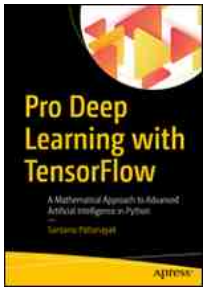

Pro Deep Learning with TensorFlow: A Mathematical Approach to Advanced Artificial Intelligence in Python

by Santanu Pattanayak

Chapter 2: Introduction to Deep-Learning Concepts and TensorFlow

S. Pattanayak, *Pro Deep Learning with TensorFlow*, https://doi.org/10.1007/978-1-4842-3096-1_2

Deep Learning and Its Evolution

Deep learning evolved from artificial neural networks, which have existed since the 1940s. Neural networks are interconnected networks of processing units called artificial neurons that loosely mimic the axons found in a biological brain. In a biological neuron, dendrites receive input signals from various neighboring neurons, typically more than one thousand of them. These modified signals are then passed on to the cell body or soma of the neuron, where these signals are summed together and then passed on to the axon of the neuron. If the received input signal is more than a specified threshold, the axon will release a signal, which will be passed on to the neighboring dendrites of other neurons. Figure 2-1 depicts the structure of a biological neuron for reference.

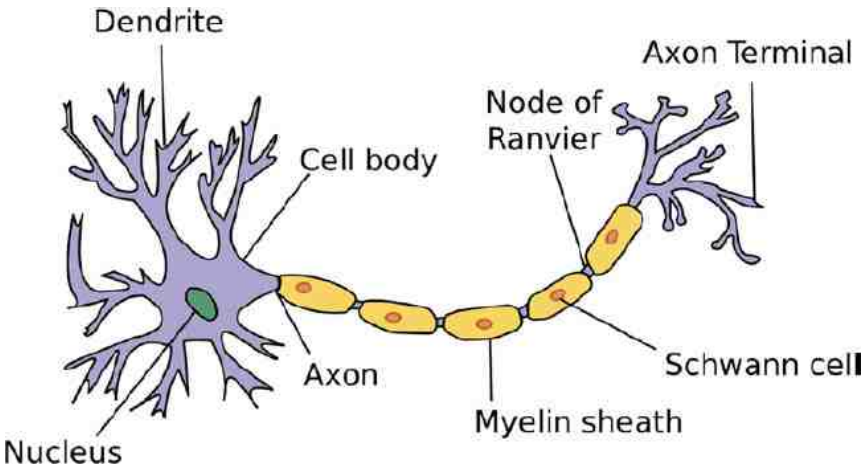

Figure 2-1: Structure of a biological neuron

Artificial neuron units are inspired by the biological neurons, with some modifications for convenience. Much like the dendrites, the input connections to the neuron carry the attenuated or amplified input signals from other neighboring neurons. The signals are passed on to the neuron, where the input signals are summed up and then a decision is made as to what to output based on the total input received. For instance, for a binary threshold neuron, an output value of 1 is provided when the total input exceeds a pre-defined threshold; otherwise, the output stays at 0. Several other types of neurons are used in artificial neural networks, and their implementation only differs with respect to the activation function on the total input to produce the neuron output. In Figure 2-2, the different biological equivalents are tagged in the artificial neuron for easy analogy and interpretation.

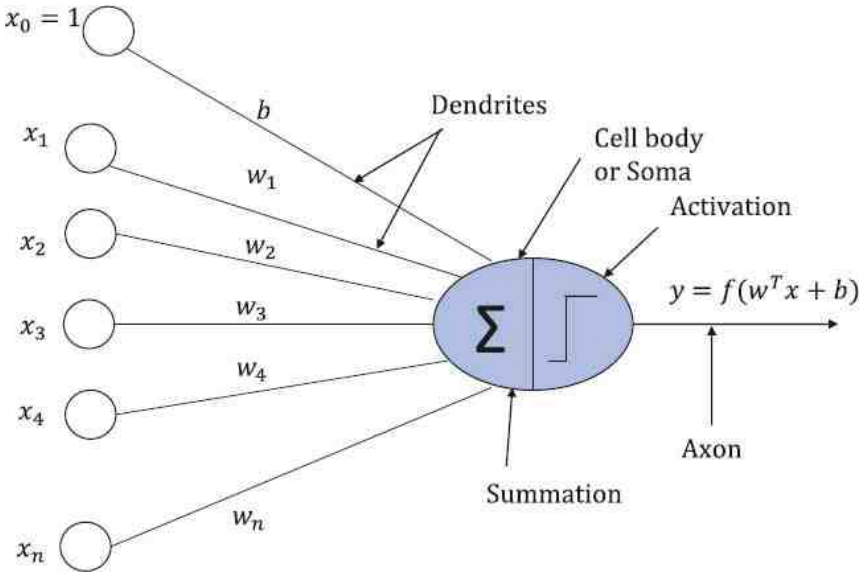

Figure 2-2: Structure of an artificial neuron

Artificial neural networks started with a lot of promise in the early 1940s. We will go through the chronology of major events in the artificial neural network community to get a feel of how this discipline has evolved over the years and what challenges were faced along the way.

- Warren McCullogh and Walter Pitts, two electrical engineers, published a paper titled "A Logical Calculus of the Ideas Immanent in

Nervous Activity," related to neural networks, in 1943. The paper can be located at http://www.cs.cmu.edu/~epxing/Class/10715/reading/McCulloch. and.Pitts.pdf. Their neurons have a binary output state, and there are two types of input to the neurons: excitatory inputs and inhibitory inputs. All excitatory inputs to the neuron have equal positive weights. If all the inputs to the neuron are excitatory and if the total input $\sum_i w_i x_i > 0$, the neuron would output 1. In cases where any of the inhibitory inputs are active or $\sum_i w_i x_i \leq 0$, the output would be 0. Using this logic, all the Boolean logic functions can be implemented by one or more such neurons. The drawback of these networks was that they had no way of learning the weights through training. One must figure out the weights manually and combine the neurons to achieve the required computation.

- The next big thing was the Perceptron, which was invented by Frank Rosenblatt in 1957. He, along with his collaborators, Alexander Stieber and Robert H. Shatz, documented their invention in a report titled *The Perceptron—A Perceiving and Recognizing Automaton*, which can be located at https://blogs.umass.edu/brain-wars/files/20l6/03/rosenblatt- 1957.pdf. The Perceptron was built with the motive of binary classification tasks. Both the weights and bias to the neuron can be trained through the Perceptron learning rule. The weights can be both positive and negative. There were strong claims made by Frank Rosenblatt about the capabilities of the Perceptron model. Unfortunately, not all of them were true.

- Marvin Minsky and Seymour A. Papert wrote a book titled *Perceptrons: An Introduction to Computational Geometry* in 1969 (MIT Press), which showed the limitations of the Perceptron learning algorithm even on simple tasks such as developing the XOR Boolean function with a single Perceptron. A better part of the artificial neural network community perceived that these limitations showed by Minsky and Papert applied to all neural networks, and hence the research in artificial neural networks nearly halted for a decade, until the 1980s.

