Chapter 3
Deep Learning

3.1 Introduction

Representation is the most fundamental issue in network analysis. More generically it affects the performance of any machine learning system. For example *weight of objects* alone is adequate to classify objects into *lighter* and *heavier* classes. Similarly, *height* is adequate to discriminate objects into *tall* and *short* classes. Such choices are simple in real life and are often based on commonsense.

However, in most of the practical applications, it is not possible to come out with a representation of the data so easily. However, a good representation is essential for successful machine learning. This may be attributed to the raise in the usage of *deep learning* systems. A routine way of appreciating deep learning is that the underlying learning system is realized using a cascade of systems that successively process data and pass on the information to subsequent levels; the size of the cascade is an indication of the *depth* of the learning system.

A hallmark of a deep learning systems is:

- **Representation Learning:**
  Can the system learn the representation automatically from the given data?
  In order to answer this question, we need to pose additional questions like:

  1. What is the size of the data required to learn the *right representation* automatically?
  2. What is the type of data that can be processed?
  3. Is it required to scale/normalize the data?
  4. Will the performance be affected by the order in which the data is processed?
  5. Is the model learnt for one application generic enough to be used in other applications?

Even though it is possible for a variety of realizations to answer one or more of these questions convincingly, it is the *artificial neural network* based systems that are

M. Aggarwal and M. N. Murty, *Machine Learning in Social Networks*,
SpringerBriefs in Computational Intelligence,
https://doi.org/10.1007/978-981-33-4022-0_3

shown to answer most of these questions. So, they are the most popular and perhaps only systems available for deep learning currently. So, it is convenient to view deep learning and deep neural networks as synonymous; we take this stand in rest of the chapter.

3.2 Neural Networks

Artificial neural networks ($ANNs$) are used by default for deep learning. Before we go into an exposition of deep neural networks, we will examine the basic building blocks that are essential in understanding the functionality of deep neural networks in this section. Historically, there were several developments in the early days but one of the simplest and important milestones was perceptron. We examine it and then consider more deeper architectures.

3.2.1 Perceptron

The working of Perceptron may be explained using Fig. 3.1

- Let X be an l-dimensional vector corresponding to a train or a test pattern. Such a pattern is given by $X = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_l \end{pmatrix}$.
- Let $\phi_i(X),\ i = 1, \ldots, d$ be features extracted from X. So, $\phi_i s$ are mappings from $\Re^l$ to $\Re$. For example, if X is a 2-dimensional vector given by $X = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$, then $\phi_1(X) = x_1$, $\phi_2(X) = x_2$, and $\phi_3(X) = x_1 x_2$ is a possible set of 3 features. Here, $l = 2$ and $d = 3$.

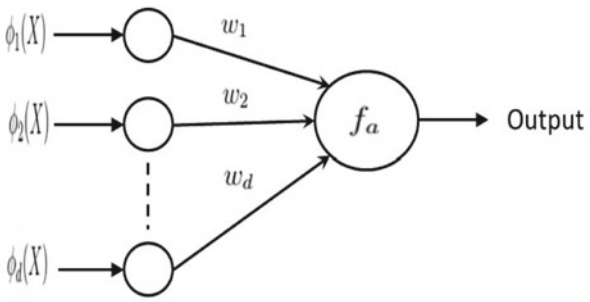

Fig. 3.1 Perceptron in the feature space

- The *weights* $w_1, w_2, \ldots, w_d$ indicate the importance of $\phi_1(X), \phi_2(X), \ldots, \phi_d(X)$ respectively. We call the perceptron using such a generic representation as a *perceptron in the feature space*
- The unit indicated by f_a is called the *activation unit* where f_a is the *activation function*. The sum of weighted features given by $\sum_{i=1}^{d} {}_i\phi_i(X)$ is the input to f_a; f_a is a function from $\Re$ to $\Re$. A simple example of f_a is

$$f_a(\alpha) = \begin{cases} 1 & if \quad \alpha > 0 \\ 0 & otherwise \end{cases} \tag{3.1}$$

  Such an activation function f_a is called a *Linear Threshold Function*.
- Note that $Output = f_a(\sum_{i=1}^{d} w_i\phi_i(X))$ which in general is a *nonlinear function* of the weighted sum $\sum_{i=1}^{d} w_i\phi_i(X))$.
- Let us consider a simple example to illustrate its working:
  Example: Consider the following five 2-dimensional patterns.
  Negative Class: $\begin{pmatrix}1\\2\end{pmatrix}, \begin{pmatrix}2\\3\end{pmatrix}$
  Positive Class: $\begin{pmatrix}3\\1\end{pmatrix}, \begin{pmatrix}4\\1\end{pmatrix}, \begin{pmatrix}4\\2\end{pmatrix}$
  - Note that we have $l = 2$ in this example. Further let us assume that $\phi_1(X) = x_1$ and $\phi_2(X) = x_2$. So, in this case $l = d = 2$ and the input features are the features used.
  - Using some algorithm suppose we *learn the weights* to be $w_1 = 1$ and $w_2 = -1$. So the weighted sum is given by $\sum_{i=1}^{2} w_i\phi_i(X) = x_1 - x_2$.
  - If we use the linear threshold function f_a on the weighted sum, we make the following decision:
    for a pattern $X = \begin{pmatrix}x_1\\x_2\end{pmatrix}$, if $x_1 - x_2 > 0$, then classify X as a positive class pattern and if $x_1 - x_2 < 0$ then assign X to the negative class.
  - Note that for pattern $\begin{pmatrix}1\\2\end{pmatrix}$, $x_1 - x_2 = -1 < 0$; So, it is classified as a member of the negative class. Similarly, for $\begin{pmatrix}3\\1\end{pmatrix}$, $x_1 - x_2 = 2 > 0$; So, it is assigned to the positive class. Further, $x_1 - x_2 = 0$ characterizes the boundary to decide between the two classes.
  - We depict the example using Fig. 3.2. In the figure, the two patterns of the negative class and the three patterns of the positive class are shown. Further, the dotted line $x_1 = x_2$ or equivalently $x_1 - x_2 = 0$ is the *decision boundary* between the two classes characterized by $w_1 = 1$ *and* $w_2 = -1$.
  - There is another line, a broken line, which is parallel to the earlier line and is described by $x_1 = x_2 + 1$. Note that even this line also is a decision boundary between the two classes. It is possible to see that if there exists one decision boundary, there can be infinite decision boundaries between the two classes.

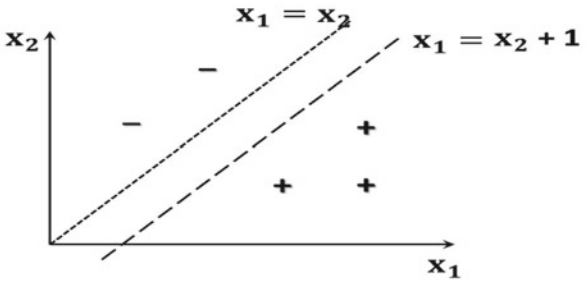

Fig. 3.2 Decision boundaries

  - The decision boundary $x_1 = x_2 + 1$ or equivalently $x_1 - x_2 - 1 = 0$ may be explained by a different choice of $\phi_i s$ and $w_i s$. If we define $\phi_0(X) = 1$ and $w_0 = -1$ and retain the earlier predicates and weights, then the decision boundary is described by $\sum_{i=0}^{2} w_i \phi_i(X) = -1 + x_1 - x_2 = 0$. This is a more generic form that relaxes the constraint that the decision boundary goes through the origin.

- In the two-dimensional example we have considered the decision boundary to be a line and its generic form is $\sum_{i=0}^{2} w_i \phi_i = 0$. These ideas can be extended to deal with binary classification (two-class) problems in any l dimensions by using $\sum_{i=0}^{d} w_i \phi_i$ as the weighted sum or input to the activation function f_a. In such a case, the decision boundary is a hyperplane.
- A popular choice for the features is $\phi_0(X) = 1$ and $\phi_i(X) = x_i$ for $i = 1, 2, \ldots, l$. The advantage of this representation is that it requires $d + 1$ features (including ϕ_0) where $d = l$. So, the complexity is linear in the input dimension l. Let us call a perceptron using such a representation as a *perceptron in the input space* However, the resulting decision boundary cannot deal with two classes of patterns that are not linearly separable.
- An important point is that by considering a larger value of d, it is possible to deal with nonlinear classification problems. Specifically, when the patterns are l-bit binary strings, it is possible to represent any boolean function on l bits using all possible subsets (minterms of different sizes) as features.
- For example, the function $odd - parity(x_1, x_2, x_3)$ returns a 1 if $x_1 + x_2 + x_3$ is odd and returns a value 0 otherwise. It is not linear in terms of the four features $\phi_0(X) = 1$ and $\phi_i(X) = x_i$ for $i = 1, 2, 3$ where X is a 3-bit binary pattern. However by using additional features, or minterms, it is possible to represent the odd-parity function using the form $x_1 + x_2 + x_3 - 2(x_1x_2 + x_1x_3 + x_2x_3) + 4x_1x_2x_3$.. Note that such a representation involves features that are nonlinear in $x_1, x_2, \ and \ x_3$ like x_1x_2, x_1x_3, x_2x_3, and $x_1x_2x_3$.
- Any vector can be represented as a binary string of some length l on a boolean computer. So, any classification problem based on training patterns can be dealt with a *perceptron in the feature space* that employs 2^l features. However, in most of the practical problems the value of l, the number of bits could be very large and training a perceptron using 2^l features could be *computationally prohibitive.* That is the reason for employing a *perceptron in the input space*.

