 0.08097437
teacher 0.152337
singer 0.1137264
mascara 0.04712166
literature 0.08261854
```

If you read the output, it appears, almost all of the words have a gender bias- "mascara" is closer to the female gender and "science" is closer to the male gender.

Let us endeavor to debias (neutralize) and reduce the bias of these vectors, using an algorithm attributed to [36].

In the following function `neutralize`, we will first calculate the unit vector g, which is in the bias direction; then project the word vector e on the bias vector g and, finally divide the resultant vector by the unit vector of g.

This will give us the biased component of the **word vector**.

We then subtract the biased component of the word vector from the original word vector to get the debiased word vector.

```
neutralize <- function(word, g) {
    e = as.numeric(embeddings_index[[word]])
    unit_vec_g = (g/vector.norm(g))
    dot_product = sum(e * g)
    e_bias_component = dot_product/unit_vec_g
    e_debiased = e - e_bias_component
    return(e_debiased)
}
```

```
word = "receptionist"
e_w = as.numeric(embeddings_index[[word]])
cat("cosine similarity without debiasing:", cosine_sim(e_w, g))
```

```
cosine similarity without debiasing: 0.2806876
```

```
word = "receptionist"
e_db = neutralize(word, g)
cat("cosine similarity after debiasing:", cosine_sim(e_db, g))
```

```
cosine similarity after debiasing: -0.06464001
```

The key idea behind equalization is to make sure that a particular pair of words are equidistant from the n-dimensional $g_{\perp}$ vector.

```
equalize <- function(pair, bias_axis) {
    word_1 <- pair[1]
    word_2 <- pair[2]
    ew_1 <- as.numeric(embeddings_index[[word_1]])
    ew_2 <- as.numeric(embeddings_index[[word_2]])

    mu = (ew_1 + ew_2)/2

    mu_bias = (mu * bias_axis/vector.norm(bias_axis)) * bias_axis
    mu_orth = mu - mu_bias

    ew_1_bias = (ew_1 * bias_axis/vector.norm(bias_axis)) * bias_axis
    ew_2_bias = (ew_2 * bias_axis/vector.norm(bias_axis)) * bias_axis

    corrected_e_w_1_bias = sqrt(abs(1 - sum(mu_orth^2))) * (ew_1_bias -
        mu_bias)/abs(ew_1 - mu_orth - mu_bias)
    corrected_e_w_2_bias = sqrt(abs(1 - sum(mu_orth^2))) * (ew_2_bias -
        mu_bias)/abs(ew_2 - mu_orth - mu_bias)

    e_1 = corrected_e_w_1_bias + mu_orth
    e_2 = corrected_e_w_2_bias + mu_orth

    return(list(e_1, e_2))
}
```

```
word = "man"
e_w = as.numeric(embeddings_index[[word]])
cat("cosine similarity before equalizing for", "'", word, "'",
    "is", cosine_sim(e_w, g))
```

```
cosine similarity before equalizing for ' man ' is -0.1876906
```

```
word = "woman"
e_w = as.numeric(embeddings_index[[word]])
cat("cosine similarity before equalizing for", "'", word, "'",
    "is", cosine_sim(e_w, g))
```

```
cosine similarity before equalizing for ' woman ' is 0.388177
```

```
e_1 = equalize(c("man", "woman"), g)[[1]]
e_2 = equalize(c("man", "woman"), g)[[2]]
cat("cosine similarity after equalizing for", "'", word, "'",
    "is", cosine_sim(e_1, g))
```

```
cosine similarity after equalizing for ' woman ' is -0.4564555
```

```
cat("cosine similarity after equalizing for", "'", word, "'",
    "is", cosine_sim(e_2, g))
```