- In the 1980s, interest in artificial neural networks was revived by Geoffrey Hilton, David Rumelhart, Ronald Williams, and others, primarily because of the backpropagation method of learning multi-layered problems and because of the ability of neural networks to solve non-linear classification problems.

- In the 1990s, support vector machines (SVM), invented by V. Vapnik and C. Cortes, became popular since neural networks were not scaling up to large problems.

- Artificial neural networks were renamed deep learning in 2006 when Geoffrey Hinton and others introduced the idea of unsupervised pre-training and deep-belief networks. Their work on deep-belief networks was published in a paper titled "A Fast Learning Algorithm for Deep Belief Nets." The paper can be located at https://www.cs.toronto.edu/~hinton/absps/fastnc.pdf.

- ImageNet, a large collection of labeled images, was created and released by a group in Stanford in 2010.

- In 2012, Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton won the ImageNet Competition for achieving an error rate of 16 percent, whereas in the first two years the best models had around 28 percent and 26 percent error rates. This was a huge margin of win. The implementation of the solution had several aspects of deep learning that are standard in any deep-learning implementation today.

  – Graphical processing units (GPUs) were used to train the model. GPUs are very good at doing matrix operations and are computationally very fast since they have thousands of cores to do parallel computing.
  – Dropout was used as a regularization technique to reduce overfitting.
  – Rectified linear units (ReLU) were used as activation functions for the hidden layers.

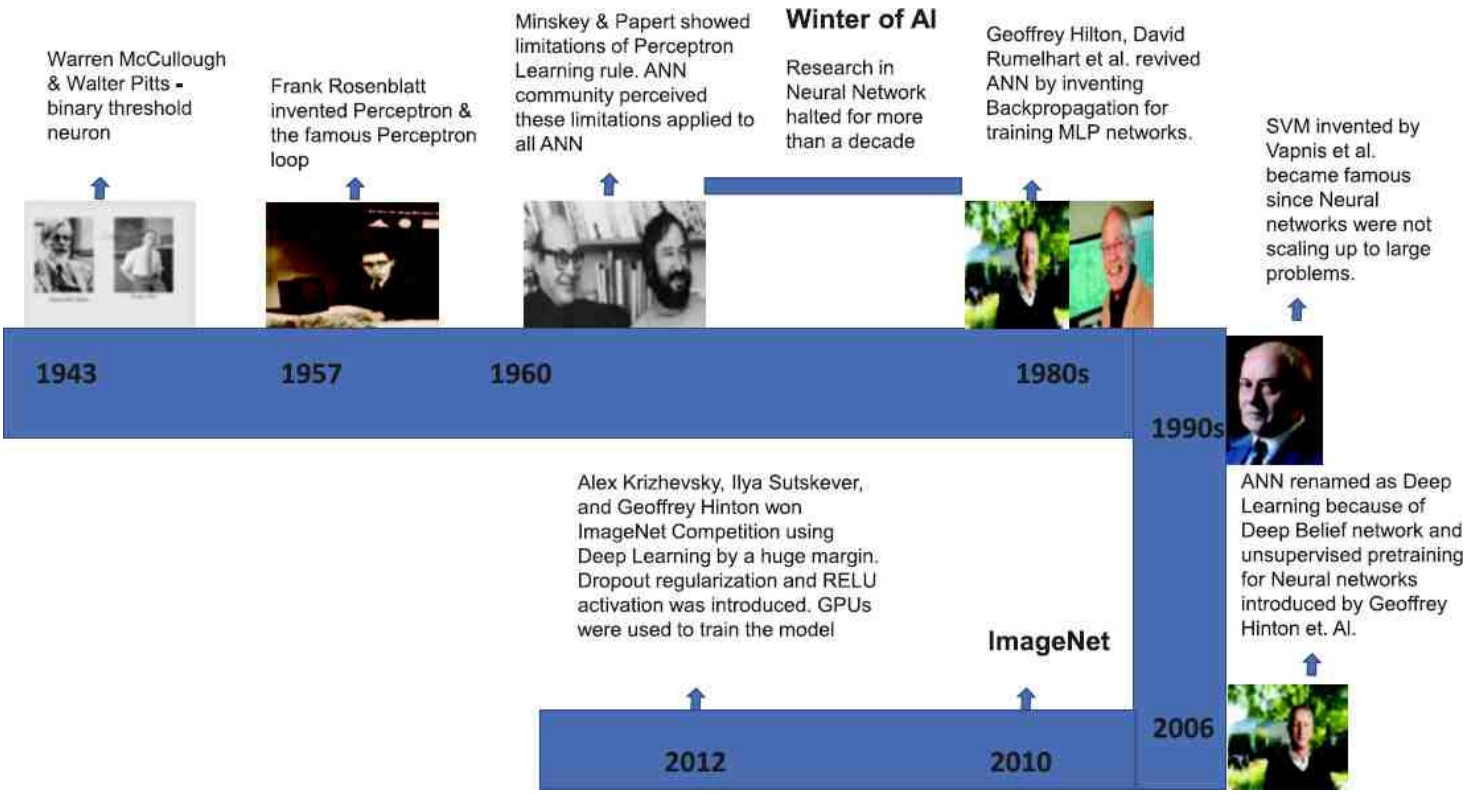

Figure 2-3: Evolution of artificial neural networks

Figure 2-3 shows the evolution of the artificial neural network into deep learning. ANN stands for artificial neural network, MLP stands for multi-layer Perceptron, and AI stands for artificial intelligence.

Perceptrons and Perceptron Learning Algorithm

Although there are limitations to what Perceptron learning algorithms can do, they are the precursor to the advanced techniques in deep learning we see today. Hence, a detailed study of Perceptrons and the Perceptron learning algorithm is worthwhile. Perceptrons are linear binary classifiers that use a hyperplane to separate the two classes. The Perceptron learning algorithm is guaranteed to fetch a set of weights and bias that classifies all the inputs correctly, provided such a feasible set of weights and bias exist.

Perceptron is a linear classifier, and as we saw in Chapter 1, linear classifiers generally perform binary classification by constructing a hyperplane that separates the positive class from the negative class.

The hyperplane is represented by a unit weight vector $w' \in \mathbb{R}^{n\times 1}$ that is perpendicular to the hyperplane and a bias term b that determines the distance of the hyperplane from the origin. The vector $w' \in \mathbb{R}^{n\times 1}$ is chosen to point toward the positive class.

