Chapter 17
Deep Learning

Manish Gupta

1 Introduction

Deep learning has caught a great deal of momentum in the last few years. Research in the field of deep learning is progressing very fast. Deep learning is a rapidly growing area of machine learning. Machine learning (ML) has seen numerous successes, but applying traditional ML algorithms today often means spending a long time hand-engineering the domain-specific input feature representation. This is true for many problems in vision, audio, natural language processing (NLP), robotics, and other areas. To address this, researchers have developed deep learning algorithms that automatically learn a good high-level abstract representation for the input. These algorithms are today enabling many groups to achieve groundbreaking results in vision recognition, speech recognition, language processing, robotics, and other areas.

The objective of the chapter is to enable the readers:

- Understand what is deep learning
- Understand various popular deep learning architectures, and know when to use which architecture for solving their business problem
- Know how to perform image analysis using deep learning
- Know how to perform text analysis using deep learning

Electronic supplementary material The online version of this chapter (https://doi.org/10.1007/978-3-319-68837-4_17) contains supplementary material, which is available to authorized users.

M. Gupta (✉)
Microsoft Corporation, Hyderabad, India
e-mail: manishg.iitb@gmail.com

B. Pochiraju, S. Seshadri (eds.), *Essentials of Business Analytics*, International Series in Operations Research & Management Science 264,
https://doi.org/10.1007/978-3-319-68837-4_17

Introduction to Deep Learning

Wikipedia defines deep learning as follows. “Deep learning (deep machine learning, or deep structured learning, or hierarchical learning, or sometimes DL) is a branch of machine learning based on a set of algorithms that attempt to model high-level abstractions in data by using model architectures, with complex structures or otherwise, composed of multiple non-linear transformations.” The concept of deep learning started becoming very popular around 2012. This was mainly due to at least two “wins” credited to deep learning architectures. In 2012, Microsoft’s top scientist *Rick Rashid* demonstrated a voice recognition program that translated Rick’s English voice into *Mandarin Chinese* in Tianjin, China.[1] The high accuracy of the program was supported by deep learning techniques. Similarly, in 2012, a deep learning architecture won the ImageNet challenge for the image captioning task.[2]

Now deep learning has been embraced by companies in a large number of domains. After the 2012 success in speech recognition and translation, there has been across the board deployment of deep neural networks (DNNs) in the speech industry. All the top companies in machine learning including Microsoft, Google, and Facebook have been making huge investments in this area in the past few years. Popular systems like IBM *Watson* have also been given a deep learning upgrade. Deep learning is practically everywhere now. It is being used for image classification, speech recognition, language translation, language processing, sentiment analysis, recommendation systems, etc. In medicine and biology, it is being used for cancer cell detection, diabetic grading, drug discovery, etc. In the media and entertainment domain, it is being used for video captioning, video search, real-time translation, etc. In the security and defense domain, it is being used for face detection, video surveillance, satellite imagery, etc. For autonomous machines, deep learning is being used for pedestrian detection, lane tracking, recognizing traffic signs, etc. This is just to name a few use cases. The field is growing very rapidly—not just in terms of new applications for existing deep learning architectures but also in terms of new architectures.

In this chapter, we primarily focus on three deep supervised learning architectures: multilayered perceptrons (MLPs), convolutional neural networks (CNNs), and recurrent neural networks (RNNs). This chapter is organized as follows. In Sect. 2, we discuss the biological inspiration for the artificial neural networks (ANN), the artificial neuron model, the perceptron algorithm to learn the artificial neuron, the MLP architecture and the backpropagation algorithm to learn the MLPs. MLPs are generic ANN models. In Sect. 3, we discuss convolutional neural networks which are an architecture specially designed to

[1] http://deeplearning.net/2012/12/13/microsofts-richard-rashid-demos-deep-learning-for-speech-recognition-in-china/ (accessed on Jan 16, 2018).

[2] https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf (accessed on Jan 16, 2018).

learn from image data. Finally, in Sect. 4, we discuss the recurrent neural networks architecture which is meant for sequence learning tasks (mainly text and speech).

2 Artificial Neural Network (ANN) and Multilayered Perceptron (MLP)

2.1 Biological Inspiration and the Artificial Neuron Model

Deep learning is an extension of research in the area of artificial neural networks (ANNs) as discussed in Chap. 16 on supervised learning. In this section, we elaborate on training a simple neuron using the perceptron algorithm.

Training an artificial neuron involves using a set of labeled examples to estimate the values of the weights w_i (a vector of the same size as the number of features). Rosenblatt (1962) proposed the perceptron algorithm to train the weights of an artificial neuron. It is an iterative algorithm to learn the weight vector. The basic idea is to start with a random weight vector and to update the weights in proportion to the error contributed by the inputs. Algorithm 17.1 presents the pseudo-code for the perceptron algorithm.

Algorithm 17.1: The Perceptron Algorithm

```
1. Randomly initialize weight vector $w_0$
2. Repeat until error is less than a threshold $\gamma$ or max_iterations M:
   (a) For each training example $(x_i, t_i)$:
       • Predict output $y_i$ using current network weights $w_n$
       • Update weight vector as follows: $w_{n+1} = w_n + \eta \times (t_i - y_i) \times x_i$
```

Note that here η is called as the learning rate, t_i is the true label for the instance x_i, and y_i is the predicted class label for the instance x_i. Thus, $t_i - y_i$ is the error made by the neuron with the current weight vector on the instance x_i. Note that a neuron also takes a bias term b as part of the weights to be learned. The bias term is often folded in into the weight vector w by assuming a dummy input and setting it to 1. In that case, the size of the weight vector is number of features + 1. The correction (or the update) of the weights using the perceptron algorithm is equivalent to translation and rotation of the separating hyper-plane for a binary classification problem.

Minsky and Papert (1969) proved that a single artificial neuron is no better than a linear classifier. To be able to learn nonlinear patterns, one can progress in two ways: change the integration function or consider MLP. One can change the integration function from a simple linear weighted summation to a quadratic function $\left(f = \sum_{j=1}^{m} w_j x_j^2 - b\right)$ or a spherical function $\left(f = \sum_{j=1}^{m} \left(x_j - w_j\right)^2 - b\right)$. We will discuss MLP next.

2.2 Multi-layered Perceptrons (MLPs)

Figure 17.1 shows a typical multilayered perceptron architecture. Interestingly such a multilayered perceptron can learn very complex boundaries much more beyond linear boundaries. In fact, it can also learn nonlinear boundaries. It has an input layer, an output layer, and one or more hidden layers. The number of units (neurons) in the input layer corresponds to the dimensionality of the input data. The number of units in the output layer corresponds to the number of unique classes. If there are a large number of hidden layers, the architecture is called a deep learning architecture. The "deep" in "deep learning" refers to the depth of the network due to multiple hidden layers.

In an MLP, each edge corresponds to a weight parameter to be learned. Note that each neuron in a layer k produces an input for every other neuron in the next layer $k + 1$. Thus, this is a case of dense connectivity. Learning the MLP means learning each of these weights. Note that a perceptron cannot be directly used to learn weights for an MLP because there is no supervision available for the output of the internal neurons (neurons in the hidden layers). Thus, we need a new algorithm for training an MLP.

Given a particular fixed weight vector for each edge in the MLP, one can compute the predicted value y_i for any data point x_i. Thus, given a training dataset, one can plot an error surface where each point on the error surface corresponds to a weight configuration.