```
cosine similarity after equalizing for ' woman ' is 0.5599987
```

8.9 Conclusion

We have discussed, explored and constructed sequence models from scratch. We have also used the `keras` API to generate sequence character models to recreate language paragraphs.

To say the least, we have touched upon the basics of recurrent neural networks and its variant—LSTMS. There is still a lot of distance to be covered.

In the final chapter of this book, I will touch upon ways to collaborate and progress further on this fascinating subject.

Chapter 9
Epilogue

Self-education is the only kind of education there is.
Issac Asimov

Congratulations on completing your journey of exploring deep learning with R. Starting with the basics of machine learning, we have explored all the three deep learning architectures and have created our own deep learning applications from scratch. We have learnt many new optimization techniques and understand how they improve convergence in many different ways. We have constructed our own transfer learning models using ConvNets and character generation models using LSTMs.

We have also used the Keras API to create a model up and going quickly. This helps us to understand how we need to shape our hyperparameters, which optimization algorithm to use, and what model architecture is best suited for the data. Thereafter, it is up to our ingenuity and expertise to come up with the best model.

In short, we have now dug our trenches and we have many more miles to go.

9.1 Gathering Experience and Knowledge

One of the best ways to gather experience is by coming up with solutions to real-world problems; and the best way to gather this experience is by trying to solve problems presented at Kaggle (https://www.kaggle.com). There are many organizations who have put up their data on Kaggle to find solutions to their respective problem statements and many of them involve deep learning.

© Springer Nature Singapore Pte Ltd. 2019

A. Ghatak, *Deep Learning with R*,
https://doi.org/10.1007/978-981-13-5850-0_9

9.1.1 Research Papers

Most of the topics discussed in this book were based on recently published research papers and I owe the authors, my gratitude.

Reading old and new research papers is a smart thing to do. Deep learning research and it's related papers are freely available for everyone to read (unlike other subject areas). Most of the authors who have been cited here have had their papers uploaded on [https://arxiv.org]. arXiv, pronounced as "archive" (X is the Greek *chi*), is an open-access server where research papers can be accessed.

It is possible that some of the topics were a bit difficult to grasp but, that should not deter the deep learning enthusiast.

If I can rephrase the catch line from the movie *Forrest Gump*, I would say "Life is like a box of neural nets- you never know what astounding deep learning research paper you get".

9.2 Towards Lifelong Learning

To the discerned reader, a few of learning areas are presented

1. Neural Network Journals
   - *Neural Networks*; Publisher Pergamon Press
   - *Neural Computation*; Publisher MIT Press
   - *IEEE Transactions on Neural Networks*; Publisher Institute of Electrical and Electronics Engineers (IEEE)
   - *International Journal of Neural Systems*; Publisher World Scientific Publishing
   - *International Journal of Neurocomputing*; Publisher Elsevier Science
   - *Neural Network News*; Publisher- AIWeek Inc.
   - *Network: Computation in Neural Systems*; Publisher IOP Publishing Ltd
   - *Connection Science: Journal of Neural Computing, Artificial Intelligence and Cognitive Research*; Publisher- Carfax Publishing
   - *Neural Network News*; Publisher- AIWeek Inc.
   - *The Journal of Experimental and Theoretical Artificial Intelligence*; Publisher Taylor and Francis, Ltd.
2. Neural Network Conferences
   - Neural Information Processing Systems (NIPS)
   - International Joint Conference on Neural Networks (IJCNN)
   - Annual Conference on Neural Networks (ACNN)
   - International Conference on Artificial Neural Networks (ICANN)
   - European Symposium on Artificial Neural Networks (ESANN)
   - Artificial Neural Networks in Engineering (ANNIE)

- International Joint Conference on Artificial Intelligence (IJCAI)
- International Joint Conference on Artificial Intelligence (IJCAI)

3. `keras`

- https://keras.rstudio.com
- https://keras.io
- https://github.com/rstudio/keras
- https://blog.keras.io
- https://tensorflow.rstudio.com/blog.html

9.2.1 Final Words

Once again, thanks for going through this book.