As illustrated in Figure 2-4, for any input vector $x' \in \mathbb{R}^{n\times 1}$ the dot product with the negative of the unit vector $w' \in \mathbb{R}^{n\times 1}$ would give the distance b of the hyperplane from the origin since x' and w' are on the side opposite of the origin. Formally, for points lying on the hyperplane,

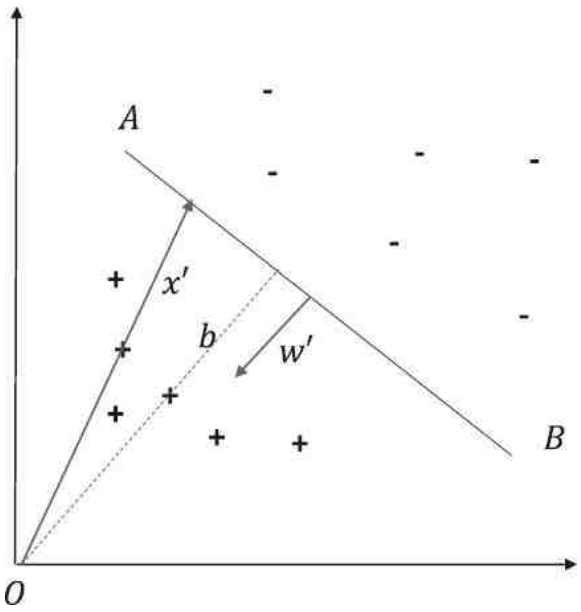

Figure 2-4: Hyperplane separating two classes

$$-w'^{T}x' = b \Rightarrow w'^{T}x' + b = 0$$

Similarly, for points lying below the hyperplane, i.e., the input vectors $x'_{+} \in \mathbb{R}^{n\times 1}$ belonging to the positive class, the negative of the projection of x'_{+} on w' should be less than *b*. So, for the points belonging to the positive class,

$$-w'^{T}x' < b \Rightarrow w'^{T}x' + b > 0$$

Similarly, for points lying above the hyperplane, i.e., the input vectors $x'_{-} \in \mathbb{R}^{n\times 1}$ belonging to the negative class, the negative of the projection of x'_{-} on w' should be greater than *b*. So, for the points belonging to the negative class,

$$-w'^{T}x' > b \Rightarrow w'^{T}x' + b < 0$$

Summarizing the preceding deductions, we can conclude the following:

- $w'^{T}x' + b = 0$ corresponds to the hyperplane and all $x' \in \mathbb{R}^{n\times 1}$ that lie on the hyperplane will satisfy this condition. Generally, points on the hyperplane are taken to belong to the negative class.
- $w'^{T}x' + b > 0$ corresponds to all points in the positive class.
- $w'^{T}x' + b \leq 0$ corresponds to all points in the negative class.

However, for Perceptron we don't keep the weight vector *w′* as a unit vector but rather keep it as any general vector. In such cases, the bias *b* would not correspond to the distance of the hyperplane from the origin, but rather it would be a scaled version of the distance from the origin, the scaling factor being the magnitude or the l^2 norm of the vector *w′* i. e. $||w'||_2$. Just to summarize, if *w′* is any general vector perpendicular to the hyperplane and is pointing toward the positive class, then $w'^{T}x + b = 0$ still represents a hyperplane where *b* represents the distance of the hyperplane from the origin times the magnitude of *w′*.

In the machine-learning domain, the task is to learn the parameters of the hyperplane (i.e., *w′* and *b*). We generally tend to simplify the problem to get rid of the bias term and consume it as a parameter within *w* corresponding to a constant input feature of 1,1 as we discussed earlier in Chapter 1.

Let the new parameter vector after adding the bias be $w \in \mathbb{R}^{(n+1)\times 1}$ and the new input features vector after adding the constant term 1 be $x \in \mathbb{R}^{(n+1)\times 1}$, where

$$x' = \begin{bmatrix} x_1 \, x_2 \, x_3 \, .. \, x_n \end{bmatrix}^T$$

$$x = \begin{bmatrix} 1 \, x_1 \, x_2 \, x_3 \, .. \, x_n \end{bmatrix}^T$$

$$w' = \begin{bmatrix} w_1 \, w_2 w_3 \, .. \, w_n \end{bmatrix}^T$$

$$w = \begin{bmatrix} b \, w_1 \, w_2 w_3 \, .. \, w_n \end{bmatrix}^T$$

By doing the preceding manipulation, we've made the hyperplane in R^n at a distance from the origin pass through the origin in the $\mathbb{R}^{(n+1)}$ vector space. The hyperplane is now only determined by its weight parameters vector $w \in \mathbb{R}^{(n+1)\times 1}$, and the rules for classification get simplified as follows:

- $w^T x = 0$ corresponds to the hyperplane, and all $x \in \mathbb{R}^{(n+1)\times 1}$ that lies on the hyperplane will satisfy this condition.
- $w^T x > 0$ corresponds to all points in the positive class. This means the classification is now solely determined by the angle between the vectors w and x. If input vector x makes an angle within –90 degrees to +90 degrees with the weight parameter vector w, then the output class is positive.
- $w^T x \leq 0$ corresponds to points in the negative class. The equality condition is treated differently in different classification algorithms. For Perceptrons, points on the hyperplanes are treated as belonging to the negative class.

Now we have everything we need to proceed with the Perceptron learning algorithm.

Let $x^{(i)} \in \mathbb{R}^{(n+1)\times 1} \;\; \forall i = \{1,2,\ldots,m\}$ represent the m input feature vectors and $y^{(i)} \in \{0,1\} \; \forall i = \{1,2,\ldots,m\}$ the corresponding class label.

The Perceptron learning problem is as follows:

- Step 1 – Start with a random set of weights $w \in \mathbb{R}^{(n+1)\times 1}$.
- Step 2 – Evaluate the predicted class for a data point. For an input data point $x^{(i)}$ if $w^T x^{(i)} > 0$ then the predicted class $y_p^{(i)} = 1$, else $y_p^{(i)} = 0$. For the Perceptron classifier, points on the hyperplane are generally considered to belong to the negative class.
- Step 3 – Update the weight vector w as follows:
- If $y_p^{(i)} = 0$ and the actual class $y^{(i)} = 1$, update the weight vector as $w = w + x^{(i)}$
- If $y_p^{(i)} = 1$ and the actual class $y^{(i)} = 0$, update the weight vector as $w = w \_ x^{(i)}$
- If $y_p^{(i)} = y^{(i)}$, no updates are required to w.
- Step 4 – Go to Step 2 and process the next data point.
- Step 5 – Stop when all the data points have been correctly classified.

Perceptron will only be able to classify the two classes properly if there exists a feasible weight vector w that can linearly separate the two classes. In such cases, the Perceptron Convergence theorem guarantees convergence.

Geometrical Interpretation of Perceptron Learning

The geometrical interpretation of Perceptron learning sheds some light on the feasible weight vector w that represents a hyperplane separating the positive and the negative classes.