3.2.2 Characteristics of Neural Networks

Some of the important characteristics of ANNs which are related to the discussion so far are:

- They may be viewed as linear classifiers. They can handle even non-linear classification problems using an appropriate representation.
- If the classes are linearly separable in the input (l-dimensional) space, then the learning algorithms behind units like *perceptron in the input space* can find one out of the infinite possible linear decision boundaries in the $(l+1)$-dimensional space. Popularly, perceptron in the input space is called perceptron and henceforth we too will follow this popular terminology.
- There are other linear Classifiers including the ones based on support vector machines (SVMs). An SVM constrains the search space for the decision boundary by specifying an appropriate objective function. It is possible to view an SVM also as an ANN.
- It is possible to choose an appropriate activation function to realize the associated/pre-specified nonlinearity.
- The notion of weighted sum that is used as the input of an activation function naturally imposes a constraint on the type of data that can be processed. ANNs are intrinsically capable of processing only numeric data unlike some other classifiers including the ones based on decision trees and Bayes decision theory.
- Even though some of the ANNs including SVMs normalize the data as a processing step, in theory *normalization* is not required in using ANNs. For example, if a component $\phi_i(X)$ is more important than another component $\phi_j(X)$, then the associated weight w_i can be chosen to be larger than w_j.

3.2.3 Multilayer Perceptron Networks

A perceptron cannot handle classes that are not linearly separable. Further, to get the right representation is difficult. A Multilayer perceptron (MLP) is a feedforward network that combines multiple layers, where each layer may have multiple perceptrons. A major advantage of such a network is that it has the potential to learn the required representation from the input data. As an example, consider the exclusive or (XOR) function given in Table 3.1:

Table 3.1 is the truth table of the boolean function exclusive or (XOR). The output is 1 when exactly one of the inputs is 1, but not both. The first three columns characterize the truth table. The fourth and fifth columns in the table show equivalent representations of the XOR function.

Table 3.1 Exclusive or representations

x_1	x_2	$x_1 \oplus x_2$	$x_1 + x_2 - 2x_1x_2$	$\overline{x_1}x_2 + x_1\overline{x_2}$
0	0	0	0	0
0	1	1	1	1
1	0	1	1	1
1	1	0	0	0

A perceptron cannot represent it in terms of inputs x_1 and x_2 alone as it is not a linear function in these inputs. However the fourth column suggests that if we use an additional feature x_1x_2 then it can be realized. Similarly the fifth column gives an equivalent representation using the features $\overline{x_1}x_2$ and $x_1\overline{x_2}$. These two representations can be represented using the following MLPs.

1. Representing XOR as $x_1 + x_2 - 2x_1x_2$: The corresponding MLP is shown in Fig. 3.3.

   - Note that in the figure there are three layers.
   - The *input layer* receives the inputs x_1 and x_2; note that each pattern is a two-dimensional vector here. The input layer is indicated by the presence of small circles.
   - There are two additional layers. In the output there is a perceptron whose output is the exclusive or of x_1 and x_2 shown as $x_1 \oplus x_2$. It is equivalent to $x_1 + x_2 - 2x_1x_2$.
   - There is a perceptron in the middle layer; it is also called the *hidden layer*.
   - The hidden layer perceptron outputs the AND of the inputs x_1 and x_2, that is $x_1 \wedge x_2$. It is characterized by the equivalent form $x_1 + x_2 > 1$; so it outputs 1 only when both x_1 and x_2 are 1, else a 0 (zero) exactly like an AND gate.
   - The perceptron in the output layer has 3 inputs; they are x_1, x_2 and $-2x_1x_2$. It outputs the sum of the three inputs giving the equivalent $x_1 + x_2 - 2x_1x_2$ of the $Exclusive\ OR$ of x_1 and x_2 and is represented by $x_1 \oplus x_2$

2. Considering the other representation of XOR using $x_1\overline{x_2} + \overline{x_1}x_2$, the corresponding MLP is depicted in Fig. 3.4.

   - Note that there are 3 layers in this case also. The inputs are x_1 and x_2.
   - In the hidden layer there are two perceptrons. The top one outputs $x_1\overline{x_2}$; it is represented by the equivalent form $x_1 - x_2 > 0$. Similarly, the second perceptron in the layer outputs $\overline{x_1}x_2$; its equivalent representation is $x_2 - x_1 > 0$.
   - The output layer has a single perceptron which is an $Inclusive\ OR$ gate; it has two inputs. So, its output is $x_1\overline{x_2} + \overline{x_1}x_2$ that is equivalent to the XOR function.

- Even though these two ANNs are very simple, early work on MLPs exploited the results on these networks to highlight the fact it is possible to learn the weights connecting perceptrons (or neurons as they are called) in successive layers.

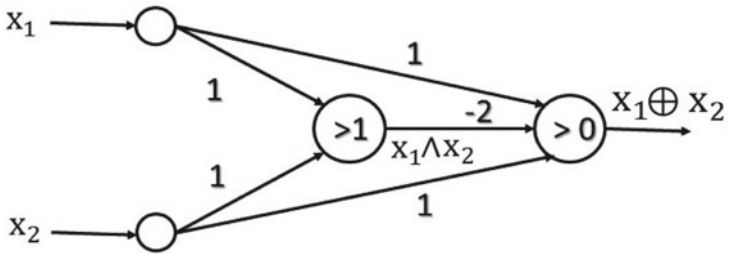

Fig. 3.3 Exclusive or represented as $x_1 + x_2 - 2x_1x_2$

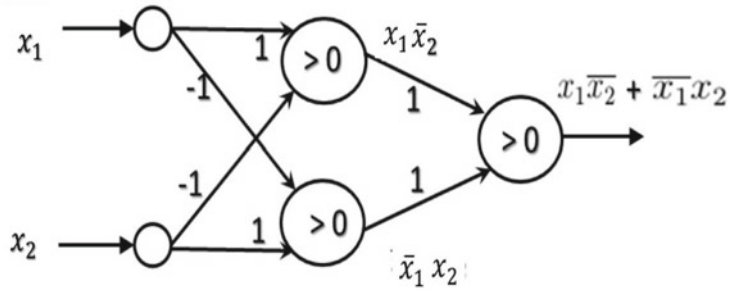

Fig. 3.4 Exclusive or represented as $x_1\overline{x_2} + \overline{x_1}x_2$

- *MLP*s are called *feedforward networks* as can be illustrated by using Fig. 3.4. The outputs of the neurons in layer i become the inputs to neurons in the $(i + 1)$th layer.
- If we observe Fig. 3.3, we see that the $(i + 1)$th layer gets inputs not only from the ith layer but also from the earlier layers. Typically, in an *MLP*, the weights connect neurons in two successive layers only.
- Learning in *MLP* networks amounts to starting with a set of initial weights and keep changing or updating the weights based on some criterion.

3.2.4 Training MLP Networks

The earliest and still the most popular algorithm for training *MLP*s is *backpropagation*. Before we consider the backpropagation algorithm, let us consider a related problem of training a single perceptron using the so called *delta rule*.

3.2.4.1 Delta Rule

Let us consider training a single perceptron.

- Let y^i_{obt} be the output obtained by the perceptron for input X_i.
- Let the target or expected output for the pattern X_i be y^i_{tar}.
- Let there be n training patterns given by $\{(X_1, y^1_{tar}), (X_2, y^2_{tar}), \ldots, (X_n, y^n_{tar})\}$.
- The idea is to start with some initial weight vector and update the weight vector so that *error between the target outputs and obtained outputs is minimized*.
- The error, $Error(W)$ is defined as

$$Error(W) = \frac{1}{2}\sum_{i=1}^{n}(y^i_{tar} - y^i_{obt})^2 \tag{3.2}$$

  Here, $\frac{1}{2}$ is used for the convenience of calculus.
- We know that $y^i_{obt} = f_a(W^t X_i + b) = f_a(W^{Aug}.X^{aug}_i)$ where $W^{Aug} = (b, w_1, \ldots, w_d)^t$ and $X^{aug}_i = (1, x_{i1}, x_{i2}, \ldots, x_{id})^t$ are the augmented vectors that subsume the *bias* b into W.
- So, the process of learning W and b is converted into learning W^{aug}. Henceforth, we use W instead of W^{aug} for the sake of simplicity in notation. Correspondingly X^{aug}_i is called X_i.
- We assume that f_a is a linear function defined as $f_a(wsum) = wsum$. So, $y^i_{obt} = f_a(W^t X_i) = W^t X_i$ where W and X_i are augmented respectively.
- Finding the *optimal* W is done in the case of the delta rule by using gradient descent. The partial derivatives involved in computing the gradient of $Error(W)$ with respect to W are calculated by using the *chain rule*:

$$\frac{\delta Error(W)}{\delta w_j} = \frac{\delta Error(W)}{\delta y^i_{obt}}.\frac{\delta y^i_{obt}}{\delta w_j} \tag{3.3}$$

- Note that

$$\frac{\delta Error(W)}{\delta y^i_{obt}} = -(y^i_{tar} - y^i_{obt}) \tag{3.4}$$

  and

$$\frac{\delta y^i_{obt}}{\delta w_j} = x_{ij}, \ \ for\ j = 0, 1, \ldots, d \tag{3.5}$$

  by assuming that $w_0 = b$ and $x_{i0} = 1$ for $i = 1, 2, \ldots, n$.