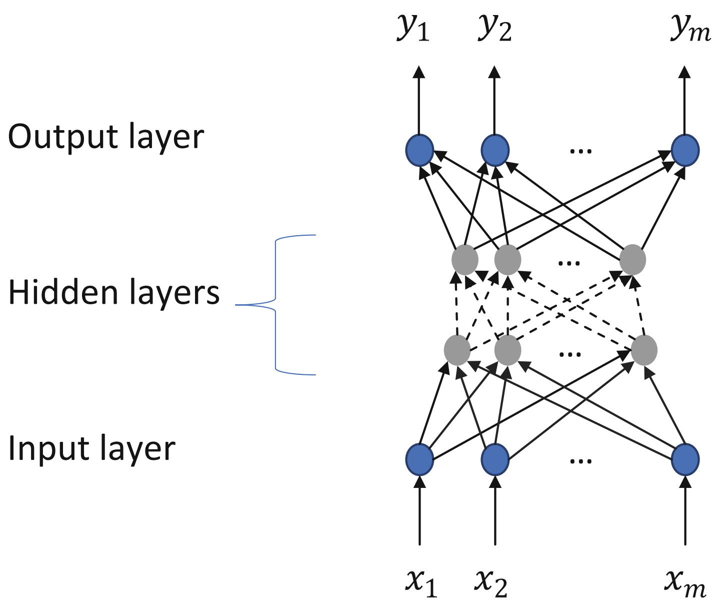

Fig. 17.1 Multilayered perceptron (MLP)

To learn a good weight vector, we present a gradient descent-based algorithm which starts with a random point on this error surface and over multiple iterations moves down the error surface in the hope of finding the deepest valley on this surface. This means that we start with a randomly initialized weight vector and update the weight vector so as to always move in the direction of the negative gradient. Gradient descent algorithms suffer from local minima issues. This means that the valley that we end up at after the gradient descent converges may not be the deepest valley globally. However, just like other algorithms with local optima issues, the problem can be solved by doing multiple runs of gradient descent each with differently initialized weight vectors and then choosing the one with the smallest error. Formally the algorithm is called as back propagation algorithm which works as follows.

Algorithm 17.2: Back Propagation Algorithm

```
1. Initialize network weights (often small random values).
2. Repeat until error is less than a threshold $\gamma$ or max_iterations M:
   (a) For each training example $(x_i, t_i)$:
       • Predict output $y_i$ using current network weights $w_n$ (forward pass).
       • Compute error at the output unit: $error = t_i - y_i$.
       • Propagate back error from output units to all the hidden units right until
         the input layer (backward error propagation step).
       • Update network weights using the gradient descent update equation:
           new weight = old weight – $\eta \times$gradient of the error with respect to the
         weight.
```

The error is backpropagated from a neuron n_2 in layer k to a neuron n_1 in layer *k − 1* in the ratio of the weight w_{12} on the edge between n_1 and n_2 to the weight on all the inputs to the neuron n_2.

There are multiple variants of the backpropagation algorithm. If the weight update is done after every instance, it is called *stochastic gradient descent*. Often times, batch-wise updates lead to quick convergence of the algorithm, where weights are updated after looking at a batch of instances. In such a case, the algorithm is called as *batch-wise gradient descent*.

The weights can be updated using a constant learning rate. However, if the learning rate is too small, it leads to slow convergence. If the learning rate is too large, it can lead to divergence rather than convergence. Hence, setting the learning rate is tricky. This has led to the development of various update methods (e.g., momentum, averaging, AdaGrad (Duchi et al. 2011), RMSProp (Hinton et al. 2012), Adam (Kingma and Ba 2014), and AdaDelta (Zeiler 2012)). Interested readers can read more about some of these update schedules in the paper by Rumelhart et al. (1986).

2.3 Practical Advice When Using ANNs and an Overview of Deep Learning Architectures

The fundamental difference between ANNs and other traditional classifiers is the following. For building traditional classifiers, a data scientist first needs to perform domain-specific feature engineering and then build models on top of featurized data. This needs domain knowledge, and a large amount of time is spent in coming up with innovative features that could help predict the class variable. In case of ANNs, the data scientist simply supplies the raw data to the ANN classifier. The hope is that the ANN can itself learn both the representation (features) and the weights too. This is very useful in hard-to-featurize domains like vision and speech. Multiple layers of a deep ANN capture different levels of data abstraction.

There are multiple hyper-parameters one has to tune for various deep learning architectures. The best way to tune them is by using validation data. But here are a few tips in using MLPs. The initial values for the weights of a hidden layer i could be uniformly sampled from a symmetric interval that depends on the activation function. For the *tanh activation* function, the interval could be $\left[-\sqrt{\left(\frac{6}{fan_{in}+fan_{out}}\right)}, \sqrt{\left(\frac{6}{fan_{in}+fan_{out}}\right)}\right]$ where fan_{in} is the number of units in the $(i-1)$-th layer and fan_{out} is the number of units in the i-th layer. For the *Sigmoid* function, the suggested interval is $\left[-4\sqrt{\left(\frac{6}{fan_{in}+fan_{out}}\right)}, 4\sqrt{\left(\frac{6}{fan_{in}+fan_{out}}\right)}\right]$. This initialization ensures that, early in training, each neuron operates in a regime of its activation function where information can easily be propagated both upward (activations flowing from inputs to outputs) and backward (gradients flowing from outputs to inputs).

How many hidden layers should one have? How many hidden units per layer? There is no right answer to this. One should start with one input, one hidden, and one output layers. Theoretically this can represent any function. Add additional layers only if the above does not work well. If we train for too long, possible overfitting can happen—the test/validation error increases. Hence, while training, use validation error to check for overfitting. Simpler models are better—try them first (Occam's razor).

Overview of Deep Learning Architectures

A large number of deep learning architectures have been proposed in the past few years. We will discuss just a few of these in this chapter. We mention a partial list of them below for the sake of completeness.

1. Deep supervised learning architectures: classification—multilayered perceptron (MLP); similarity/distance measure—DSSM, convolutional NN; sequence-to-sequence—recurrent neural net (RNN)/long short-term memory (LSTM); question answering and recommendation dialog—memory network (MemNN); reasoning in vector space—tensor product representation (TPR).

Table 17.1 Few popular libraries to build deep learning models

	Caffe	Torch	Theano	Tensorflow	CNTK
Language	C++	Lua	Python	Python	C++
Multi-GPU	Yes	Yes	Ok	Yes	Yes
Readability	Yes	Yes	Very poor	Very poor	Yes
Complex models	No	Yes	Yes	Yes	Yes
Visualization	No	Ok	No	Yes	Yes
Training	Windows/Linux	Linux only	Windows/Linux	Windows/Linux	Windows/Linux

2. Deep unsupervised learning: pre-training—denoising auto-encoder (DA) and stacked DA; energy-based models—restricted Boltzmann machines (RBM) and deep belief networks (DBN).
3. Deep reinforcement learning: an agent to play games, Deep Q-Network (DQN).

Training deep learning models is usually a compute-intensive task. Deep learning models work well when you have large amounts of data to train them. Hence, most people use graphics processing units (GPUs) to train good models. There are a few popular libraries to easily build deep learning models. Table 17.1 presents a comparison of these libraries.

2.4 Summary

ANN is a computational model inspired from the workings of the human brain. Although a perceptron can simply represent linear functions, multiple layers of perceptrons can represent arbitrary complex functions. The backpropagation algorithm can be used to learn the parameters in a multilayered feed-forward neural network. The various parameters of a feed-forward ANN such as learning rate, number of hidden layers, and initial weight vectors need to be carefully chosen. An ANN allows for learning of deep feature representations from raw training data.