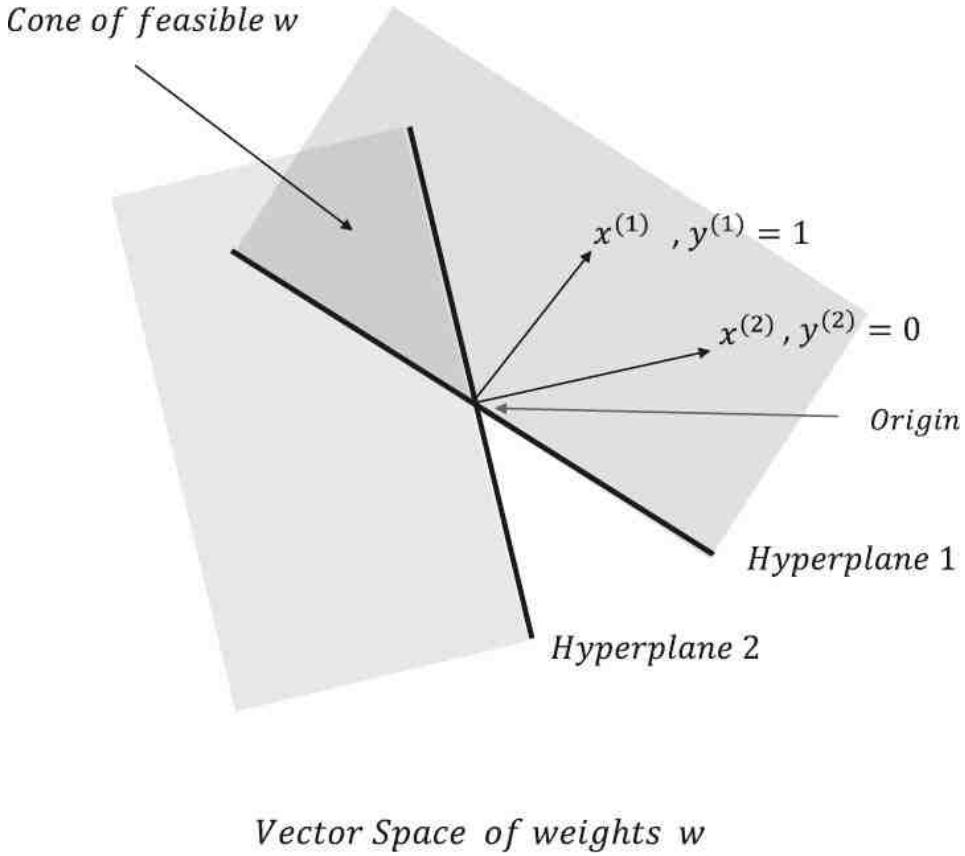

Figure 2-5: Hyperplanes in weight space and feasible set of weight vectors

Let us take two data points, ($x^{(1)}$, $y^{(1)}$) and ($x^{(2)}$, $y^{(2)}$), as illustrated in Figure 2-5. Further, let $x^{(i)} \in \mathbb{R}^{3\times1}$ include the constant feature of 1 for the intercept term. Also, let's take $y^{(1)} = 1$ and $y^{(2)} = 0$ (i.e., data point 1 belongs to the positive class while data point 2 belongs to the negative class).

In an input feature vector space, the weight vector determines the hyperplane. Along the same lines, we need to consider the individual input vectors as being representative of hyperplanes in the weight space to determine the feasible set of weight vectors for classifying the data points correctly.

In Figure 2-5, hyperplane 1 is determined by input vector $x^{(1)}$, which is perpendicular to hyperplane 1. Also, the hyperplane passes through the origin since the bias term has been consumed as a parameter within the weight vector w. For the first data point, $y^{(1)} = 1$. The prediction for the first data point would be correct if $w^T x^{(1)} > 0$. All the weight vectors w that are within an angle of –90 to +90 degrees from the input vector $x^{(1)}$ would satisfy the condition $w^T x^{(1)} > 0$. They form the feasible set of weight vectors for the first data point, as illustrated by the shaded region above hyperplane 1 in Figure 2-5.

Similarly, hyperplane 2 is determined by input vector $X^{(2)}$, which is perpendicular to hyperplane 2. For the second data point, $y^{(2)} = 0$. The prediction for the second data point would be correct if $w^T x^{(2)} \leq 0$. All the weight vectors w that are beyond an angle of –90 to +90 degrees from the input vector $X^{(2)}$ would satisfy the condition $w^T x^{(2)} \leq 0$. They form the feasible set of weight vectors for the second data point as illustrated by the shaded region below hyperplane 2 in Figure 2-5.

So, the set of weight vectors w that satisfy both the data points is the region of overlap between the two shaded regions. Any weight vector w in the region of overlap would be able to linearly separate the two data points by the hyperplanes they define in the input vector space.

Limitations of Perceptron Learning

The perceptron learning rule can only separate classes if they are linearly separable in the input space. Even the very basic XOR gate logic can't be implemented by the perceptron learning rule.

For the XOR logic below are the input and the corresponding output label or classes

- $x_1 = 1$, $x_2 = 0$ $y = 1$
- $x_l = 0$, $x_2 = 1$ $y = 1$

- $x_1 = 1, x_2 = 1$ $y = 0$
- $x_1 = 0, x_2 = 0$ $y = 0$

Let us initialize the weight vector $w \rightarrow [0\ 0\ 0]^T$, where the first component of the weight vector corresponds to the bias term. Similarly, all the input vectors would have their first components as 1.

- For $x_1 = 1, x_2 = 0, y = 1$ the prediction is $w^T x = \begin{bmatrix}0 & 0 & 0\end{bmatrix}\begin{bmatrix}1\\1\\0\end{bmatrix} = 0$. Since $w^T x = 0$ the data point would be classified as 0, which doesn't match the actual class of 1. Hence, the updated weight vector as per the Perceptron rule should be $w \rightarrow w + x = \begin{bmatrix}0\\0\\0\end{bmatrix} + \begin{bmatrix}1\\1\\0\end{bmatrix} = \begin{bmatrix}1\\1\\0\end{bmatrix}$.
- For $x_1 = 0, x_2 = 1, y = 1$ the prediction is $w^T x = \begin{bmatrix}1 & 1 & 0\end{bmatrix}\begin{bmatrix}1\\0\\1\end{bmatrix} = 1$. Since $w^T x = 1>0$, the data point would be correctly classified as 1. Hence, there would be no update to the weight vector, and it stays at $\begin{bmatrix}1\\1\\0\end{bmatrix}$.
- For $x_1 = 1, x_2 = 1, y = 0$ the prediction is $w^T x = \begin{bmatrix}1 & 1 & 0\end{bmatrix}\begin{bmatrix}1\\1\\1\end{bmatrix} = 2$. Since $w^T x = 2$, the data point would be classified as 1, which doesn't match the actual class of 0. Hence the updated Weight vector should be $w \rightarrow w - x = \begin{bmatrix}1\\1\\0\end{bmatrix} - \begin{bmatrix}1\\1\\1\end{bmatrix} = \begin{bmatrix}0\\0\\-1\end{bmatrix}$.
- For $x_1 = 0, x_2 = 0, y = 0$ the prediction is $w^T x = \begin{bmatrix}0 & 0 & -1\end{bmatrix}\begin{bmatrix}1\\0\\0\end{bmatrix} = 0$. Since $w^T x = 0$, the data point would be correctly classified as 0. Hence, there would be no update to the weight vector w.