- So,

$$\frac{\delta Error(W)}{\delta w_j} = \frac{\delta Error(W)}{\delta y^i_{obt}}.\frac{\delta y^i_{obt}}{\delta w_j} = -(y^i_{tar} - y^i_{obt}).x_{ij} \tag{3.6}$$

- So, the gradient descent that employs the negation of the gradient will mean

$$W(k+1) = W(k) - \eta(-(y^i_{tar} - y^i_{obt})X_i) = W(k) + \eta(y^i_{tar} - y^i_{obt})X_i. \tag{3.7}$$

Table 3.2 Training data for the delta rule

Pattern	Value	y_{tar}
X_1	0	0
X_2	1	3
X_3	2	6
X_4	3	9

where the updated weight vector, $W(k+1)$ is obtained by updating the current weight vector, $W(k)$ and η is the learning parameter.

- It is called the *delta rule* or the *delta learning rule* because the difference (delta) between the target output and the obtained output is involved in the computation of the update.
- Note that the linear activation function is used instead of the linear threshold function, popularly used by Perceptron, to ensure that $\frac{\delta y^i_{obt}}{\delta w_j}$ can be computed; this is not possible when we use the linear threshold function.
- *Algorithm for Learning W*:

1. Choose $k = 1$ and initialize $W(k)$ with small values, and η with a small value.
2. Consider each pattern X_i in the training set and update to get $W(k+1) = W(k) + \eta(y^i_{tar} - y^i_{obt})X_i$. Set $k = k+1$. Update till all the patterns are considered; this is called an *epoch*.
3. Stop if there is no change in the weight vector for an entire epoch, else iterate by going to step 2.

- *Example:*
  - Let us consider a function $g : \Re \rightarrow \Re$ given by $g(X) = 3X$. Let the training data be as shown in the Table 3.2.
  - We consider a simple perceptron with no bias that is shown in Fig. 3.5.
  - Let us initialize the value of weight $W(1)$ to 0.1 and η to 1.
  - The first pattern in Table 3.2 is input to the perceptron in Fig. 3.5, It is correctly classified as $X_1 = 0 \Rightarrow y^1_{obt} = W.X_1 = 0 = y^1_{tar}$. So, $W(1)$ is not updated.
  - We consider $X_2 = 1$. The value of $y^2_{obt} = W.X_2 = 0.2\ .1 = 0.2$ and $y^2_{tar} = 3$. So, $W(2) = W(1) + \eta(y^2_{tar} - y^2_{obt}).X_2 = 0.2 + 1(3 - 0.2).1 = 3$.
  - Using the value of $W(2)$ all the patterns will be correctly classified. So, the algorithm stops and the weight value is 3.

- The example considered is very simple and it is primarily used to illustrate the *delta rule* based learning of the weight W. Also, it is a simple curve fitting/regression problem that may be viewed as a generalized version of the classification problem.

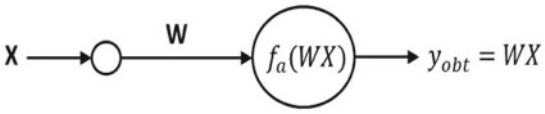

Fig. 3.5 Example perceptron with bias, b = 0

3.2.4.2 Backpropagation

Let us consider a multi-layer network (MLP).

- It will have $p(> 2)$ layers with the input layer (first layer), output layer (pth $layer$), and $p - 2$ hidden layers. For example, in Fig. 3.4 the value of p is 3 with one hidden layer.
- There will be n_q neurons in the qth layer for $q = 1, 2, \ldots, p$. In Fig. 3.4, there are 2 ($n_q = 2$) neurons in each of the input and hidden layers.
- Every neuron in layer q is connected to every neuron in layer $q + 1$ for $q = 1, \ldots, p - 1$. The connections in Fig. 3.4 exemplify this.
- The weight specified by w_{rs}^q is the weight associated with the connection between rth ($r = 1, 2, \ldots, n_q$) neuron in the qth layer and sth neuron in the $(q + 1)$th layer.

Some important properties of backpropagation may be summarized as follows:

- Let there be n training patterns given by $\{(X_i, y_t^{ip}), i = 1, 2, \ldots, n\}$, where y_t^{ip} stands for the target output at the pth layer (output layer) when X_i is input. Further, X_i is a vector of dimension n_1 and y_t^{ip} is a vector of dimension n_p.
- Let $wsum_{is}^{q+1}$ be the weighted sum input to the activation function at node s in the $(q + 1)$th layer for input X_i, where

$$wsum_{is}^{q+1} = \sum_{j=1}^{n_q} w_{js}^q y_o^{iqj}, \tag{3.8}$$

  where y_o^{iqj} is the output obtained at the jth node of the qth layer for input X_i.
- We assume that the same activation function, f_a, is used at all the neurons in the entire MLP network.
- Note that w_{js}^q affects the final error through $wsum_s^{q+1}$.
- It is a feedforward neural network. So, when training vector X_i is presented at the input layer of the network, then y_o^{i1j} $(= X_{ij})$ is the output of the jth node in the first layer (input layer).
- The output of the jth node in layer q is given by y_o^{iqj} $(= f_a(wsum_{ij}^{q-1}))$.
- Let the obtained output at the jth node in the output layer be y_o^{ipj}.
- We need to learn all the weights w_{rs}^q, $q = 1, \ldots, p$, $r = 0, 1, \ldots, n_q$, and $s = 1, \ldots, n_{q+1}$. Let W be the collection of these weights.
- Training the MLP is achieved by getting W that minimizes *squared error* across all the n patterns. It is given by

$$Error(W) = \sum_{i=1}^{n} \frac{1}{2} \sum_{j=1}^{n_p} (y_t^{ipj} - y_o^{ipj})^2. \tag{3.9}$$

- We use gradient descent and the update rule is given by

$$w_{rs}^q(k+1) = w_{rs}^q(k) - \eta \frac{\delta Error(W)}{\delta w_{rs}^q} \tag{3.10}$$

- We can compute $\frac{\delta Error(W)}{\delta w_{rs}^q}$, using the *chain rule* similar to the one used by the delta rule as

$$\frac{\delta Error(W)}{\delta w_{rs}^q} = \frac{\delta Error(W)}{\delta wsum_{is}^{q+1}} . \frac{\delta wsum_{is}^{q+1}}{\delta w_{rs}^q} \tag{3.11}$$

- Note that $\frac{\delta wsum_{is}^{q+1}}{\delta w_{rs}^q} = y_o^{iqr}$.
- Let $error_{is}^{q+1} = \frac{\delta Error(W)}{\delta wsum_{is}^{q+1}}$. It is possible to view it as backpropagated error at node s in layer $q+1$ when X_i is input to the MLP.
- The process of backpropagation of error is initiated at the output layer; once we know $error_{ij}^p$ for node j in the output layer, we propagate it back to nodes in layer $p-1$, then to nodes in layer $p-2$, and so on till the nodes in the input layer.
- The error propagation is characterized by

$$error_{ij}^q = \sum_{s=1}^{n_q+1} error_{is}^{q+1} W_{js}^q f_a'(wsum_{ij}^q) \tag{3.12}$$

- The previous error computation is a result of the following chain rule:

$$error_{ij}^q = \sum_{s=1}^{n_q+1} error_{is}^{q+1} . \frac{\delta wsum_{is}^{q+1}}{\delta y_o^{iqj}} . \frac{\delta y_o^{iqj}}{\delta wsum_{ij}^p} \tag{3.13}$$

- In propagating the error back, we first start with the output layer (layer p)

$$error_{ij}^p = \frac{\delta Error(W)}{\delta wsum_{ij}^p} = \frac{\delta Error(W)}{\delta y_o^{ipj}} . \frac{\delta y_o^{ipj}}{\delta wsum_{ij}^p} = (y_o^{ipj} - y_t^{ipj}) . f_a'(wsum_{ij}^q) \tag{3.14}$$

- This is easy to compute because for input X_i we go through the forward pass to compute y_o^{iqj} for $j = 1, 2, \ldots, n_q$ and $q = 1, 2, \ldots, p$. Once we have y_o^{ipj}, we can compute $(y_o^{ipj} - y_t^{ipj})$.
- The quantity $f_a'(wsum_{ij}^q)$ can be computed because the form of f_a' is known in advance based on the functional form of f_a.
- Once we compute $error_{ij}^p$, for all the nodes in the pth layer (output layer), then we can propagate back, using the earlier equation, to get error of nodes in the previous

layer, and iteratively till we get error at every node in the MLP. We can use these errors to update the weights across the network for pattern X_i.
- The process is repeated for all the patterns; such an iteration over all the patterns is called an *epoch*. This process is repeated over several such epochs till some *termination criterion* on the error at the output layer is met.
- During the early days of MLP research, there was more effort on
  - Why linear threshold activation is inadequate? There was a need for an activation function f_a that is differentiable for the backpropagation of error. One of the most popular is the sigmoid function given by $f_a(x) = \frac{1}{1+e^{-x}}$.
  - How many hidden layers are required to learn a required function on an MLP? The universal function approximation theorem showed that one hidden layer is adequate to approximate any function. The radial basis function networks were based on this.
  - What happens if we use more hidden layers? The number of weights in the MLP network contribute to the dimensionality of the problem. So, more hidden layers mean more weights and a higher dimensional problem. With smaller training sets, the learnt MLP can overfit.