2.5 An Example: MNIST Data

The following section explains how to build a simple MLP using the “mxnet” package in R for the MNIST handwritten digit recognition task. The MNIST data comprises of handwritten digits (60,000 in training dataset and 10,000 in test dataset) produced by different writers. The sample is represented by a 28 $\times$ 28 pixel map with each pixel having value between 0 and 255, both inclusive. You may refer to the MNIST data website[3] for more details. Here, we provide a sample of only 5000 digits (500 per digit) in the training sample and 1000 digits (100 per digit) in the test dataset. The task is to recognize the digit.

The main stages of the code below are as follows:

1. Download and perform data cleaning.
2. Visualize the few sample digits.
3. Specify the model.
   (a) Fully connected
   (b) Number of hidden layers (neurons)
   (c) Activation function type
4. Define the parameters and run the model.
   (a) "softmax" to normalize the output
   (b) X: Pixel data (X values)
   (c) Y: Dependent variable (Y values)
   (d) ctx: Processing device to be used
5. Predict the model output on test data.
6. Produce the classification (confusion) matrix and calculate accuracy.

Sample code "MLP on MNIST.R" and datasets "MNIST_train_sample.csv" and "MNIST_test_sample.csv" are available on the website.

3 Convolutional Neural Networks (CNNs)

In this section, we discuss a deep learning architecture called as convolutional neural networks. This architecture is mainly applied to image data. However, there have also been some use-cases where CNNs have been applied to embedding matrices for text data. In such cases, a text sequence is mapped onto a matrix where each word in the sequence is represented as a row using the word embedding for the word. Further, such an embedding matrix is treated very similar to an image matrix. We will first talk about ImageNet and various visual recognition problems. After that, we will discuss the technical details of a CNN.

3.1 ImageNet and Visual Recognition Problems

ImageNet[4] is an image dataset organized according to the WordNet (Miller 1995) hierarchy. Each meaningful concept in WordNet, possibly described by multiple

[3] http://yann.lecun.com/exdb/mnist/ (accessed on Jan 16, 2018).

[4] Imagenet dataset is hosted on http://image-net.org/ (accessed on Aug 1, 2018).

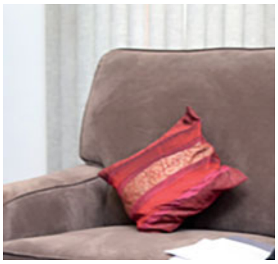
Fig. 17.2 A sample image

words or word phrases, is called a “synonym set” or “synset.” There are more than 100,000 synsets in WordNet, majority of them are nouns (80,000+). The ImageNet project is inspired by a growing sentiment in the image and vision research field—the need for more data. There are around 14,197,122 images labeled with 21,841 categories.

This dataset is used for the ImageNet Large Scale Visual Recognition Challenge held every year since 2010. The challenge runs for a variety of tasks including image classification/captioning, object localization, object detection, object detection from videos, scene classification, and scene parsing. The most popular task is image captioning.

The image classification task is as follows. For each image, competing algorithms produce a list of at most five object categories in the descending order of confidence. The quality of a labeling is evaluated based on the label that best matches the ground truth label for the image. The idea is to allow an algorithm to identify multiple objects in an image and not be penalized if one of the objects identified was in fact present but not included in the ground truth (labeled values). For example, for the image in Fig. 17.2, “red pillow” is a good label, but “flying kite” is a bad label. Also, “sofa” is a reasonable label, although it may not be present in the hand-curated ground truth label set.

Table 17.2 shows the winners for the past few years for this task. Notice that in 2010, the architecture was a typical feature engineering-based model. But since 2012 all the winning models have been deep learning-based models. The depth of these models has been increasing significantly as the error has been decreasing over time.

CNNs have been used to solve various kinds of vision-related problems including the image classification challenge. Such tasks include object detection, action classification, image captioning, pose estimation, image retrieval, image segmentation for self-driving cars, traffic sign detection, face recognition, video classification, whale recognition from ocean satellite images, and building maps automatically from satellite images.

Table 17.2 ImageNet challenge winning architectures (compiled by author)

Year	2010	2012	2014	2015	2016	2017
Model, Institution	Linear SVM, NEC-UIUC	AlexNet, SuperVision	Visual Geometry Group (VGG) Oxford, Googlenet	Resnet, MSRA	Trimps-Soushen, The Third Research Institute of the Ministry of Public Security, P.R. China	Squeeze-and-Excitation Networks, NUS-Qihoo_DPNs (CLS-LOC)
#layers	Not a neural network	7 layers	27 layers	152 layers	Ensemble of Inception-v3 (48 layers), Inception-v4 (∼114 layers), Residual Network (152 layers), Inception-ResNet-v2 (200+ layers), Wide Residual Network (∼16 layers)	Integrated SE blocks to stacked ResNet-152
Accuracy	28%	16%	7%	3.6%	2.99%	2.25%

3.2 Biological Inspiration for CNNs

Hubel and Wiesel (1962) made the following observations about the visual cortex system. Nearby cells in the cortex represented nearby regions in the visual field. Visual cortex contains a complex arrangement of cells. These cells are sensitive to small subregions of the visual field, called a receptive field. The subregions are tiled to cover the entire visual field and may overlap. These cells act as local filters over the input space and are well suited to exploit the strong spatially local correlation present in natural images. Additionally, two basic cell types have been identified. Simple cells respond maximally to specific edge-like patterns within their receptive field. Complex cells have larger receptive fields and are locally invariant to the exact position of the pattern.

The question is how to encode these biological observations into typical MLPs. Fukushima and Miyake (1982) proposed the neocognitron, which is a hierarchical, multilayered artificial neural network, and can be considered as the first CNN in some sense.

Besides the visual cortex system, in general, we tend to think in terms of hierarchy, for example, the vision hierarchy (pixels, edges, textons, motifs, parts, objects), the speech hierarchy (samples, spectral bands, formants, motifs, phones, words), and the text hierarchy (character, word, phrases, clauses, sentences, paragraphs, story). To encode this hierarchical behavior into a neural framework, we will study CNNs in this section.

Why cannot we rely on MLPs for image classification? Consider a simple task where you want to learn a classifier to detect images with dogs versus those without. In the popular CIFAR-10 image dataset, images are of size $32 \times 32 \times 3$ (32 wide, 32 high, 3 color channels) only, so a single fully connected neuron in a first hidden layer of a regular neural network would have $32 \times 32 \times 3 = 3072$ weights. A 200×200 image, however, would lead to neurons that have $200 \times 200 \times 3 = 120{,}000$ weights. Such network architecture does not take into account the spatial structure of data, treating input pixels which are far apart and close together on exactly the same footing. Clearly, the full connectivity of neurons is wasteful in the framework of image recognition, and the huge number of parameters quickly leads to overfitting. This motivates us to build specific architecture to deal with images, as discussed below.

3.3 Technical Details of a CNN

Figure 17.3 shows four kinds of layers that a typical CNN has: the convolution (CONV) layer, the rectified linear units (RELU) layer, the pooling (POOL) layers, and the fully connected (FC) layers. FC layers are the ones that we have seen so far in MLPs. In this section, we will discuss the other three layers (CONV, RELU, and POOL) in detail one by one.