So, the weight vector after the first pass over the data points is $w = [0\ 0 – 1]^T$. Based on the updated weight vector w, let's evaluate how well the points have been classified.

- For data point 1, $w^T x = \begin{bmatrix}0 & 0 & -1\end{bmatrix}\begin{bmatrix}1\\1\\0\end{bmatrix} = 0$, so it is wrongly classified as class 0.
- For data point 2, $w^T x = \begin{bmatrix}0 & 0 & -1\end{bmatrix}\begin{bmatrix}1\\0\\1\end{bmatrix} = -1$, so it is wrongly classified as class 0.
- For data point 3, $w^T x = \begin{bmatrix}0 & 0 & -1\end{bmatrix}\begin{bmatrix}1\\1\\1\end{bmatrix} = -1$, so it is wrongly classified as class 0.
- For data point 4, $w^T x = \begin{bmatrix}0 & 0 & -1\end{bmatrix}\begin{bmatrix}1\\0\\0\end{bmatrix} = 0$, so it is wrongly classified as class 0.

Based on the preceding classifications, we see after the first iteration that the Perceptron algorithm managed to classify only the negative class correctly. If we were to apply the Perceptron learning rule over the data points again, the updates to the weight vector *w* in the second pass would be as follows:

- For data point 1, $w^T x = \begin{bmatrix} 0 & 0 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} = 0$, so it is wrongly classified as class 0. Hence, the updated weight as per the Perceptron rule is $w \to w + x = \begin{bmatrix} 0 \\ 0 \\ -1 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix}$.
- For data point 2, $w^T x = \begin{bmatrix} 1 & 1 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} = 0$, so it is wrongly classified as class 0. Hence, the updated weight as per the Perceptron rule is $w \to w + x = \begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \\ 0 \end{bmatrix}$.
- For data point 3, $w^T x = \begin{bmatrix} 2 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} = 3$, so it is wrongly classified as class 1. Hence, the updated weight as per perceptron rule is $w \to w - x = \begin{bmatrix} 2 \\ 1 \\ 0 \end{bmatrix} - \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix}$.
- For data point 4, $w^T x = \begin{bmatrix} 1 & 0 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = 1$, so it is wrongly classified as class 1. Hence, the updated weight as per perceptron rule is $w \to w - x = \begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix} - \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ -1 \end{bmatrix}$.

The weight vector after the second pass is $[0\ 0\ -1]^T$, which is the same as the weight vector after the first pass. From the observations made during the first and second passes of Perceptron learning, it's clear that no matter how many passes we make over the data points we will always end up with the weight vector $[0\ 0\ -1]^T$. As we saw earlier, this weight vector can only classify the negative class correctly, and so we can safely infer without loss of generality that the Perceptron algorithm will always fail to model the XOR logic.

Need for Non-linearity

As we have seen, the Perceptron algorithm can only learn a linear decision boundary for classification and hence can't solve problems where non-linearity in the decision boundary is a need. Through the illustration of the XOR problem we saw that the Perceptron is incapable of linearly separating the two classes properly.

We need to have two hyperplanes to separate the two classes, as illustrated in Figure 2-6, and the one hyperplane learned through the Perceptron algorithm doesn't suffice to provide the required classification. In Figure 2-6, the data points in between the two hyperplane lines belong to the positive class while the other two data points belong to the negative class. Requiring two hyperplanes to separate two classes is the equivalent of having a non-linear classifier.

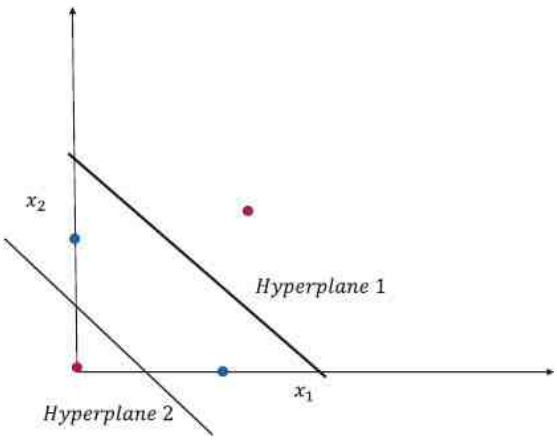

Figure 2-6: XOR problem with two hyperplanes to separate the two classes

A multi-layer Perceptron (MLP) can provide non-linear separation between classes by introducing non-linearity in the hidden layers. Do note that when a Perceptron outputs a 0 or 1 based on the total input received, the output is a non-linear function of its input. Everything said and done while learning the weights of the multi-layer Perceptron is not possible to achieve via the Perceptron learning rule.

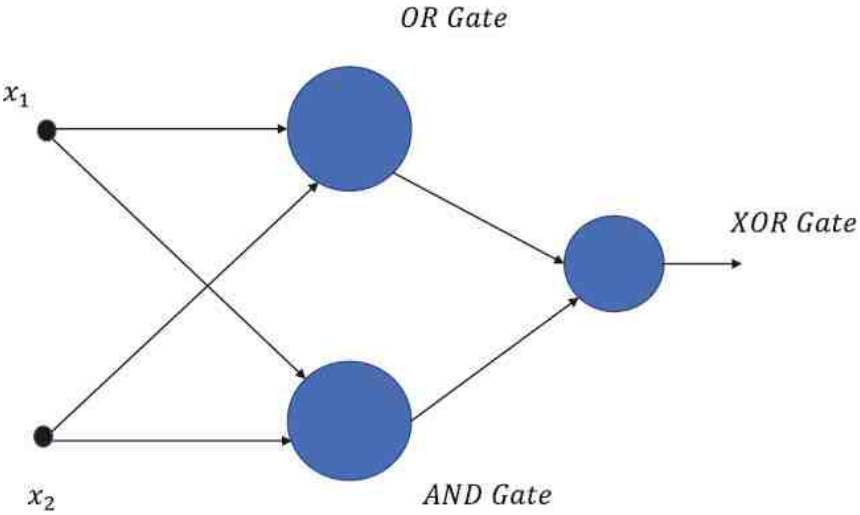

Figure 2-7: XOR logic implementation with multi-layer Perceptrons network

In Figure 2-7, the XOR logic has been implemented through a multi-layer Perceptrons network. If we have a hidden layer comprising two Perceptrons, with one capable of performing the OR logic while the other is capable of performing the AND logic, then the whole network would be able to implement the XOR logic. The Perceptrons for the OR and AND logic can be trained using the Perceptron learning rule. However, the network as a whole can't be trained through the Perceptron learning rule. If we look at the final input to the XOR gate, it will be a non-linear function of its input to produce a non-linear decision boundary.

Hidden Layer Perceptrons' Activation Function for Non-linearity

If we make the activation functions of the hidden layers linear, then the output of the final neuron would be linear, and hence we would not be able to learn any non-linear decision boundaries. To illustrate this, let's try to implement the XOR function through hidden layer units that have linear activation functions.