3.3 Convolutional Neural Networks

In the previous section, we have examined the MLP network. Some of the problems associated with it are:

1. The sigmoid activation function can have *vanishing or exploding gradient*; so, it is not the right activation function. This is also linked with how the initial weights are chosen.
2. Overfitting the training data can occur if the number of hidden layers/neurons is large; this happens if the training data is small.
3. Most of the backpropagation training scenarios used software simulations on slower machines; in the early days people were even restricting the weights to have integer values to run the simulations faster.

There are better and efficient processing platforms available now. We will consider the details associated with the activation functions and weight initialization in the next two subsections.

3.3.1 Activation Function

- *Earlier Activation Functions:*
  In the case of delta rule, we have seen the use of the linear activation function. It is not useful in dealing with any required nonlinearity across multiple layers

as it collapses multiple layers in the network into one; this is because the relation between the input and the final output will be through another linear function. This is similar to multiplication of several matrices giving rise to another matrix. This prompts the use of a nonlinear activation function. A popularly used nonlinear activation function is the *sigmoid function*. Some of its properties are:

- It is given by $f_s(x) = \frac{1}{1+e^{-x}}$. So, it maps any real number to a value in [0, 1]. Further, values of the input x above 5 will take the output close to 1 and values below -5 make the output close to 0.
- It needs to compute the exponential of the argument which could be time consuming.
- Its derivative is $f_s'(x) = f_s(x)(1 - f_s(x))$. As x tends to a large value, $(1 - f_s(x))$ tends to 0 (zero) and if it tends to a small value $f_s(x)$ tends to zero. So, in either case the derivative tends to 0. Thus the gradient can vanish.
- It is not zero centered. It assumes only positive values. This can affect the resulting output badly when there are many hidden layers in the MLP.
- If the $wsum$ is small as the initial weights are small, then the derivative of the sigmoid function will be close to 0 and may even vanish. So, if there are more layers to be trained, then backpropagation may fail to update the weights in the earlier (closer to the input) layers as weights in such layers are considered for updation towards the end of error backpropagation. This is because of the *vanishing gradient* problem.
- On the contrary, if the weights are initialized with larger values, then it is possible to have the *exploding gradient problem* where the gradient can assume a value that is prohibitively large.
- A solution offered, to handle the zero-center problem, is in the form of the tanh function, f_t, that may be defined as

$$f_t(x) = 2f_s(2x) - 1. \tag{3.15}$$

  It is easy to see that f_t maps any real number to a value in the rang $[-1, +1]$.
- Both sigmoid and tanh functions are still used, even though they are not as popular as earlier as both may have difficulty with their gradients. It may lead to vanishing or exploding gradient problem which can impact the training accuracy and time.

- Activation Functions Popular with Deep Neural Networks (DNNs)
  - Rectified Linear Unit (ReLU): It is a popular activation function. It has the following characteristics:
    - It is defined as

$$f_r(x) = \begin{cases} 0 & x \leq 0 \\ x & x > 0 \end{cases} \tag{3.16}$$

    - It is popular because it is computationally simpler.
    - It is used only in the hidden layers.

  - It works well when x is positive. Its gradient vanishes when x is 0 or negative; so not useful for backpropagation when its input falls in this range. This is called the *Dying ReLU problem.*

– Leaky ReLU: It offers a solution to the *dying ReLU problem*. Its properties are:

  - It is defined as $f_l(x) = max(0.01x, x)$. So, it is a variant of ReLU function.
  - It permits backpropagation for input values that are less than or equal to zero also. However, in this range predictions based on $f_l(x)$ may be inconsistent.
  - It trains faster then ReLU.

– Softmax: It permits us to convert a vector of values to another vector of same size that has normalized values adding upto 1. Its characteristics are:

  - It is a mapping from $\Re^{n_p}$ to $(0, 1)^{n_p}$. It is specified as

$$f_{smax}(y_o^{ipj}) = \frac{e^{(y_o^{ipj})}}{\sum_{j=1}^{n_p} e^{(y_o^{ipj})}} \tag{3.17}$$

  - It is used at the output layer of a DNN to convert a vector of real numbers into vector of probabilities; the sum of the values of its outputs is 1.

3.3.2 Initialization of Weights

Different schemes have been used to initialize weights in the past.

- *Zero Weights*: Typically in perceptron training based on fixed increment rule, it is convenient to start with a zero weight vector and still guarantee convergence of the update algorithm when the classes are linearly separable. However, in the case of a DNN, *initializing all the weights to 0* or in general any constant value can lead to highly symmetric behaviour across the network leading every weight to be the same across the iterations.
- *Random weights*: It is possible to view a *deep neural network (DNN)* as a device that transforms the input, to match with the desired output, through successive layers. It is a lossy transformation. So, if we select the weights randomly, then the information loss in the initial layers may be so bad that the backpropagation algorithm may not be able to abstract the desired overall mapping even over a good number of iterations/epochs.
- *Smaller or larger weights:* In the previous subsection we have considered how initialization with smaller or larger weights can lead to vanishing or exploding gradient problems.

These issues associated with initialization were responsible for an appropriately normalized scheme to work. Some important normalization schemes that try to maintain zero mean and specified variance of the weights in the DNN are:

- *Xavier initialization and variants*:
  - Here the weights in layer q, $q = 1, \ldots, p$ are initialized by

$$w_{rs}^{q} \sim \left[-\frac{\sqrt{6}}{\sqrt{n_q + n_{q+1}}}, \frac{\sqrt{6}}{\sqrt{n_q + n_{q+1}}} \right] \tag{3.18}$$

    where weights are randomly drawn from a uniform distribution in the normalized range specified.
  - The bias for each neuron is initialized to 0 (zero).
  - This normalization is to ensure that the weights are chosen with a zero mean and a standard deviation that is a normalized version of 1.
  - It is a replacement of earlier normalization schemes that took into account only n_q.
  - This modification helps in ensuring that the activation outputs and gradients encountered in the backpropagation runs have variances that are neither too small nor too large.
- Kaiming Initialization:
  - A variant is proposed by Kaiming He et al. where values of weights w_{rs}^{q} are randomly chosen from the standard normal distribution and are multiplied by $\frac{\sqrt{2}}{\sqrt{n_q}}$ in this initialization.
  - This works better than Xavier initialization when ReLU activation is used.
  - It was observed that both training and testing errors converged to be requiring a smaller number of iterations/epochs to converge; 20 epochs appeared to be adequate in practice for a good performance.
- Typically these schemes conduct analysis based on looking at the variance of the product of weights and outputs of the neurons.

3.3.3 Deep Feedforward Neural Network

Before the year 2000, it was strongly believed that one or two hidden layers are adequate to deal with most of the machine learning tasks. One major observation was that backpropagation is based on gradient descent and it can only guarantee to reach a *locally optimal value* of the criterion function. This was the reason for *support vectors machines* ($SVMs$) to flourish, for more than two decades, as the machine learning benchmark tool as it guarantees globally optimal margin based learning in theory. However, in the past decade the earlier views ware significantly altered due to some important contributions in the area of deep learning. It became so important that every problem in the area of artificial intelligence (AI) is invariably solved using deep learning. Convolutional neural network is the popular feed forward

deep learning architecture. Some of the contributions related to convolutional neural networks are discussed below:

- *Convolution:* It is a well-known operation in signal processing with applications to speech signals (one-dimensional) and images (two-dimensional).
- Let I be a two-dimensional image input which is represented as an array of size $r \times s$; So, I has r rows and s columns.
- Let the convolution template (or kernel) T be a smaller size pattern of size $M \times N$.
- The convolution output, O, that is an array of size $(r - M + 1) \times (s - N + 1)$ is given by

$$O(i, j) = f(\sum_{m=1}^{M} \sum_{n=1}^{N} I(i + m - 1, j + n - 1) T1(m, n). \tag{3.19}$$

  The role of template T in convolution is to locate parts of the input image I that match with the pattern present in T.
- Function f may be defined as $f(x) = 1$ if $x > \theta$ else $f(x) = 0$ where θ is a threshold.
- Let us illustrate the convolution operation in two dimensions using the example in Fig. 3.6.
  - There are two parts labeled (a) (upper part) and (b) (lower part) in the figure. The input image in both the parts is the same. It is a 9×9 binary image array $INPUT$, of *character 7*, consisting of 81 pixels labeled $INPUT(1, 1)$ to $INPUT(9, 9)$. Note that $r = s = 9$ in this example.
  - It has a horizontal line segment in the top part against $INPUT(2, 1)$ to $INPUT(3, 9)$ (rows 2 and 3) and a vertical line segment in columns 7 to 9 across rows 2 to 9.