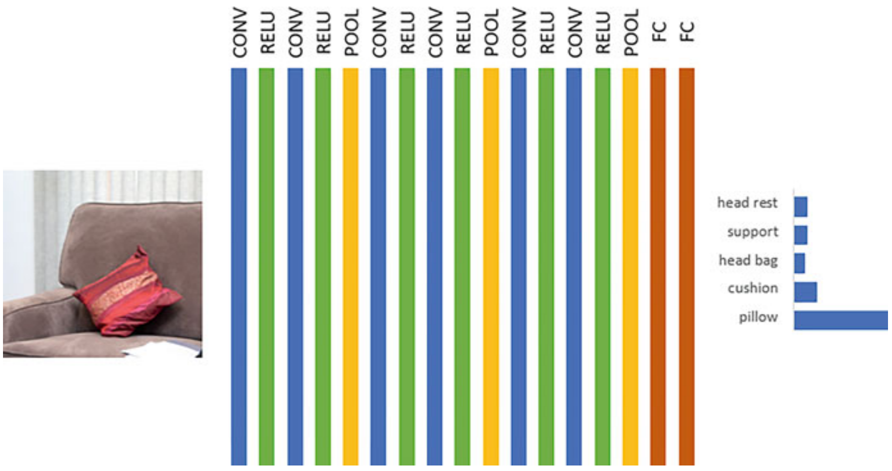

Fig. 17.3 ConvNet: CONV, RELU, POOL, and FC layers

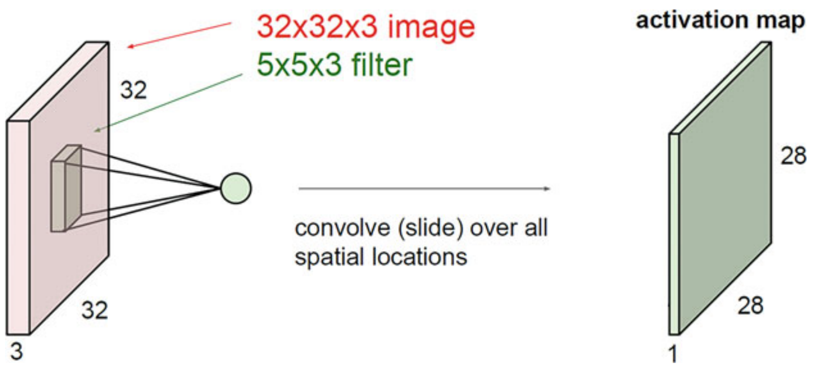

Fig. 17.4 Convolution layer (Source: CS231N Stanford course slides)

CONV Layer

Let us start by understanding the convolution layer. Given an original image, the convolution layer applies multiple filters on the image to obtain *feature maps*. Filters are rectangular in nature and always extend the full depth of the input volume. For example, in Fig. 17.4, the input image has a size of $32 \times 32 \times 3$, and a filter of size $5 \times 5 \times 3$ is being applied. To get the entire *feature map*, the filter is convolved with the image by sliding over the image spatially and computing the dot products. The sliding can be done one-step or multiple steps at a time; this is controlled using a parameter called the stride. Filters are like features defined over the input volume. Rather than just using one filter, we could use multiple filters. The final output volume depth depends on the number of filters used. For example, if we had six

$5 \times 5 \times 3$ filters, we will get six different activation maps each of size $28 \times 28 \times 1$[5] leading to an output volume size of $28 \times 28 \times 6$. Note that an activation map can also be seen as a 28×28 sheet of neuron outputs where each neuron is connected to a small region in the input, and all of them share parameters.

The elements of the filters are the weights that are learned using backpropagation during training. The convolution layer helps us implement two important concepts in a CNN:

1. **Sparse Connectivity**: Convolution layer enforces a local connectivity pattern between neurons of adjacent layers. The inputs of hidden units in layer *m* are from a subset of units in layer $m - 1$, units that have spatially contiguous receptive fields.
2. **Shared Weights**: In CNNs, each filter is replicated across the entire visual field. These replicated units share the same parameterization (weight vector and bias) and form a feature map. Gradient descent can still be used to learn such shared parameters, with only a small change to the original algorithm. The gradient of a shared weight is simply the sum of the gradients of the parameters being shared. Replicating units in this way allows for features to be detected regardless of their position in the visual field. Weight sharing increases learning efficiency by greatly reducing the number of free parameters being learnt.

Convolution can be done by sliding the filter across the entire space of the input volume with a stride of 1 or larger stride values. Larger stride values lead to small output volumes. Also, sometimes, the original input volume is padded with zeroes at the border to prevent the loss of information at the border. In general, it is common to see CONV layers with stride 1, filters of size $F \times F$, and zero padding with $(F - 1)/2$. For example, if a $32 \times 32 \times 3$ image is padded by two zeros all around, then the activation map size will be $((36 - 5)/1) + 1 = 32$. So now there is no loss of the information (at the borders) because the whole image is covered.

Due to weight sharing, the number of weights to be learned in a CONV layer is much lesser compared to the weight in a layer in an MLP.

RELU Layer

Next, we discuss about the RELU (rectified linear units) layer. This is a layer of neurons that applies the activation function $f(x) = max(0,x)$. It increases the nonlinear properties of the decision function and of the overall network without affecting the receptive fields of the convolution layer. Other functions are also used to increase nonlinearity, for example, the hyperbolic tangent $f(x) = tanh(x)$ and the sigmoid function. This layer clearly does not involve any weights to be learned.

POOL Layer

There are several nonlinear functions to implement pooling among which max pooling is the most common. It partitions the input image into a set of nonoverlapping

[5]Activation Map Size = ((image size − filter size)/stride) + 1. Here, Image size is 32. Filter Size is 5. Stride = 1. Activation Map size = $((32 - 5)/1) + 1$ which is equal to 28.

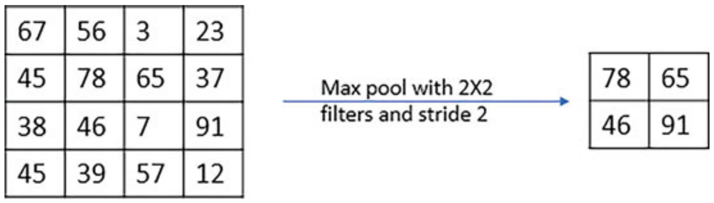

Fig. 17.5 Pooling example

rectangles and, for each such subregion, outputs the maximum. The intuition is that the exact location of a feature is less important than its rough location relative to other features. The pooling layer serves to progressively reduce the spatial size of the representation, to reduce the number of parameters and amount of computation in the network, and hence to also control overfitting. Figure 17.5 shows an example of max pooling with a pool size of 2 × 2.

Finally, after several convolutional and max pooling layers, the high-level reasoning in the neural network is done via fully connected layers. Neurons in a fully connected layer have connections to all activations in the previous layer, as seen in regular MLPs.

3.4 Summary

In summary, we have discussed an interesting deep learning architecture, CNNs, for images in this section. CNNs are very popular these days across a large variety of image processing tasks. Convolution networks are inspired by the hierarchical structure of the visual cortex. Things that differentiate CNNs from DNNs are sparse connectivity, shared weights, feature maps, and pooling.

3.5 An Example: MNIST Data (Similar to MLP Approach)

The main stages of the code are as follows:

1. Download and perform data cleaning.
2. Visualize few sample digits.
3. Specify the model:

   (a) First convolution layer and specifying kernel
   (b) Activation function type
   (c) Pooling layer and specifying the type of pooling (max or average)
   (d) Second convolution layer, activation function, and pooling layer
   (e) First fully connected and specifying the number of hidden layers (neurons)
   (f) Second fully connected
   (g) Applying softmax to normalize the output

4. Define the parameters and run the model:
    (a) lenet: pointer to the last computation node in the network definition
    (b) X: pixel data (X values)
    (c) Y: dependent variable (Y values)
    (d) ctx: processing device to be used
    (e) num.round: maximum number of iterations over the dataset
    (f) array.batch.size: batch size for batch-wise gradient descent
    (g) Learning rate
    (h) Momentum: for momentum based gradient descent updates
    (i) WD: weight decay
5. Predict the model output on test data.
6. Produce confusion matrix and calculate accuracy.