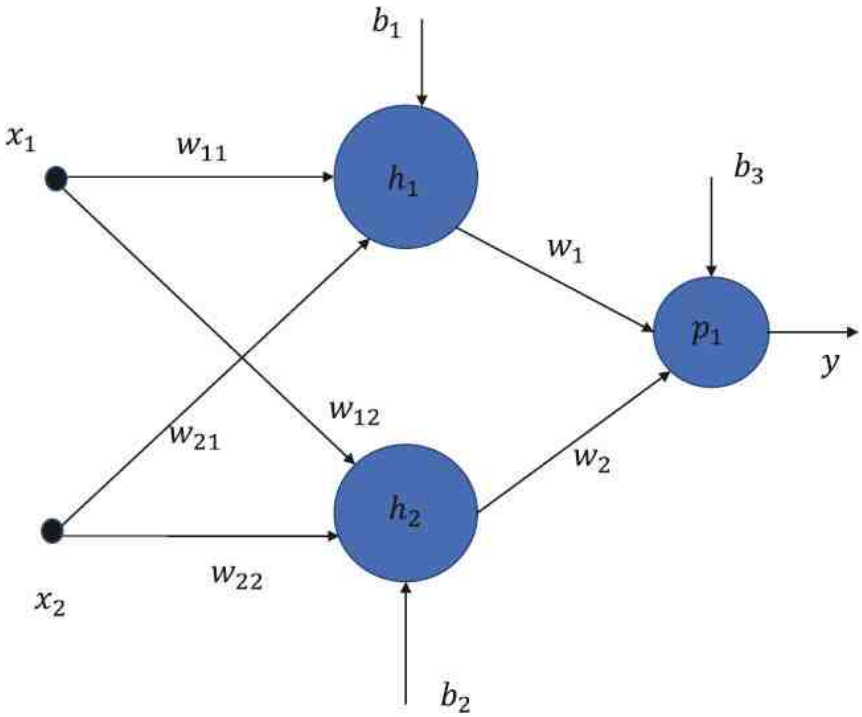

Figure 2-8: Linear output hidden layers in a two-layer Perceptrons network

Figure 2-8 shows a two-layer Perceptrons network with one hidden layer. The hidden layer consists of two neuron units. We look at the overall output of the network when the activations in the hidden units are linear:

- Output of the hidden unit $h_1 = w_{11}x_1 + w_{21}x_2 + b_1$
- Output of the hidden unit $h_2 = w_{12}x_1 + w_{22}x_2 + b_2$
- Output of the output unit $p_1 = w_1(w_{11}x_1 + w_{21}x_2 + b_1) + w_2(w_{12}x_1 + w_{22}x_2 + b_2) + b_3 = (w_1w_{11} + w_2w_{12})x_1 + (w_1w_{21} + w_2w_{22})x_2 + w_1b_1 + w_2b_2 + b_3$

As deduced in the preceding lines, the final output of the network—i.e., the output of unit p_1—is a linear function of its inputs, and thus the network can't produce a non-linear separation between classes.

If instead of a linear output being produced by the hidden layer we introduce an activation function represented as $f(x) = 1/(1 + e^{-x})$, then the output of the hidden unit $h_1 = 1/(1 + e^{-(w_{11}x_1 + w_{21}x_2 + b_1)})$.

Similarly, the output of the hidden unit $h_2 = 1/(1 + e^{-(w_{12}x_1 + w_{22}x_2 + b_2)})$.

The output of the output unit $p_1 = w_1/(1 + e^{-(w_{11}x_1 + w_{21}x_2 + b_1)}) + w_2/(1 + e^{-(w_{12}x_1 + w_{22}x_2 + b_2)}) + b_3$.

Clearly, the preceding output is non-linear in its inputs and hence can learn more complex non-linear decision boundaries rather than using a linear hyperplane for classification problems. The activation function for the hidden layers is called a sigmoid function, and we will discuss it in more detail in the subsequent sections.

Different Activation Functions for a Neuron/Perceptron

There are several activation functions for neural units, and their use varies with respect to the problem at hand and the topology of the neural network. In this section, we are going to discuss all the relevant activation functions that are used in today's artificial neural networks.

Linear Activation Function

In a linear neuron, the output is linearly dependent on its inputs. If the neuron receives three inputs x_1, x_2, and x_3, then the output y of the linear neuron is given by $y = w_1x_1 + w_2x_2 + w_3x_3 + b$ where w_1, w_2, and w_3 are the synaptic weights for the input x_1, x_2, and x_3 respectively, and b is the bias at the neuron unit.

In vector notation, we can express the output $y = w^T + b$.

If we take $w^T + b = z$, then the output with respect to the net input z will be as represented in Figure 2-9.

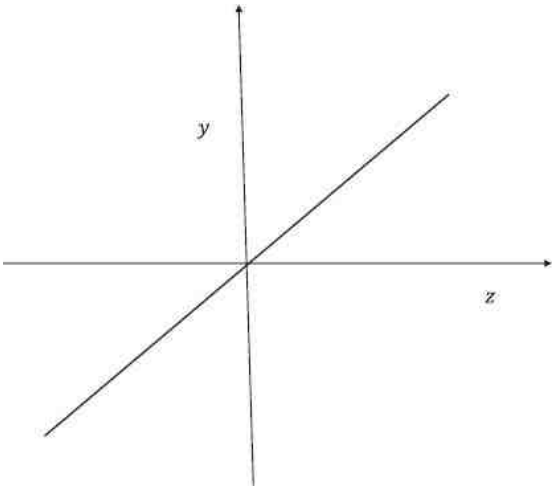

Figure 2-9: Linear output hidden layers in a two-layer Perceptrons network

Binary Threshold Activation Function

In a binary threshold neuron (see Figure 2-10), if the net input to the neuron exceeds a specified threshold then the neuron is activated; i.e., outputs 1 else it outputs 0. If the net linear input to the neuron is $z = w^T x + b$ and k is the threshold beyond which the neuron activates, then

$$y = 1 \qquad if\ z > k$$

$$y = 0 \qquad if\ z \leq k$$

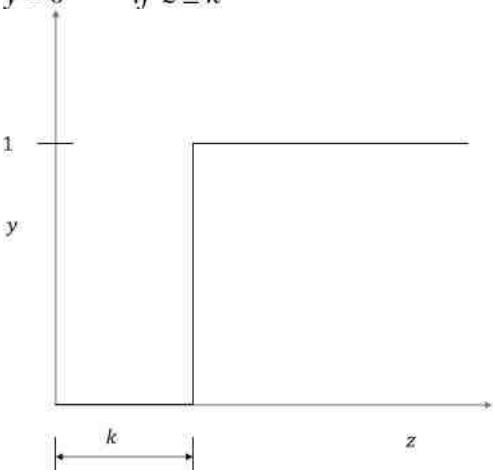

Figure 2-10: Binary threshold neuron

Generally, the binary threshold neuron is adjusted to activate at threshold 0 by adjusting the bias. The neuron is activated when

$$w^T x + b > k \Rightarrow w^T x + (b - k) > 0$$

Sigmoid Activation Function

The input–output relation for a sigmoid neuron is expressed as the following:

$$y = 1/(1 + e^{-z})$$

where $z = w^T x + b$ is the net input to the *sigmoid* activation function.