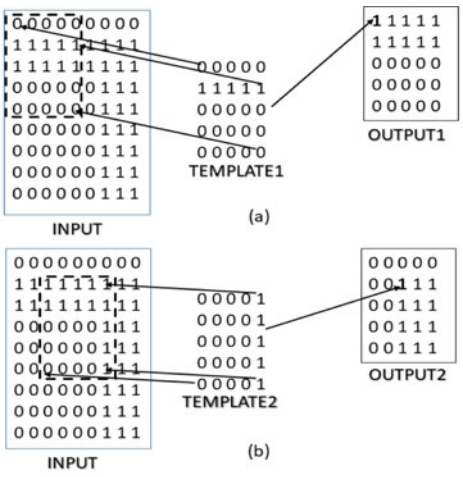

Fig. 3.6 Example convolution operation on input image of character 7

- In part (a), $Template1$ is used for convolution. It is a 5×5 binary pattern, $Template1$ of 25 binary pixels addressed by $Template1(1, 1)$ to $Template1$ $(5, 5)$. So, $M = N = 5$ here.
- Further, Note that in (a) $Template1$ is aligned with the top left part in $INPUT$ such that $Template1(1, 1)$ is aligned with $INPUT(1, 1)$ and $Template1$ $(5, 5)$ is aligned with $INPUT(5, 5)$.
- The pixel wise multiplication and addition as indicated in the equation for O gives us $OUTPUT1(1, 1) = f(5)$. If $\theta = 3$, then $f(5) = 1$. This value 1 is indicated in $OUTPUT(1, 1)$ in (a). If f is not used then we get the value $O(1, 1) = 5$.
- By shifting one position horizontally and multiplying and adding we get $OUTPUT1(1, 2)$. Further, while computing $OUTPUT1(1, 5)$, $TEMPLATE1$ will be aligned completely with the top 5 rows and rightmost 5 columns of $INPUT$. So, moving right further is not done which will make only a part of $TEMPLATE1$ align with a part of $INPUT$. Under these conditions, the other option is to create virtual columns filled with 0s. However, we consider alignment of the Nth column of the template with the sth column of the input and move no further.
- In order to compute $OUTPUT1(i, j)$ for $i = 2, \ldots, r - M + 1$ and $j = 1, \ldots, s - N + 1$ we need to align the top row of $TEMPLATE1$ with the ith row of $INPUT$.
- This results in the output image given by $OUTPUT1$ shown in (a). Note that this template has captured the horizontal lines in the input. It is popularly called as *mask* in image processing and *kernel/filter* is the popularly used term in $CNNs$.
- Similarly $TEMPLATE2$ in part (b) captures the vertical lines present in the character image in $INPUT$.
- This example is meant to illustrate the notion of convolution more than being a real mask for use in image convolution. Further, the threshold based function f is used here to get a binary output; such a function is not used in practice.

- *Feature Maps:* In a CNN, we will have multiple convolution layers. For example in Fig. 3.6 we have seen two different templates working on the same input image. $TEMPLATE1$ looks for horizontal lines in the input; this may be viewed as extracting one kind of feature. Similarly $TEMPLATE2$ looks for vertical lines; so extracts a different kind of feature. Each of the resulting outputs may be viewed as a *feature map*. In a more generic setting, we will have

  - Multiple templates/kernels each looking for a different kind of feature.
  - It is possible to have more than one occurrence of a feature in the same input image. For example, instead of character 7, if we consider the character 0 (zero) shown in Fig. 3.7 that has two horizontal (leftmost and rightmost) and two vertical (top and bottom) segments, then the same templates, $TEMPLATE1$ and $TEMPLATE2$ will each extract the respective features twice.
  - In practice, we may have images that are much larger in size compared to the small 9×9 input images shown in Figs. 3.6 and 3.7.

Fig. 3.7 Example of character zero with two horizontal and two vertical segments

- Also there can be a good number of templates each looking for one or more occurrences of the feature embedded in it. Correspondingly, there can be several feature maps one for each template.
- Each hidden layer may be viewed as made up of such multiple feature maps as many as the number of templates used in convolutions.
- The output of convolution is generally defined as

$$O(i, j) = f(\sum_{m=1}^{M} \sum_{n=1}^{N} w_{mn} I(i + m - 1, j + n - 1) T1(m, n)). \tag{3.20}$$

  In Fig. 3.6, the value of $w_{mn}, \forall m, n$ is taken to be 1. However, in a CNN these weights are learnt.
- An important aspect of learning these weights is that for each feature map we need to learn only $MN + 1$ weights where $M \times N$ is the size of the template and the extra 1 is to learn the bias term associated with the node in the hidden layer. This is an important characteristic of $CNNs$ and is called *weight sharing*.
- Further, the value of $MN + 1$ is much smaller in practice than the size of the image given by $r \times s$.
- We have assumed that the shifting of the template, after each multiply and add operations, is done by one column horizontally or one row vertically (by one pixel); in such a case the stride is 1. We can have strides of length 2 or more.

- Convolution and Pooling Layers: Each convolution layer has the input and hidden layers as shown in Fig. 3.8; a hidden layer has some L feature maps. So, the hidden layer will have $L \times M \times N$ neurons.
  - In a CNN there will be more than one such convolution layer. Typically after each convolution layer, there will be a pooling layer to reduce the dimensionality further.
  - A *pooling layer* is obtained from the features maps in the hidden layer of the previous convolution layer.
  - Let the size of each feature map be $u \times v$; so number of neurons in a feature map is uv.
  - Let the pooling be done by using a window of size $k \times k$, where k divides both u and v, over the feature map. This is done by considering $k \times k$ neurons in

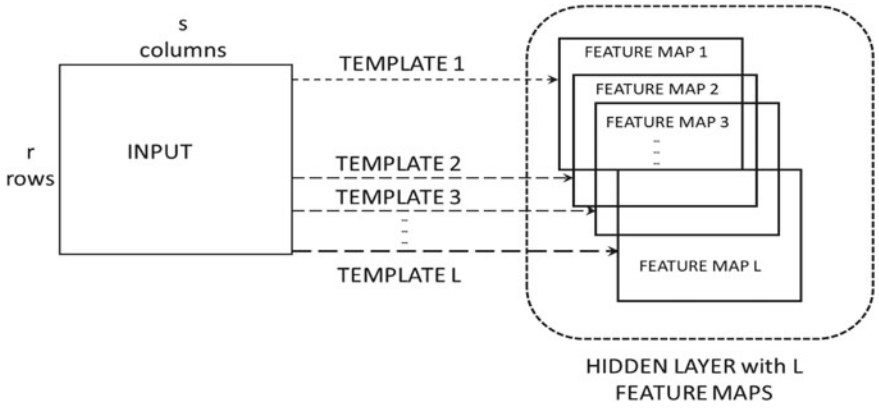

Fig. 3.8 Convolution layer with multiple feature maps in the hidden layer

the feature map at a time; the window is moved horizontally and vertically in a non-overlapping manner.
- In each window region of k^2 neurons, the respective k^2 outputs are pooled to output one value that is stored in the corresponding location output of the pooling layer.
- The output of the pooling layer is given by

$$Poolout(i, j) = g(\{fmo((i-1)k+1, (j-1)k+1), \ldots, fmo(ik, jk)\}) \tag{3.21}$$

where $fmo(p, q)$ feature map output of the neuron in the pth row and qth column of the feature map. Observe that $i = 1, \ldots, \frac{u}{k}$ and $j = 1, \ldots, \frac{v}{k}$.
- Note that the argument of g is a set of k^2 elements across rows $(i-1)k$ to ik (k rors) and columns $(j-1)k$ to jk (k columns). They are the outputs of neurons in the chosen $k \times k$ region in the feature map.
- The function g itself could popularly be the max, $average$, or $L_2 - norm$ of the k^2 values in the set.

- The overall architecture of the CNN will consist of several convolution layers; after each convolution layer there will be a pooling layer with the output of the feature maps in the layer forming the input of the pooling layer. The output of the pooling layer will be the input of the next convolution layer.
- Typically the final output layer of the CNN will be a fully connected layer that is connected to all the neurons in the previous layer.
- The CNN is trained using backpropagation. The error is propagated back from a layer to the previous layer through the relevant weights.