The sample code helps understand how to build a CNN using the "mxnet" R package. The code "Mxnet-MNIST_CNN.R" and the datasets "MNIST_train_sample.csv" and "MNIST_test_sample.csv" are available on the website.

4 Recurrent Neural Networks (RNNs)

In this section, we will discuss a deep learning architecture to handle sequence data, RNNs. We will first motivate why sequence learning models are needed. Then we will talk about technical details of RNNs (recurrent neural networks) and finally discuss about their application to image captioning and machine translation.

4.1 Motivation for Sequence Learning Models

Sequences are everywhere. Text is a sequence of characters. Speech is a sequence of phonemes. Videos are sequences of images. There are many important applications powered by analytics on top of sequence data. For example, machine translation is all about transforming a sequence written in one language to another. We need a way to model such sequence data using neural networks. Humans do not start their thinking from scratch every second. As you read this section, you understand each word based on your understanding of previous words. You do not throw everything away and start thinking from scratch again. Your thoughts have persistence. Thus, we need neural networks with some persistence while learning. In this chapter, we will discuss about RNNs as an architecture to support sequence learning tasks. RNNs have loops in them which allow for information to persist.

Language models are the earliest example of sequence learning for text sequences. A language model computes a probability for a sequence of words:

$P(w_1, \ldots, w_m)$. Language models are very useful for many tasks like the following: (1) *next word prediction*: for example, predicting the next word after the user has typed this part of the sentence. "Stocks plunged this morning, despite a cut in interest rates by the Federal Reserve, as Wall ..."; (2) *spell checkers*: for example, automatically detecting that minutes has been spelled incorrectly in the following sentence. "They are leaving in about fifteen minuets to go to her house"; (3) *mobile auto-correct*: for example, automatically suggesting that the user should use "find" instead of "fine" in the following sentence. "He is trying to fine out."; (4) *speech recognition*: for example, automatically figuring out that "popcorn" makes more sense than "unicorn" in the following sentence. "Theatre owners say unicorn sales have doubled..."; (5) *automated essay grading*; and (6) *machine translation*: for example, identifying the right word order as in *p(the cat is small) > p(small the is cat)*, or identifying the right word choice as in *p(walking home after school) > p(walking house after school).*

Traditional language models are learned by computing expressing probability of an entire sequence using the chain rule. For longer sequences, it helps to compute probability by conditioning on a window of n previous words. Thus, $P(w_1, \ldots, w_m) = \Pi_{i=1}^{m} P(w_i | w_1, \ldots w_{i-1}) \approx \Pi_{i=1}^{m} P\left(w_i | w_{i-(n-1)}, \ldots, w_{i-1}\right)$. Here, we condition on the previous n values instead of previous all values. This approximation is called the Markov assumption. To estimate probabilities, one may compute unigrams, bigrams, trigrams, etc., as follows, using a large text corpus with T tokens.

Unigram model: $p(w_1) = \frac{count(w_1)}{T}$
Bigram models: $p(w_2|w_1) = \frac{count(w_1, w_2)}{count(w_1)}$
Trigram models: $p(w_3|w_1, w_2) = \frac{count(w_1, w_2, w_3)}{count(w_1, w_2)}$

Performance of n-gram language models improves as n for n-grams increases. Smoothing, backoff, and interpolation are popular techniques to handle low frequency n-grams. But the problem is that there are a lot of n-grams, especially as n increases. This leads to gigantic RAM requirements. In some cases, the window of past consecutive n words may not be sufficient to capture the context. For instance, consider a case where an article discusses the history of Spain and France and somewhere later in the text, it reads, "The two countries went on a battle"; clearly the information presented in this sentence alone is not sufficient to identify the name of the two countries.

Can we use MLPs to model the next word prediction problem? Figure 17.6 shows a typical MLP for the next word prediction task as proposed by Bengio et al. (2003). The MLP is trained to predict the t-th word based on a fixed size context of previous $n - 1$ words. This network assumes that we have a mapping C from any word i in the vocabulary to a distributed feature vector like word2vec.[6] Thus, if m is the

[6]Word2vec is an algorithm for learning a word embedding from a text corpus. For further details, read Mikolov et al. (2013).

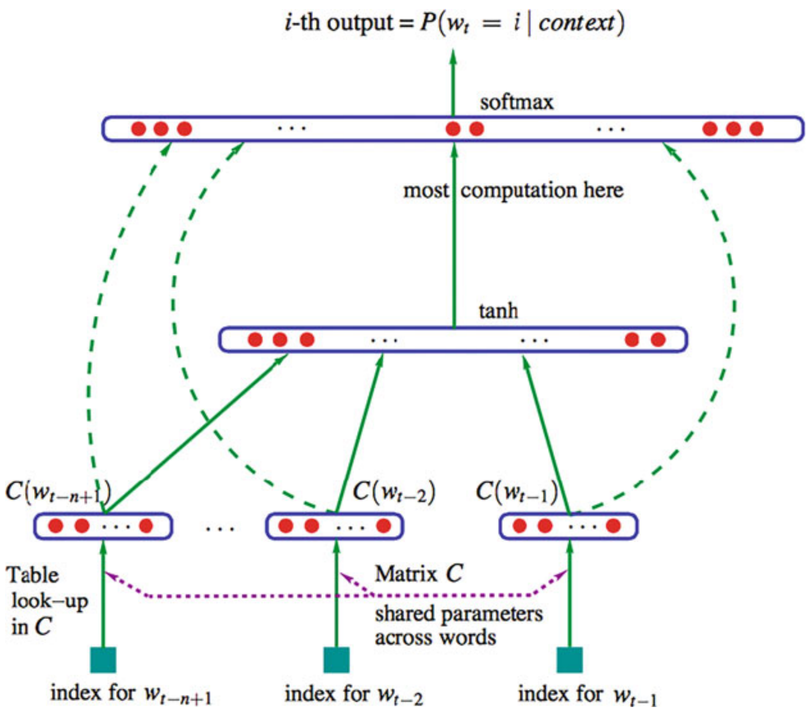

Fig. 17.6 MLP for next word prediction task (Source: Bengio et al. 2003)

dimension for the feature vector representation, and $|V|$ is vocabulary size, C is a $|V| \times m$ sized matrix. $C(w_{t-i})$ is the vector representation of the word that came i words ago. C could also be learned along with the other weights in the network. Further, the model contains a hidden layer with a nonlinearity. Finally, at the output layer, a softmax is performed to return the probability distribution of size $|V|$ which is expected to be as close as possible to the one-hot encoded representation of the actual next word.

In all conventional language models, the memory requirements of the system grow exponentially with the window size n making it nearly impossible to model large word windows without running out of memory. But in this model, the RAM requirements grow linearly with n. Thus, this model supports a fixed window of context (i.e., n). There are two drawbacks of this model: (1) the number of parameters increase linearly with the context size, and (2) it cannot handle contexts of different lengths. RNNs help address these drawbacks.