If you have any word of appreciation or otherwise, I would like to know. In either case, it would be a learning.

My best wishes for your journey in deep learning.

References

1. Bush, V. *As we may think*. https://www.theatlantic.com/magazine/archive/1945/07/as-we-may-think/303881/.
2. Young, T., Hazarika, D., Poria, S., & Cambria, E. *Recent trends in deep learning based natural language processing*. arXiv:1708.02709.
3. Rumelhart, D. E., Hinton, G. E., & Williams, R. J. *Learning internal representations by error propagation*. https://web.stanford.edu/class/psych209a/ReadingsByDate/02_06/PDPVolIChapter8.pdf.
4. Breiman, L. *Statistical modeling: The two cultures*. http://www2.math.uu.se/~thulin/mm/breiman.pdf.
5. Ghatak, A. (2017). *Machine learning with R*. Singapore: Springer (ISBN 978-981-10-6807-2).
6. Bergstra, J., & Bengio, Y. (2012). Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, *13*, 281–305. http://www.jmlr.org/papers/volume13/bergstra12a/bergstra12a.pdf.
7. Hinton, G., Srivastava, N., & Swersky, K. *Neural networks for machine learning*. https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf.
8. Hutter, F., Holger, H. H., & Leyton-Brown, K. *Sequential model-based optimization for general algorithm configuration*. https://www.cs.ubc.ca/~hutter/papers/10-TR-SMAC.pdf.
9. Shannon, C. E. (1948). A mathematical theory of communication. *The Bell System Technical Journal*, *27*(379–423), 623–656.
10. Shannon, L. (2001). Statistical modeling: The two cultures. *Statistical Science*, *16*(3). https://projecteuclid.org/download/pdf_1/euclid.ss/1009213726.
11. Schacter, D. (2011). *Psychology*. New York: Worth Publishers.
12. Krizhevsky, A., Sutskever, I., & Hinton, G. E. *ImageNet classification with deep convolutional neural networks*. http://www.cs.toronto.edu//%7Efritz/absps/imagenet.pdf.
13. Knuth: Computers and typesetting. http://www-cs-faculty.stanford.edu/~uno/abcde.html.
14. Glorot, X., & Bengio, Y. *Understanding the difficulty of training deep feedforward neural networks*. http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf/~uno/abcde.html.
15. He, K., Zhang, X., Ren, S., & Sun, J. *Delving deep into rectifiers: surpassing human-level performance on ImageNet classification*. https://arxiv.org/abs/1502.01852.
16. Sutskever, I., Martens, J., Dahl, G., & Hinton, G. (2013). On the importance of initialization and momentum in deep learning. In *Proceedings of Machine Learning Research, PMLR* (Vol. 28, Issue 3, pp. 1139–1147). http://www.cs.utoronto.ca/~ilya/pubs/ilya_sutskever_phd_thesis.pdf/~uno/abcde.html.

© Springer Nature Singapore Pte Ltd. 2019

A. Ghatak, *Deep Learning with R*,
https://doi.org/10.1007/978-981-13-5850-0

17. Bengio, Y., Boulanger-Lewandowski, N., & Pascanu, R. *Advances in optimizing recurrent networks*. arXiv:1212.0901v2.
18. Duchi, J., Hazan, E., & Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, *12*, 2121–2159.
19. Kingma, D. P., & Ba, L. J. (2015). Adam: A method for stochastic optimization. *International Conference on Learning Representations*.
20. Srivastava, N., Hinton, G., Krizhevsky, A., & Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, *15*, 1929–1958. http://jmlr.org/papers/volume15/srivastava14a.old/srivastava14a.pdf.
21. LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*. http://yann.lecun.com/exdb/publis/pdf/lecun-01a.pdf.
22. Krizhevsky, A., Sutskever, I., & Hinton, G. E. ImageNet classification with deep convolutional neural networks. In *NIPS 2012: Neural Information Processing Systems*, Lake Tahoe, Nevada. http://www.cs.toronto.edu/~fritz/absps/imagenet.pdf.
23. Simonyan, K., & Zisserman, A. (2015). Very deep convolutional networks for large scale image recognition ICLR. arXiv:1409.1556.
24. Gatys, L. A., Ecker, A. S., & Bethge, M. (2015). *A neural algorithm of artistic style*. https://www.robots.ox.ac.uk/~vgg/rg/papers/1508.06576v2.pdf.
25. Narayanan, H. *Convolutional neural networks for artistic style transfer*. https://harishnarayanan.org/writing/artistic-style-transfer/.
26. Chollet, F., & Allaire, J. J. *Deep learning with R*. https://www.manning.com/books/deep-learning-with-r.
27. Cho, K., van Merrienboer, B., Bahdanau, D., & Bengio, Y. *On the properties of neural machine translation: encoder-decoder approaches*. arXiv:1409.1259.
28. Chung, J., Gulcehre, C., Cho, K., & Yoshua, B. *Empirical evaluation of gated recurrent neural networks on sequence modeling*. arXiv:1412.3555.
29. Hochreiter, S., & Schmidhuber, J. *Long short term memory*. http://www.bioinf.jku.at/publications/older/2604.pdf.