Some important properties of CNN are that

- It is the *state-of-the-art tool* for classification and prediction.

- It has been successfully used in large-scale applications where both the number of training patterns and/or the dimensionality of the data is large. In fact it works well only when the training data is large.
- It became popular because of its applications in image processing and speech processing applications.
- One of the important outcomes is a variant that has become popular in network applications in the form of *graph convolutional net* (GCN).

3.4 Recurrent Networks

Earlier in this chapter, we discussed MLP and CNN models. Some of the limitations associated with them are:

- They expect inputs of predetermined size and transform them into fixed-size vectors. In contrast, many real-world problems have an unknown size, such as machine translation, document classification tasks, which makes MLP and CNN type networks unsuitable for these applications.
- In many applications such as sentiment classification, sentence classification, etc., the input is a sequence of words and the computation at a step of the sequence depends on the current word and the previous words too. But MLP assumes that all the inputs are not dependent on each other and thus cannot process these inputs. Therefore, we require tools to deal with sequence data, where previous words also effect the computations at a later step.

Recurrent Neural Networks (RNNs), Long Short Term Memory ($LSTM$), Gated Recurrent Units (GRUs) are developed to solve the problems mentioned above. In the next two subsections, we discuss the RNN and the $LSTM$ models in detail.

3.4.1 Recurrent Neural Networks

A *Recurrent Neural Network* (RNN) is a multi-layered model that processes inputs sequentially. Some important characteristics are:

- RNN is a neural network model where previous outputs play a major role in determining the next output. These models have shown great success in many sequential tasks, especially in the *natural language processing* (NLP) domains.
- For example, a character level RNN considers each word as a single input (sequence), each character in the word as an element of the sequence, and each successive element is called a time step.
- RNNs use the same set of parameters for all the time steps of an input, which not only avoids overfitting but also learns dependencies between the elements at different time steps of the input. So, RNN performs the same operation on each element of the serial input. Thus these models are called recurrent.

- On the contrary, in a *vanilla neural network*, each input element is associated with a different set of weights due to which the network cannot work with serial inputs of varying sizes.
- RNN has *hidden states or units* which encapsulate the relations between elements of a serial input. Hidden states are also interpreted as *memory units.*
- Output calculation at each time step depends on the information present in the hidden state and the current input, thereby updating the weights of the model and the information in the hidden state.
- Figure 3.9 shows one time step of RNN. Figure 3.10 depicts the complete architecture of RNN.
- At each time step t, it takes the one-hot encoding of an element from the input sequence and outputs a vector whose each entry denotes the probability of the corresponding element being the next element in the sequence.

3.4.1.1 Working of Recurrent Neural Networks

Let us denote the input, hidden and output states at step t by x_t, hs_t and hy_t respectively. Initial hidden state hs_0 is generally initialized by zeros and x_1 (initial input) is a one-hot vector of the first element in a serial input.

- Current state of RNN is calculated by:

$$hs_t = \sigma\left(W_s hs_{t-1} + W_x x_t\right) \tag{3.22}$$

  where x_t is the present input, hs_{t-1} is the hidden state at time step t-1 and hs_t is the new hidden state at time step t. σ is an activation function (tanh or ReLU). W_s and W_x are the collection of trainable weight parameters.
- Output state at time step t is calculated using the current hidden state

$$hy_t = W_y hs_t \tag{3.23}$$

  where hy_t and hs_t are the output vector and the hidden state at time step t and W_y are the weights.
- To convert the output to a probability distribution (which is required for many tasks such as classification), softmax activation is used on hy_t,

$$o_t = \text{softmax}(hy_t) \tag{3.24}$$

- RNNs can have output at each time step or at only the final step. For example, task of classifying the entire sequence generates only one output at the last time step with no outputs at the intermediate time steps.

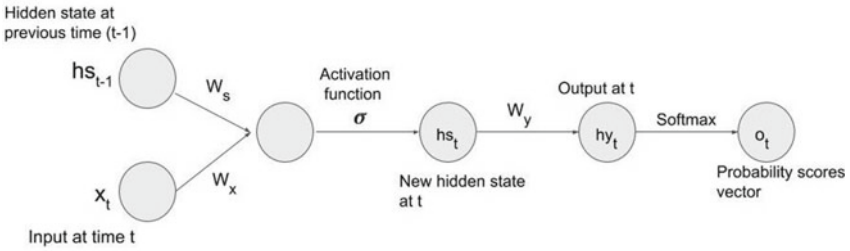

Fig. 3.9 A single layer of RNN

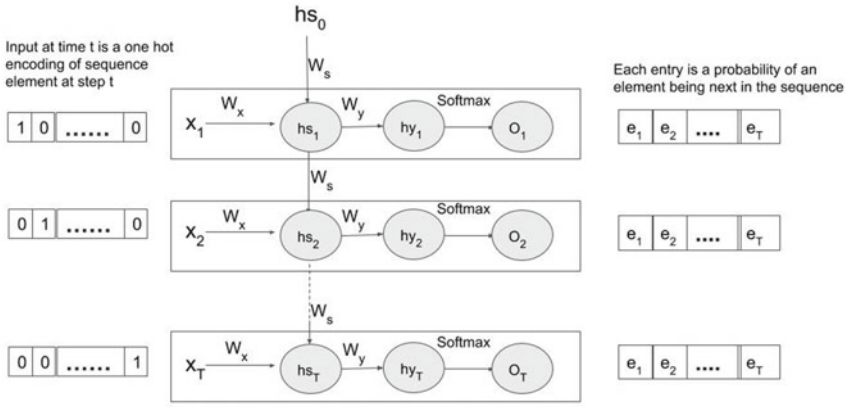

Fig. 3.10 Complete architecture of RNN

3.4.1.2 Backpropagation Through Time

RNN is also trained using the backpropagation algorithm. The recurrent network is a time sequence model. Therefore, backpropagation means going back in time and hence is called Backpropagation through time ($BPTT$).

- First of all, the total error for an input will be the sum of errors from each step.
  - Suppose loss at each step(t) is calculated by cross entropy between predicted vector($\hat{o_t}$) and the actual one hot encoding(l_t) of the correct output word. The total error for all the time steps is calculated as follows:

$$J(\hat{o}, o) = -\sum_{t=1}^{T} \hat{o_t} log(l_t) \tag{3.25}$$

  $J(\hat{o}, o)$ is the total error and T is the total number of time steps.

- Similar to errors, gradients are also summed up over all the time steps. We get the following equations corresponding to W_s, W_h and W_y:

$$\frac{\partial J}{\partial W_s} = \sum_{t=1}^{T} \frac{\partial J_t}{\partial W_s} \tag{3.26}$$

$$\frac{\partial J}{\partial W_h} = \sum_{t=1}^{T} \frac{\partial J_t}{\partial W_h} \tag{3.27}$$

$$\frac{\partial J}{\partial W_y} = \sum_{t=1}^{T} \frac{\partial J_t}{\partial W_y} \tag{3.28}$$

Here J_t is the error at time t. Each term in the summation will be evaluated similarly. So we will focus on the error term at time step t.

1. First we compute gradients wrt parameter W_y. The derivation of J_t wrt W_y depends only on the current time t. Formally, J_t depends on the predicted label ($\hat{o}$) (3.25) which depends on hy_t (3.24) and hy_t is a function of W_y (3.23). Thus, using the chain rule of differentiation we get the following equation:

$$\frac{\partial J_t}{\partial W_y} = \frac{\partial J_t}{\partial W_y} = \frac{\partial J_t}{\partial \hat{o}_t} \frac{\partial o_t}{\partial hy_t} \frac{\partial hy_t}{\partial W_y} \tag{3.29}$$

2. The process of calculating gradients wrt to W_x and W_s is different. Here we will calculate the gradients wrt W_s, and the same process can be repeated for the other.
   - Following the same steps as used for calculating the gradients for W_y and further noting that in RNN, W_s parameters are shared at all the time steps because of which changes in W_s will effect the error at time step t (J_t) even when $hs_1, hs_2, \ldots, hs_{t-1}, hs_t$ states are being computed. We get the following equation using the points mentioned above.

$$\frac{\partial J_t}{\partial W_s} = \sum_{q=0}^{t} \frac{\partial J_t}{\partial \hat{o}_t} \frac{\partial \hat{o}_t}{\partial hs_t} \frac{\partial hs_t}{\partial hs_q} \frac{\partial hs_q}{\partial W_s} \tag{3.30}$$

   - More generally, the third term in Eq. 3.30 is a chain of derivatives. As hs_k is a function of hs_{k-1} which depends on hs_{k-2} and this continues with first hidden state depending only on W_s. Based on this, the above equation can be rewritten as:

$$\frac{\partial J_t}{\partial W_s} = \sum_{q=0}^{t} \frac{\partial J_t}{\partial \hat{o}_t} \frac{\partial \hat{o}_t}{\partial hs_t} \left(\prod_{p=q+1}^{t} \frac{\partial hs_p}{\partial hs_{p-1}} \right) \frac{\partial hs_q}{\partial W_s} \tag{3.31}$$

- For example, while calculating gradient at time step t = 3 and examining the effect of the change in W_s on J_3 when hs_1 is being evaluated.

$$\frac{\partial J_t}{\partial W_s} = \frac{\partial J_3}{\partial \hat{o}_3} \frac{\partial \hat{o}_3}{\partial hs_3} \frac{\partial hs_3}{\partial hs_2} \frac{\partial hs_2}{\partial hs_1} \frac{\partial hs_1}{\partial W_s} \tag{3.32}$$

3.4.1.3 Vanishing and Exploding Gradients

The problems of vanishing and exploding gradients occur in deep feedforward neural networks and are already discussed. These problems also exist in RNNs. In this subsection, we discuss these problems and some existing solutions.