4.2 Technical Details of RNNs

RNNs is a deep learning neural architecture that can support next word prediction with variable n. RNNs tie the weights at each time step. This helps in conditioning

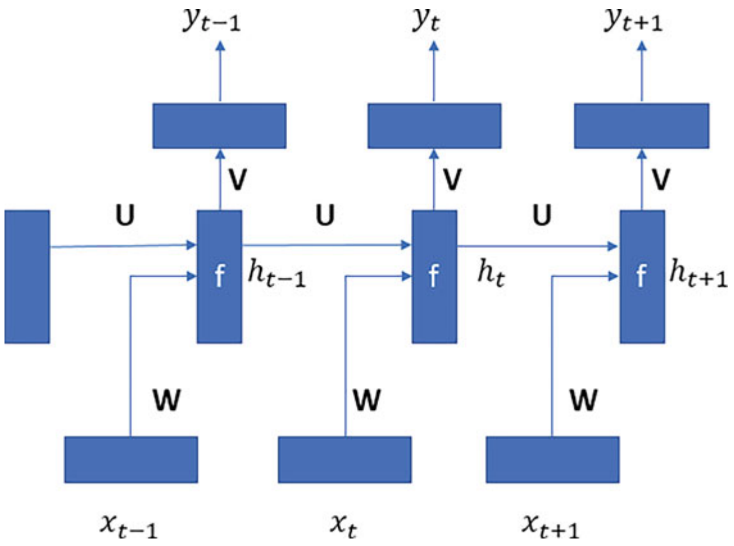

Fig. 17.7 Basic RNN architecture

the neural network on all previous words. Thus, the RAM requirement only scales with the number of words in the vocabulary. Figure 17.7 shows the architecture of a basic RNN model with three units. **U**, **V**, and **W** are the shared weight matrices that repeat across multiple time units. Overall the parameters to be learned are **U**, **V**, and **W**.

RNNs are called recurrent because they perform the same task for every element of a sequence. The only thing that differs is the input at each time step. Output is dependent on previous computations. RNNs can be seen as neural networks having "memory" about what has been calculated so far. The information (or the state) h_t at any time instance t is this memory. In some sense, h_t captures a thought that summarizes the words seen so far. RNNs process a sequence of vectors x by applying a recurrence formula at every time step: $h_t = f_{U,W}(h_{t-1}, x_t)$, where h_t is the new state, $f_{U,W}$ is some function with parameters U and W, h_{t-1} is the old state, and x_t is the input vector at current time step. Notice that the same function and the same set of parameters are used at every time step.

The weights for an RNN are learned using the same backpropagation algorithm, also called as backpropagation through time (BPTT) in the context of RNNs. The training data for BPTT should be an ordered sequence of input-output pairs $\langle x_0, y_0\rangle, \langle x_1, y_1\rangle, \ldots, \langle x_{n-1}, y_{n-1}\rangle$. An initial value must be specified for the hidden layer output h_0 at time t_0 . Typically, a vector of all zeros is used for this purpose. BPTT begins by unfolding a recurrent neural network through time. When the network is unfolded through time, the unfolded network contains k instances of *a unit, each containing an input, a hidden layer, and an output*. Training then proceeds in a manner similar to training a feed-forward neural network with backpropagation, except that each epoch must run through the observations, y_t, in sequential order. Each training pattern consists of $\langle h_t, x_t, x_{t+1}, x_{t+2}, \ldots, x_{t+k-1}, y_{t+k} \rangle$. Typically, backpropagation is applied in an online manner to update the weights as each training pattern is presented. After each pattern is presented, and the weights have been updated, the weights in each instance of $\boldsymbol{U}$, $\boldsymbol{V}$, *and* $\boldsymbol{W}$ are averaged together so that they all have the same weights, respectively. Also, h_{t+1} is calculated as $h_{t+1} = f_{U,W}(h_t, x_{t+1})$, which provides the information necessary so that the algorithm can move on to the next time step, $t + 1$. The output y_t is computed as follows: $y_t = \mathit{softmax}(\boldsymbol{V}\ h_t)$. Usually the cross entropy loss function is used for the optimization: Given an actual output distribution y_t and a predicted output distribution $\widehat{y_t}$, cross entropy loss is defined as $-\sum_{j=1}^{|V|} y_{t,j} \log \widehat{y}_{(t,j)}$. Note that y_t is the true vector; it could be a one-hot encoding of the expected word or a word2vec representation of the expected word at the t-th time instant.

4.3 Example: Next Word Prediction

The following pseudo-code shows how to build an RNN using the “mxnet” R package for the next word prediction task. Below are the main code stages:

1. Download the data and perform cleaning.
2. Create Word 2 Vector, dictionary, and lookup dictionary.
3. Create multiple buckets for training data.
4. Create iterators for multiple buckets data.
5. Train the model for multiple bucket data with the following parameters:
   (a) Cell_type = “lstm” #Using lstm cell which can hold the results
   (b) num_rnn_layer = 1
   (c) num_embed = 2
   (d) num_hidden = 4 #Number of hidden layers
   (e) loss_output = “softmax”
   (f) num.round = 6
6. Predict the output of the model on “Test” data.
7. Calculate the accuracy of the model.

The sample code helps understand how to build an RNN using the “mxnet” R package. The code “Next_word_RNN.R” and the datasets “corpus_bucketed_train.rds” and “corpus_bucketed_test.rds” are available on the website.

The basic RNN architecture can be extended in many ways. Bidirectional RNNs are RNNs with two hidden vectors per unit. The first hidden vector maintains the state of the information seen so far in the sequence in the forward direction, while the other hidden vector maintains the state representing information seen so far in the sequence in the backward direction. The number of parameters in bidirectional RNNs is thus twice the number of parameters in the basic RNN.

RNNs could also be deep. Thus, a deep RNN has stacked hidden units, and the output neurons are connected to the most abstract layer.

4.4 Applications of RNNs: Image Captioning and Machine Translation

Recurrent networks offer a lot of flexibility. Thus, they can be used for a large variety of sequence learning tasks. Such tasks could be classified as one-to-many, many-to-one, or many-to-many depending on the number of inputs and the number of outputs. An example of a one-to-many application is image captioning (image $\rightarrow$ sequence of words). An example of many-to-one application is sentiment classification (sequence of words $\rightarrow$ sentiment). An example of "delayed" many-to-many application is machine translation (sequence of words $\rightarrow$ sequence of words). Finally, an example of the "synchronized" many-to-many case is video classification on frame level.

In the following, we will discuss two applications of RNNs: image captioning and machine translation. Figure 17.8 shows the neural CNN-RNN architecture for the image captioning task. First a CNN is used to obtain a deep representation for the image. The representation is then passed on to the RNN to learn captions. Note that the captions start with a special word START and end with a special word END. Unlike image classification task where the number of captions is limited, in image captioning, the number of captions that can be generated are many more since rather than selecting one of say 1,000 captions, here the task is to generate captions.