- When the net input z to a sigmoid function is a positive large number $e^{-Z} \rightarrow 0$ and so $y \rightarrow 1$.
- When the net input z to a sigmoid is a negative large number $e^{-Z} \rightarrow \infty$ and so $y \rightarrow 0$.

- When the net input z to a sigmoid function is 0 then $e^{-z} \rightarrow = 1$ and so $y = \frac{1}{2}$.

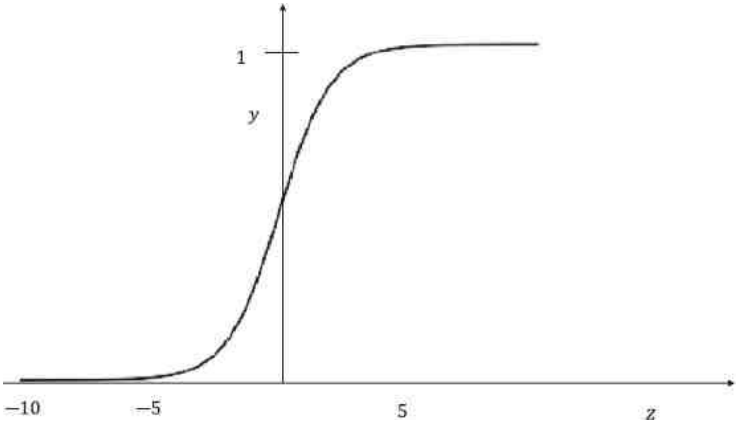

Figure 2-11: Sigmoid activation function

Figure 2-11 illustrates the input-output relationship of a sigmoid activation function. The output of a neuron that has a sigmoid activation function is very smooth and gives nice continuous derivatives, which works well when training a neural network. The output of a sigmoid activation function ranges between 0 and 1. Because of its capability to provide continuous values in the range of 0 to 1, the sigmoid function is generally used to output probability with respect to a given class for a binary classification. The sigmoid activation functions in the hidden layers introduce non-linearity so that the model can learn more complex features.

SoftMax Activation Function

The SoftMax activation function is a generalization of the sigmoid function and is best suited for multi-class classification problems. If there are k output classes and the weight vector for the *ith* class is $w^{(i)}$, then the predicted probability for the *ith* class given the input vetor $x \in \mathbb{R}^{n \times 1}$ is given by the following:

$$P\left(y_i = 1/x\right) = \frac{e^{w^{(i)T}x + b^{(i)}}}{\sum_{j=1}^{k} e^{w^{(j)T}x + b^{(j)}}}$$

where $b^{(i)}$ is the bias term for each output unit of the SoftMax.

Let's try to see the connection between a sigmoid function and a two-class SoftMax function.

Let's say the two classes are y_1 and y_2 and the corresponding weight vectors for them are $w^{(1)}$ and $w^{(2)}$. Also, let the biases for them be $b^{(1)}$ and $b^{(2)}$ respectively. Let's say the class corresponding to $y_1 = 1$ is the positive class.

$$P\left(y_1 = 1/x\right) = \frac{e^{w^{(1)T}x + b^{(1)}}}{e^{w^{(1)T}x + b^{(1)}} + e^{w^{(2)T}x + b^{(2)}}}$$

$$= \frac{1}{1 + e^{-\left(w^{(1)} - w^{(2)}\right)^T x - \left(b^{(1)} - b^{(2)}\right)}}$$

We can see from the preceding expression that the probability of the positive class for a two-class SoftMax has the same expression as that of a sigmoid activation function, the only difference being that in the sigmoid we only use one set of weights, while in the two-class SoftMax there are two sets of weights. In sigmoid activation functions, we don't use different sets of weights for the two different classes, and the set of weight taken is generally the weight of the positive class with respect to the negative class. In SoftMax activation functions, we explicitly take different sets of weights for different classes.

The loss function for the SoftMax layer as represented by Figure 2-12 is called categorical cross entropy and is given by the following:

$$C = \sum_{i=1}^{k} - y_i \log P\left(y_i = 1/x\right)$$

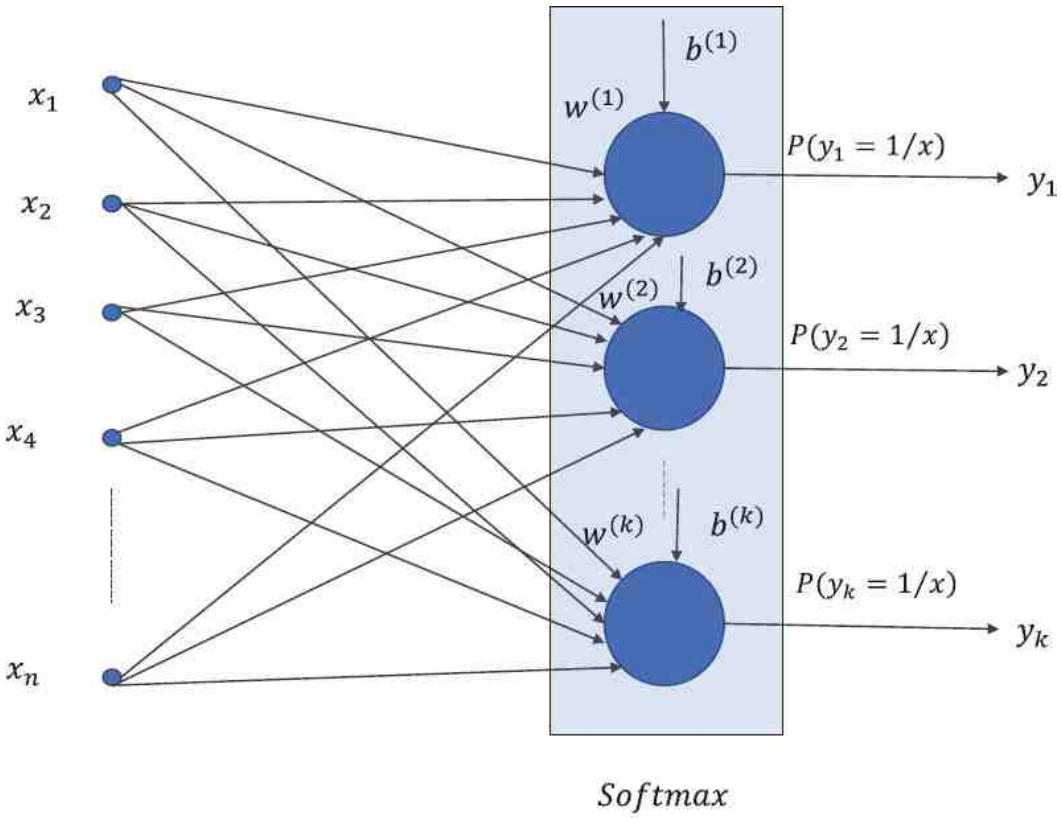

Figure 2-12: SoftMax activation function

Rectified Linear Unit(ReLU) Activation Function

In a rectified linear unit, as shown in Figure 2-13, the output equals the net input to the neuron if the overall input is greater than 0; however, if the overall input is less than or equal to 0 the neuron outputs a 0.