1. Recurrent Neural Networks suffer from the short-term memory problem, i.e., RNNs cannot learn dependencies between far apart elements. The diminished information from previous time steps is the consequence of the vanishing gradient problem.
2. Therefore, Vanilla RNNs face problems dealing with long-range dependencies. For example, in the sentence, "Tyson had a trip to a hill station with his friend", "his" is used for "Tyson", and to figure this relation, RNNs will have to remember a lot of information.
3. More formally, the expanded Eq. 3.30 includes the chain of derivatives of $\frac{\partial hs_3}{\partial hs_k}$ that depends on the derivative of the activation functions. The value of the derivatives of tanh or sigmoid activation functions can reach 1 or 1/4, respectively.
4. Also, gradients of tanh and sigmoid become 0 during saturation. As a consequence, the gradients of neurons from far away steps approach 0. The multiplication of such small values significantly shrinks the gradient, and after a few steps, it vanishes, and hence those neurons will not learn anything.
5. Some existing solutions for the Vanishing Gradient problem are:
   - Proper initialization of the W matrix needs to be used.
   - A better solution is to use a variant of RNN, such as Long Short-Term Memory ($LSTM$) or Gated Recurrent Unit (GRU). Both these models can overcome the problem of vanishing gradients.
6. Another problem with RNN is the exploding gradient, the opposite of the vanishing gradient, in which gradients become very large (will have values as NaN). This can be solved by using a threshold value as a cap on all the gradients.

3.4.2 Long Short Term Memory

The short term memory of Recurrent Neural Networks makes difficult for RNNs to carry information from previous time steps that are far apart because gradients become very small, and no learning can take place from that point. Long Short Term Memory ($LSTMs$), an improvement over $RNNs$, were developed to solve this vanishing gradient problem and handle long range dependencies between the elements. Some essential characteristics of $LSTM$s are:

1. Major difference between RNN and $LSTM$ is their cell structure. Each RNN module is a simple single layer neural network structure, while each $LSTM$ module is a more complicated structure and uses four gates or four neural network layers.
2. LSTM core idea is its cell state and its gates (input, forget, output).
   - Cell state (represented as C_t) at time t carries and passes only the appropriate information during training.
   - Gates are used to distinguishing the important and the irrelevant information from the cell state and based on the importance score update the cell state.
   - These gates are composed of multiplication operation and a neural network with a sigmoid layer.
3. Just like humans tend to forget unimportant words and remember only the main parts of a speech, gates in $LSTM$ also help learn only the relevant information. Hence, they solve problems associated with the short term memory of RNN.
4. Similar to RNNs, $LSTM$s too have hidden state hs_t at each time t.

3.4.2.1 Different Gates Used by $LSTM$

$LSTM$ uses various GATES for different purposes. All these gates are neural networks.

- Sigmoid layer is used in almost all the gates to determine the information to be updated and the information to be discarded.
- Sigmoid function outputs values between 0 and 1, with 1 representing the most important information and 0 representing the least important information.
- If any value in the cell is multiplied by 0, then the cell forgets that information and does not let that information pass through; otherwise, the value is fed to the later time steps.

Now we discuss important units of $LSTM$ architecture. In the text and equations below, we use t to denote the current time step, hs_{t-1} to represent the hidden state at the previous time step t-1, and x_t as input at time t.

- *Forget Gate* decides which values to be discarded from the cell state at the previous time step.

- It uses a sigmoid layer that takes the previous hidden state hs_{t-1} along with the current input x_t and gives values between 0 and 1.
- All important information will have values closer to 1. The forget gate can be described as follows:

$$fg_t = \sigma(W_{fg}[hs_{t-1}, x_t] + \beta_{fg}) \tag{3.33}$$

Here W_{fg}, β_{fg} are the weights and bias terms associated with the forget gate layer. σ is an activation function. fg_t is the output of the forget gate at time step t.

- The next step is to determine the information to be included in the cell state.
  - It is done by a group of 2 layers.
    - *Input gate layer* uses a sigmoid layer that takes the previous hidden state and the current input and decides which information to update. Equation 3.34 describes this step.

$$gi_t = \sigma(W_{gi}[hs_{t-1}, x_t] + \beta_{gi}) \tag{3.34}$$

Here W_{gi}, β_{gi} are the parameters of the input gate layer.

    - *tanh layer* outputs values between -1 and 1. It gives a new set of entries $\hat{ci}_t$ that can be included in the cell state. Equation 3.35 describes this step.

$$\hat{ci}_t = tanh(W_{ci}[hs_{t-1}, x_t] + \beta_{ci}) \tag{3.35}$$

Here W_{ci}, β_{ci} are the parameters of tanh layer.

  - Multiplication of these two outputs determines the useful entries of $\hat{ci}_t$ with the help of sigmoid output gi_t.

$$z_t = gi_t * ci_t \tag{3.36}$$

- The next step is to form the new cell state C_t from the information calculated so far. Remember that fg_t knows what to throw away and what to keep for further states.
  - To form a new cell state, the first step is to multiply the cell state C_{t-1} with the forget vector fg_t.
  - This helps the cell to forget unimportant information by multiplying with a value closer to zero, which is determined by the fg_t entries. That way, it can focus only on appropriate part of the sequence until the previous time step.
  - We then add z_t (the input gate output) to the resulting product. The result is the new cell state, which contains the updated, appropriate information. The following equation describes these steps.

$$C_t = fg_t * C_{t-1} + z_t \tag{3.37}$$

where z_t is defined in Eq. 3.31.

- The next important gate layer is the *output gate layer*, which determines the next hidden state based on the new cell state just formed.
  - A sigmoid layer is used to decide the information from C_t to pass on to the next states. It takes x_t and C_{t-1} as inputs.
  - The next step is to use a tanh layer on the new cell state C_t.
  - Finally, multiplication of these sigmoid and tanh outputs will determine the information for the next hidden state. These steps can be described by the following equation:

$$\begin{aligned} og_t &= \sigma(W_{og}[hs_{t-1}, x_t] + \beta_{og}) \\ hs_t &= og_t * tanh(C_t) \end{aligned} \tag{3.38}$$

  where W_{og}, β_{og} are the parameters of the output gate layer. hs_t is the new hidden state at time step t.

- Figure 3.11 shows a complete layer of the $LSTM$ model.
  - This figure shows all the steps that we explained above to generate a new hidden and a cell state by using previous hidden state, current cell state, current input and all the gates.
  - In this figure, σ represents the sigmoid layer. Each blue circle denotes one of the layers described in the text above, and each red circle represents a mathematical operation (element-wise multiplication or addition).

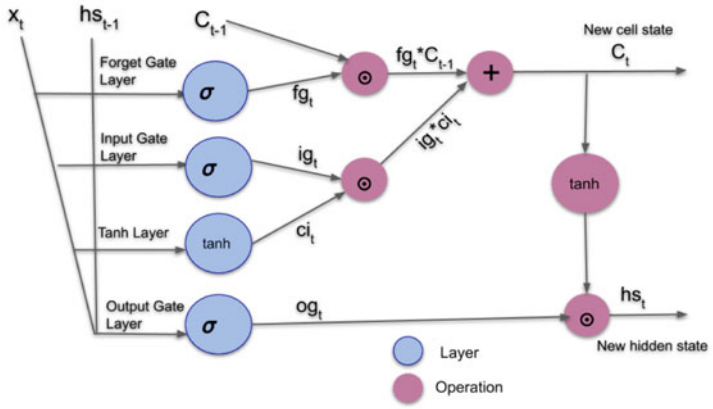

Fig. 3.11 LSTM complete architecture

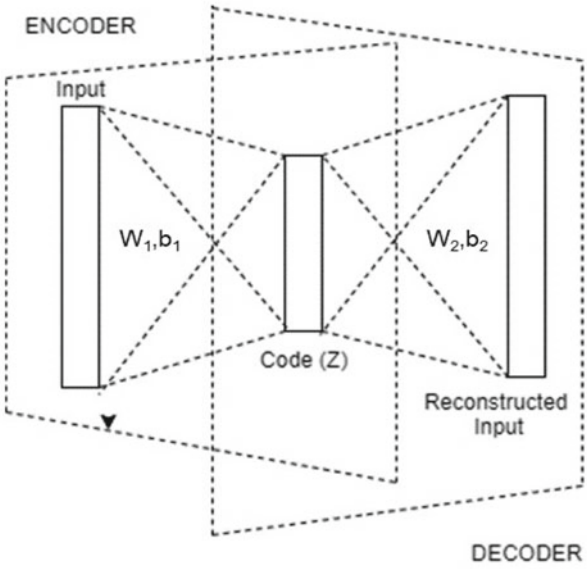

Fig. 3.12 Autoencoder architecture

3.5 Learning Representations Using Autoencoders

An autoencoder is a popular unsupervised model for learning representations in a low dimensional space.