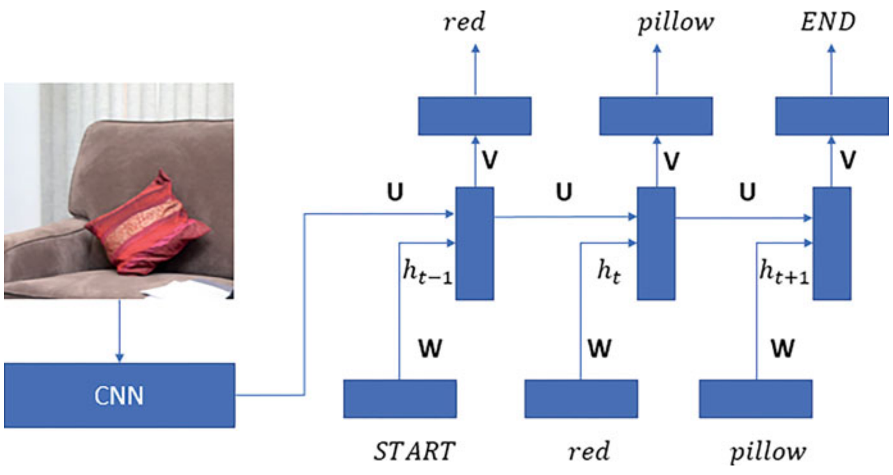

Fig. 17.8 CNN-RNNs for image captioning task

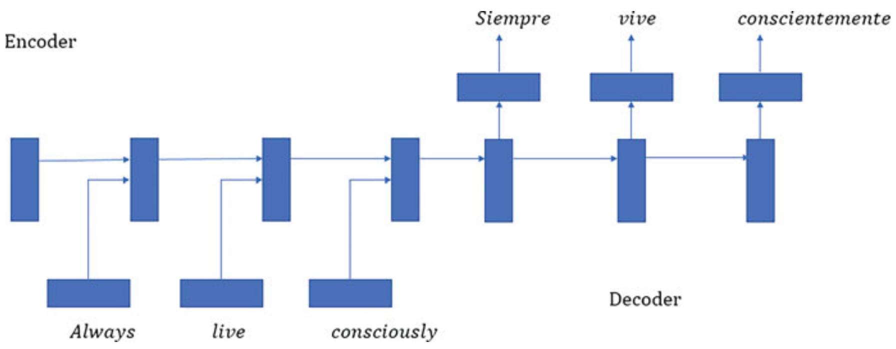

Fig. 17.9 RNNs for machine translation

As shown in Figure 17.8, a CNN trained on the ImageNet data is first used. Such a CNN was discussed in Sect. 3. The last fully connected layer of the CNN is thrown away, and the result from the CNN's penultimate layer is fed to the first unit of the RNN. One-hot encoding of the special word START is fed as input to the first unit of the RNN. At the training time, since the actual image captions are known, the corresponding word representations are actually fed as input for every recurrent unit. However, at test time, the true caption is unknown. Hence, at test time, the output of the k-th unit is fed as input to the $(k + 1)$-th unit. This is done for better learning of the order of words in the caption. Cross entropy loss is used to compute error at each of the output neurons. Microsoft COCO[7] is a popular dataset which can be used for training such a model for image captioning. The dataset has about 120K images each with five sentences of captions (Lin et al. 2014).

Lastly let us discuss about application of RNNs to machine translation. Figure 17.9 shows a basic encoder–decoder architecture for the machine translation task using RNNs. The encoder RNN tries to encode all the information from the source language into a single hidden vector at the end. Let us call this last hidden vector of the encoder as the "thought" vector. The decoder RNN uses information from this thought vector to generate words in the target language. The architecture tries to minimize the *cross entropy error* for all target words conditioned on the source words.

There are many variants of this architecture as follows. (1) The encoder and the decoder could use shared weights or different weights. (2) Hidden state in the decoder always depends on the hidden state of the previous unit, but it could also optionally depend on the thought vector and predicted output from the previous unit. (3) Deep bidirectional RNNs could be used for both encoder and decoder.

Beyond these applications, RNNs have been used for many sequence learning tasks. However, RNNs suffer from vanishing gradients problem. In theory, RNN

[7]Microsoft COCO dataset http://www.mscoco.org/ (accessed on Aug 1, 2018) or http://cocodataset.org/ (accessed on Aug 1, 2018).

can memorize in hidden state, that is, h_t, all the information about past inputs. But, in practice, standard RNN cannot capture very long-distance dependency. Vanishing/exploding gradient problem in backpropagation: gradient signal can end up being multiplied a large number of times (as many as the number of time steps) by the weight matrix associated with the connections between the neurons of the recurrent hidden layer. If the weights in transition weight matrix are small (or, more formally, if the leading eigenvalue of the weight matrix is smaller than 1.0), it can lead to vanishing gradients where the gradient signal gets so small that learning either becomes very slow or stops working altogether. It can also make more difficult the task of learning long-term dependencies in the data. Conversely, if the weights in this matrix are large (or, again, more formally, if the leading eigenvalue of the weight matrix is larger than 1.0), it can lead to a situation where the gradient signal is so large that it can cause learning to diverge. This is often referred to as exploding gradients. A solution to this problem is long short-term memory (LSTM) which are deep learning architectures similar to RNNs but with explicit memory cells. The main idea is to keep around memories to capture long-range dependencies and to allow error messages to flow at different strengths depending on the inputs. The intuition is that memory cells can keep information intact, unless inputs make them forget it or overwrite it with new input. The memory cell can decide to output this information or just store it. The reader may refer to Hochreiter and Schmidhuber (1997) for further details about LSTMs.

4.5 Summary

In summary, recurrent neural networks are powerful in modeling sequence data. But they have the vanishing/exploding gradient problem. LSTMs are better since they avoid the vanishing/exploding gradient problem by introducing memory cells. Overall, RNNs and LSTMs are really useful in many real-world applications like image captioning, opinion mining, and machine translation.

5 Further Reading

The advances being made in this field are continuous in nature due to the practice of sharing information as well as cooperating with researchers working in labs and in the field. Therefore, the most recent information is available on the Web and through conferences and workshops. The book[8] by Goodfellow et al. (2016), the

[8]https://www.deeplearningbook.org/ (accessed on Aug 1, 2018).

deep learning tutorials,[9] and specialization in deep learning[10] offered by Andrew Ng are good starting points for learning more.

Additional reference material (accessed on Aug 1, 2018):

- Good Introductory Tutorial: http://web.iitd.ac.in/∼ sumeet/Jain.pdf
- A Brief Introduction to Neural Networks: http://www.dkriesel.com/en/science/neural_networks

CNN feature visualization:

- http://people.csail.mit.edu/torralba/research/drawCNN/drawNet.html?path=imagenetCNN
- http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns/
- http://karpathy.github.io/2015/05/21/rnn-effectiveness/
- http://colah.github.io/posts/2015-08-Understanding-LSTMs/
- Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, *9*(8), 1735–1780. http://www.bioinf.jku.at/publications/older/2604.pdf
- http://jeffdonahue.com/lrcn/
- https://github.com/kjw0612/awesome-rnn#theory

Electronic Supplementary Material

All the datasets, code, and other material referred in this section are available in www.allaboutanalytics.net.

- Data 17.1: MNIST_train_sample.csv
- Data 17.2: MNIST_test_sample.csv
- Data 17.3: corpus_bucketed_test.rds
- Data 17.4: corpus_bucketed_train.rds
- Code 17.1: MLP_MNIST.R
- Code 17.2: MXNET_MNIST_CNN.R
- Code 17.3: Next_word_RNN.R

Exercises

Ex. 17.1 Which of these is false?

(a) Deep learning needs large amounts of data for learning.

[9]http://deeplearning.net/reading-list/tutorials/ (accessed on Aug 1, 2018).
[10]https://www.coursera.org/specializations/deep-learning (accessed on Aug 1, 2018).

(b) Deep learning classifiers are all linear in nature.
(c) Deep learning needs a lot of compute power.
(d) Deep learning consists of multiple model architectures.

Ex. 17.2 What is the algorithm used to train a single neuron?

(a) Backpropagation
(b) Forward propagation
(c) Perceptron
(d) None of the above

Ex. 17.3 How can you make an artificial neuron learn nonlinear patterns?