30. Karpathy, A. *The unreasonable effectiveness of recurrent neural networks*. http://karpathy.github.io/2015/05/21/rnn-effectiveness/.
31. Mikolov, A., Chen, K., Corrado, G., & Dean, J. *Efficient estimation of word representations in vector space*. arXiv:1301.3781.
32. van der Maaten, L., & Hinton, G. (2008). Visualizing data using t-SNE. *Journal of Machine Learning Research*, *9*, 2579–2605. arXiv:1301.3781.
33. Bengio, Y., Ducharme, R., Vincent, P., & Jauvin, C. (2003). A neural probabilistic language model. *Journal of Machine Learning Research*, *3*, 1137–1155. http://www.jmlr.org/papers/volume3/bengio03a/bengio03a.pdf.
34. Mikolov, T., Sutskever, I., Chen, K., Corrado, G., & Dean, J. *Distributed representations of words and phrases and their compositionality*. https://papers.nips.cc/paper/5021-distributed-representations-of-words-and-phrases-and-their-compositionality.pdf.
35. Pennington, J., Socker, R., & Manning, C. *GloVe: Global vectors for word representation*. https://nlp.stanford.edu/pubs/glove.pdf.
36. Bolukbasi, T., Chang, K. W., Zou, J., Saligrama, V., & Kalai, A. *Man is to computer programmer as woman is to homemaker? debiasing word embeddings*. https://papers.nips.cc/paper/6228-man-is-to-computer-programmer-as-woman-is-to-homemaker-debiasing-word-embeddings.pdf.
37. Hinton, G., Oriol V., & Dean, J. *Distilling the knowledge in a neural network*. arXiv:1503.02531.
38. Taigman, Y., Yang, M., Ranzato, M.A., & Wolf, L. (2008). DeepFace: Closing the gap to human-level performance in face verification. *Journal of Machine Learning Research*, *9*, 2579–2605. https://ieeexplore.ieee.org/document/6909616/.
39. Sutskever, I., Vinyals, O., & Le, Q. V. *Sequence to sequence learning with neural networks*. https://papers.nips.cc/paper/5346-sequence-to-sequence-learning-with-neural-networks.pdf.

40. Cho, K., van Merrienboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., et al. *Learning phrase representations using RNN encoder-decoder for statistical machine translation*. arXiv:1406.1078.
41. Mao, J., Xu, W., Yang, Y., Wang, J., Huang, Z., & Yuille, A. *Deep captioning with multimodal recurrent neural networks (m-RNN)*. arXiv:1412.6632
42. Vinyals, O., Toshev, A., Bengio, S., & Erhan, D. *Show and tell: A neural image caption generator*. arXiv:1411.4555.
43. Karpathy, A., Fei-Fei, L., Bengio, S., & Erhan, D. *Deep visual-semantic alignments for generating image descriptions*. https://cs.stanford.edu/people/karpathy/cvpr2015.pdf.
44. Papineni, K., Roukos, S., Ward, T., & Zhu, W. J. (2002). BLEU: A method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics (ACL)*, Philadelphia. https://www.aclweb.org/anthology/P02-1040.pdf.
45. Bahdanau, D., Cho, K., & Bengio, Y. *Neural machine translation by jointly learning to align and translate*. arXiv:1409.0473.
46. Xu, K., Ba, J., Kiros, R., Cho, K., Courville, A., Salakhutdinov, R., et al. *Show, attend and tell: Neural image caption generation with visual attention*. arXiv:1502.03044.
47. Graves, A., Fernandez S., Gomez, F., & Schmidhuber, J. *Connectionist temporal classification: Labelling unsegmented sequence data with recurrent neural networks*. https://www.cs.toronto.edu/~graves/icml_2006.pdf.
48. Young, T., Hazarika, D., Poria, S., & Cambria, E. *Recent trends in deep learning based natural language processing*. arXiv:1708.02709.
49. Wu, Y., Schuster, M., Chen, Z., Le, Q. V., Norouzi, M., et al. *Google's neural machine translation system: Bridging the gap between human and machine translation*. arXiv:1609.08144.
50. *The alan turing internet scrapbook*. https://www.turing.org.uk/scrapbook/test.html.
51. Franceschi, L., Frasconi, P., Salzo, S., Grazzi, R., & Massimiliano, P. *Bilevel programming for hyperparameter optimization and meta-learning*. arXiv:1806.04910.
52. Schmerling, E. *Whose line is it?- quote attribution through recurrent neural networks*. https://cs224d.stanford.edu/reports/edward.pdf.
53. Aleksander, I., & Morton, H. (1990). *An introduction to neural computing*. Boca Raton: Chapman and Hall (ISBN 0-412-37780-2).
54. Beale, R., & Jackson, T. (1990). *Neural computing, an introduction*. Bristol: Adam Hilger, IOP Publishing Ltd (ISBN 0-85274-262-2).
55. Ruineihart, D. E. Hinton, G., & Williams, R. (1985). *Learning internal representation by error propagation*. http://www.dtic.mil/dtic/tr/fulltext/u2/a164453.pdf.
56. LeCun, Y., Bottou, L., Orr, G.B., Muller, K-R. *Efficient BackProp*. http://www.yann.lecun.com/exdb/publis/pdf/lecun-98b.pdf.