- It is a neural network that employs a non-linear transformation on the input to compress it. This is done so that the original data can be reconstructed using this low-dimensional representation.
- An autoencoder incorporates an encoder and a decoder. The encoder compresses the input. The decoder decompresses the compressed input to get back the original input. Another important component is the code, also known as the bottleneck, which is the compressed representation of the input.
- An ideal autoencoder should be sensitive to the input to learn a less lossy reconstruction but, at the same time, should not learn an identity mapping.
- Significant applications of autoencoders are dimensionality reduction and representation learning. The recent development of variational autoencoders makes autoencoders useful as generative models also.
- Figure 3.12 shows the architecture of a simple autoencoder.
  - As shown by the outer boxes in the figure, the encoder comprises the input and the hidden layers, while the decoder is made up of the hidden and the output layers.
  - Input, output, and hidden layers can have any number of units (neurons).

  - W_1, b_1 and W_2, b_2 indicate the weights, bias between input and hidden layers and hidden and output layers respectively.
    - In some cases, these weights can be tied together, such that $W_2 = W_1^T$, which is sometimes used to avoid overfitting as the number of trainable parameters is less in this setting.

- An autoencoder computes the compressed representation of the input as follows:
  - The Encoder receives the input (x) (input layer) and computes the latent representation (code) as $h = \sigma(W_1 x + b_1)$. This is fed to the decoder which outputs the reconstructed input (z) as $z = \sigma(W_2 x + b_2)$, where σ is an activation function.
  - Autoencoders are trained using gradient descent, and parameters are learned using the backpropagation rule as used by MLPs.
  - Loss function of autoencoder depends on the input x and the output z as the loss should reflect the deviation of the reconstructed input (output of autoencoder) from the input.
    - One possible loss function is L2 Norm. Let the dataset contains $x_1, x_2, \ldots, x_n$ samples, where n is the number of samples in the dataset. The L2 norm loss is given by:

$$Loss = \frac{1}{2} \sum_{i=1}^{n} |x^i - z^i|^2 \tag{3.39}$$

    - It is clear from the loss function that there is no role of the label information during training; thus, it is an unsupervised learning scheme. But autoencoders can be trained in a supervised manner for a specific downstream task such as the classification task.
    - For the supervised classification task, some fully connected layers with the last layer being the softmax layer, are appended. The model is trained in an end to end fashion using cross entropy loss, which leverages the label information. In this case, the information loss would be less.

3.5.1 Types of Autoencoders

There are many variations of autoencoders. In this subsection we briefly describe some of them.

- *Sequence-to-Sequence Autoencoder:* It uses recurrent neural networks for encoder and decoder operations. These autoencoders first convert the entire sequence to a single lower dimension encoding, following which the decoder tries to get back the sequence from this encoding.
- *Deep Autoencoder*: This is an extension of vanilla autoencoder that has many layers in the encoder and the decoder part. The first set of layers compresses the

input while the next set of layers (decoder) will reconstruct the input from the latent representations. Also, as we go deeper, more high order or more abstract features are learned.

- *Undercomplete Autoencoder*: This variant develops a generalized model with the encoder's output dimension (code dimension) smaller than the input dimension. These autoencoders can ensure that the model is not copying the input and is learning important data distribution features because a smaller code dimension restricts the information flowing through the model. This type of model only constraints the number of hidden units in the bottleneck layer. And there can be cases where the hidden layer has only one neuron while encoder and decoder having abundant capacity tend to overfit the data and, therefore, couldn't learn anything meaningful.
- *Regularized Autoencoders*: These autoencoders provide the ability to learn other properties of data instead of copying the input to the output even if the encoder and decoder have the superabundant capacity or the encoder output dimension is equal to or greater than the input dimension. Regularized autoencoders leverage a loss function which helps learn only the variations and not the redundancies in the data and further avoids overfitting.
- Various regularization techniques are used in order to prevent encoder and decoder from learning the identity functions.
  - *Denoising Autoencoders*: These autoencoders add some random gaussian noise to the inputs before training. However, the model still reconstructs the uncorrupted data because the model loss depends on the original input and not on the noisy input. This acts as a regularizer and helps autoencoders distinguish more essential parts of the input as these autoencoders try to undo the corruption. The loss function is as follows:

$$Loss = \frac{1}{2}\sum_{i=1}^{n} |x^i - \hat{z}^i|^2 \tag{3.40}$$

    Here $\hat{z}^i$ is the output of the model with input being the corrupted data. Same mechanism can be applied at any layer of the autoencoder.
  - *Sparse Autoencoder*: This variant minimizes the number of non-zero entries in the latent representation. It constrains the capacity of the model by penalizing the activations within the hidden layer, and hence without any limitation on the number of nodes in the hidden layer, the model can learn the input data distribution irrespective of the encoder and decoder capacity.

3.6 Summary

Deep learning is an important topic that has found applications in several areas.

- The availability of large scale datasets and powerful computing platforms have played an important role making deep learning possible.
- *Deep neural networks* form the *de facto* tools for deep learning.
- Perceptron is one of the earliest and the most basic neural network models. It forms the basis for a variety of neural network models.
- The need and importance of MLPs is considered next. Backpropagation is the training algorithm that was important in training MLP networks.
- The difficulty in increasing the number of layers was analysed to identify the vanishing and exploding gradient problems.
- Important contributions behind the design and training of deep neural networks in the form of activation functions like ReLU and softmax are examined.
- Another important contribution behind the success of deep neural networks is the weight initialization and updating.
- Other factors that impacted deep learning include convolution, pooling and weight sharing.
- Several important deep learning models including CNN, RNN, $LSTM$ and autoencoders are considered.
- CNNs have been extensively used in image processing and speech processing.
- For analysing sequence data RNNs and $LSTM$ are popularly used. They find applications in natural language processing and biological sequence data.
- Autoencoders are the most popular dimensionality reduction tools that can compress input data using a non-linear transformation.
- We considered some of the important properties associated with CNNs, RNNs and autoencoders, the difficulties in training these models, and solutions provided.
- The deep learning models are important in the context of network data analysis. We will consider specific roles of CNNs, autoencoders and RNNs in the context of network embeddings in the later chapters.

Bibliography

1. Bishop CM (2005) Neural networks for pattern recognition. Oxford University Press
2. Murty MN, Raghava R (2016) Support vector machines and perceptrons. Springer briefs in computer science
3. Loiseau JCB (2019) Rosenblatt's perceptron, the first modern neural network, https://towardsdatascience.com/rosenblatts-perceptron-the-very-first-neural-network-37a3ec09038a/
4. Mazur M (2015) https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/
5. Nielsen MA (2015) Neural networks and deep learning, vol 2018. Determination Press, San Francisco, CA, USA

6. Mhaskar HN, Micchelli CA (1994) How to choose an activation function. In: Advances in neural information processing systems, pp 319–326
7. Wu J (2017) Convolutional neural networks. Published online at https://cs.nju.edu.cn/wujx/teaching/15CNN.pdf
8. Britz D (2015) Recurrent neural networks tutorial, part 3 - backpropagation through time and vanishing gradients, http://www.wildml.com/2015/10/recurrent-neural-networks-tutorial-part-3-backpropagation-through-time-and-vanishing-gradients/
9. Wolf W (2016) Recurrent neural network gradients, and lessons learned therein, http://willwolf.io/2016/10/18/recurrent-neural-network-gradients-and-lessons-learned-therein/
10. Gupta DS (2017) Fundamentals of deep learning - introduction to recurrent neural networks, https://www.analyticsvidhya.com/blog/2017/12/introduction-to-recurrent-neural-networks/
11. Olah C (2015) Understanding LSTM networks, https://colah.github.io/posts/2015-08-Understanding-LSTMs/
12. Srivastava P (2017) Essentials of deep learning : introduction to long short term memory, https://www.analyticsvidhya.com/blog/2017/12/fundamentals-of-deep-learning-introduction-to-lstm/
13. Nguyen M (2018) Illustrated guide to LSTM's and GRU's: a step by step explanation, https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21/
14. Goodfellow I, Bengio Y, Courville A (2016) Deep learning. MIT Press, http://www.deeplearningbook.org/
15. Jordan J (2018) Introduction to autoencoders, https://www.jeremyjordan.me/autoencoders/
16. Choi HI (2019) Lecture 16: autoencoders (Draft: version 0.7. 2)