(a) Change integration function to be nonlinear
(b) Use multilayered perceptrons
(c) Both of the above
(d) None of the above

Ex. 17.4 What is the weight update equation in perceptron?

(a) New w = old w + (learning rate) × (error) × (instance vector)
(b) New w = old w - (learning rate) × (error) × (instance vector)
(c) New w = (learning rate) × (error) × (instance vector)
(d) New w = (learning rate) × (error) × (instance vector)

Ex. 17.5 If an MLP has an input layer with 10 features, hidden layer with 20 neurons, and output layer with 1 output, how many parameters are there?

(a) $10 \times 20 + 20 \times 1$
(b) $(10 + 1) \times 20 + (20 + 1) \times 1$
(c) $(10 - 1) \times 20 + (20 - 1) \times 1$
(d) 10×20

Ex. 17.6 Why cannot the perceptron algorithm work for MLPs?

(a) We never discussed this in the class!
(b) MLPs have too many parameters, and perceptron is not very efficient when there are too many parameters.
(c) Supervision is not available for neurons in the hidden layers of an MLP.
(d) Perceptrons are meant to learn only linear classifiers, while MLPs can learn more complex boundaries.

Ex. 17.7 We discussed three different activation functions. Which of the following is not an activation function?

(a) Step function
(b) Spherical function
(c) Ramp function
(d) Sigmoid function

Ex. 17.8 What is false among the following?

(a) MLPs have fully connected layers, while CNNs have sparse connectivity.
(b) MLPs are supervised, while CNNs are usually used for unsupervised algorithms.
(c) MLPs have more weights, while CNNs have fewer number of weights to be learned.
(d) MLP is a general modeling architecture, while CNNs specialize for images.

Ex. 17.9 Given an image of $32 \times 32 \times 3$, a single fully connected neuron will have how many weights to be learned?

(a) $32 \times 32 \times 3 + 1$
(b) 32
(c) 3
(d) 32×32

Ex. 17.10 What is the convolution operation closest to?

(a) Jaccard similarity
(b) Cosine similarity
(c) Dot product
(d) Earth mover's distance

Ex. 17.11 How many weights are needed if the input layer has 32×32 inputs and the hidden layer has 20×20 neurons?

(a) $(32 \times 32 + 1) \times 20 \times 20$
(b) $(20 + 1) \times 20$
(c) $(32 + 1) \times 20$
(d) $(32 + 1) \times 32$

Ex. 17.12 Consider a volume of size $32 \times 32 \times 3$. If max pooling is applied to it with pool size of 4 and stride of 4, what are the number of weights in the pooling layer?

(a) $(32 \times 32 \times 3 + 1) \times (4 \times 4)$
(b) $4 \times 4 + 1$
(c) 0
(d) $32 \times 32 \times 3$

Ex. 17.13 Which among the following is false about the differences between MLPs and RNNs?

(a) MLPs can be used with fixed-sized sequences, while RNNs can handle variable-sized sequences.
(b) MLPs have more weights, while RNNs have fewer number of weights to be learned.
(c) MLP is a general modeling architecture, while RNNs specialize for sequences.
(d) MLPs are supervised, while RNNs are usually used for unsupervised algorithms.

Ex. 17.14 We looked at two neural models for next word prediction: an MLP and an RNN. Given a vocabulary of 1000 words, and a hidden layer of size 100, a context of size 6 words, what are the number of weights in an MLP?

(a) $(6 \times 1000 + 1) \times 100 + (100 + 1) \times 1000$
(b) $(1000 + 1) \times 100 + (100 + 1) \times 100 + (100 + 1) \times 1000$
(c) $(6 \times 6 + 1) \times 100 + (6 \times 6 + 1) \times 1000$
(d) $(1000 + 1) \times (100 + 1) \times 6$

Ex. 17.15 How does backpropagation through time differ from typical backpropagation in MLPs?

(a) Weights on edges supposed to have shared weights must be averaged out and set to the average after every iteration.
(b) Backpropagation in MLPs uses gradient descent, while backpropagation through time uses time series modeling.
(c) Backpropagation in MLPs has two iterations for every corresponding iteration in backpropagation through time.
(d) None of the above.

Answer in Length

Ex. 17.16 Define deep learning bringing out its five important aspects.

Ex. 17.17 Describe the backpropagation algorithm.

Ex. 17.18 RNNs need input at each time step. For image captioning, we looked at a CNN-RNN architecture.

(a) What is the input to the first hidden layer of the RNN?
(b) Where do the other inputs come from?
(c) How is the length of the caption decided?
(d) Does it generate new captions by itself or only select from those that it had seen in training data?
(e) If vocab size is V, hidden layer size is h, and average sequence size is "s," how many weights are involved in an RNN?

Hands-On Exercises

Ex. 17.19 Create a simple logistic regression-based classifier for the popular iris dataset in mxnet.

Ex. 17.20 Create an MLP classifier using three hidden layers of sizes 5, 10, 5 for the MNIST digit recognition task using mxnet. (Hint: Modify the code from Sect. 2.5 appropriately).

Ex. 17.21 Create a CNN classifier using two CONV layers each with twenty 5×5 filters with padding as 2 and stride as 1. Also use pooling layers with 2×2 filters with stride as 2. Do this for the MNIST digit recognition task using mxnet. (Hint: Modify the code from Sect. 3.5 appropriately).

Ex. 17.22 Train an RNN model in mxnet for the next word prediction task. Use a suitable text corpus from https://en.wikipedia.org/wiki/List_of_text_corpora. (Hint: Modify the code from Sect. 4.2 appropriately).

References

Bengio, Y., Ducharme, R., Vincent, P., & Jauvin, C. (2003). A neural probabilistic language model. *Journal of Machine Learning Research, 3*, 1137–1155.

Duchi, J., Hazan, E., & Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research, 12*, 2121–2159.

Fukushima, K., & Miyake, S. (1982). Neocognitron: A self-organizing neural network model for a mechanism of visual pattern recognition. In S. Amari & A. Michael (Eds.), *Competition and cooperation in neural nets* (pp. 267–285). Berlin: Springer.

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning* (Vol. 1). Cambridge: MIT Press.

Hinton, G., Srivastava, N., & Swersky, K. (2012). Lecture 6d—A separate, adaptive learning rate for each connection. *Slides of lecture neural networks for machine learning*. Retrieved Mar 6, 2019, from https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural Computation, 9*(8), 1735–1780.

Hubel, D. H., & Wiesel, T. N. (1962). Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. *The Journal of Physiology, 160*(1), 106–154.

Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. arXiv:1412.6980.

Lin, T. Y., Maire, M., Belongie, S., Hays, J., Perona, P., Ramanan, D., et al. (2014). Microsoft coco: Common objects in context. In D. Fleet, T. Pajdla, B. Schiele, & T. Tuytelaars (Eds.), *European conference on computer vision* (pp. 740–755). Cham: Springer.

Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Efficient estimation of word representations in vector space. arXiv:1301.3781

Miller, G. A. (1995). WordNet: A lexical database for English. *Communications of the ACM, 38*(11), 39–41.

Minsky, M. L., & Papert, S. (1969). *Perceptrons*. MIT Press, Cambridge, MA.

Rosenblatt, F. (1962). *Principles of neurodynamics*. Wuhan: Scientific Research Publishing.

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning internal representations by error propagation. In D. Rumelhart & J. McClelland (Eds.), *Parallel distributed processing. Explorations in the microstructure of cognition* (Vol. 1, pp. 318–362). Cambridge, MA: Bradford Books.

Zeiler, M. D. (2012). ADADELTA: An adaptive learning rate method. arXiv:1212.5701.