The output for a *ReLU* unit can be represented as follows:

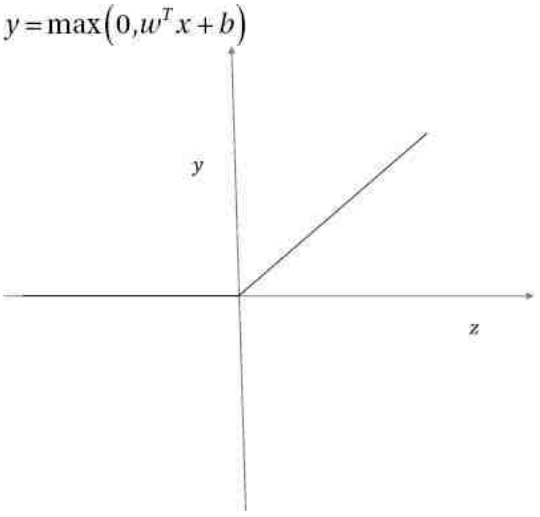

Figure 2-13: Rectified linear unit

The ReLU is one of the key elements that has revolutionized deep learning. They are easier to compute. ReLUs combine the best of both worlds—they have a constant gradient while the net input is positive and zero gradient elsewhere. If we take the sigmoid activation function, for instance, the gradient of the same is almost zero for very large positive and negative values, and hence the neural network might suffer from a vanishing-gradient problem. This constant gradient for positive net input ensures the gradient-descent algorithm doesn't stop learning because of a vanishing gradient. At the same time, the zero output for a non-positive net input renders non-linearity.

There are several versions of rectified linear unit activation functions such as parametric rectified linear unit (PReLU) and leaky rectified linear unit.

For a normal ReLU activation function, the output as well as the gradients are zero for non-positive input values, and so the training can stop because of the zero gradient. For the model to have a non-zero gradient even while the input is negative, PReLU can be useful. The input-output relationship for a PReLU activation function is given by the following:

$$y = \max(0,z) + \beta \min(0,z)$$

where $z = w^T x + b$ is the net input to the PReLU activation function and β is the parameter learned through training.

When β is set to –1, then $y = |z|$ and the activation function is called absolute value ReLU. When β is set to some small positive value, typically around 0.01, then the activation function is called leaky ReLU.

Tanh Activation Function

The input-output relationship for a tanh activation function (see Figure 2-14) is expressed as

$$y = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

Where $z = w^T x + b$ is the net input to the tanh activation function.

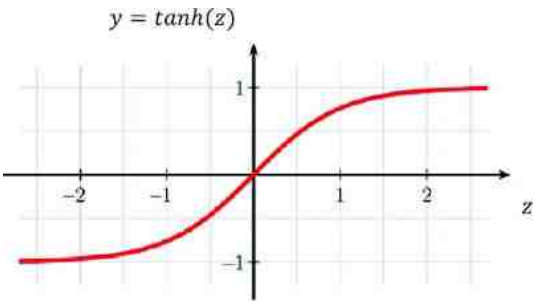

Figure 2-14: Tanh activation function

- When the net input z is a positive large number $e^{-z} \rightarrow 0$ and so $y \rightarrow 1$.
- When the net input z is a negative large number $e^{z} \rightarrow 0$ and so $y \rightarrow -1$.
- When the net input z is 0 then $e^{-z} = 1$ and so $y = 0$.

As we can see, tanh activation functions can output values between -1 and +1.

The sigmoid activation function saturates at around output 0. While training the network, if the outputs in the layer are close to zero the gradient vanishes and training stops. The tanh activation function saturates at -1 and + 1 values for the output and has well-defined gradients at around the 0 value of the output. So, with tanh activation functions such vanishing-gradient problems can be avoided around output 0.

Learning Rule for Multi-Layer Perceptrons Network

In an earlier section, we saw that the Perceptron learning rule can only learn linear decision boundaries. Non-linear complex decision boundaries can be modeled through multi-layer Perceptrons; however, such a model cannot be learned through the Perceptron learning rule. Hence, one would need a different learning algorithm.

In the Perceptron learning rule, the goal is to keep updating the weights of the model till all the training data points have been correctly classified. If there is no such feasible weight vector that classifies all the points correctly, the algorithm doesn't converge. In such cases, the algorithm can be stopped by pre-defining the number of passes (iterations) to train or by defining a threshold for the number of correctly classified training data points after which to stop the training.

For multi-layer Perceptrons and for most of the deep-learning training networks, the best way to train the model is to compute a cost function based on the error of misclassification and then to minimize the cost function with respect to the parameters of the model. Since cost-based learning algorithms minimize the cost function, for binary classifications—generally the log-loss cost function—the negative of the log likelihood function is used. For reference, how the log-loss cost function is derived from maximum likelihood methods has been illustrated in Chapter 1 under "Logistic Regression."

A multi-layer Perceptrons network would have hidden layers, and to learn non-linear decision boundaries the activation functions should be non-linear themselves, such as sigmoid, ReLu, tanh, and so forth. The output neuron for binary classification should have a sigmoid activation function in order to cater to the log-loss cost function and to output probability values for the classes.

Now, with the preceding considerations, let's try to solve the XOR function by building a log-loss cost function and then minimizing it with respect to the weight and bias parameters of the model. All the neurons in the network are taken to have sigmoid activation functions.

Referring to Figure 2-7, let the input and output at hidden unit h_1 be i_1 and z_1 respectively. Similarly, let the input and output at hidden unit h_2 be i_2 and z_2 respectively. Finally, let the input and output at output layer p_1 be i_3 and z_3 respectively.

$$i_1 = w_{11}x_1 + w_{21}x_2 + b_1$$

$$i_2 = w_{12}x_1 + w_{22}x_2 + b_2$$

$$z_1 = 1/\left(1+e^{-i_1}\right)$$

$$z_2 = 1/\left(1+e^{-i_2}\right)$$

$$i_3 = w_1 z_1 + w_2 z_2 + b_3$$

$$z_3 = 1/\left(1+e^{-i_3}\right)$$

Considering the log-loss cost function, the total cost function for the XOR problem can be defined as follows:

$$C = \sum_{i=1}^{4} -y^{(i)} \log z_3^{(i)} - (1-y^{(i)}) \log(1-z_3^{(i)})$$

If all the weights and biases put together can be thought of as a parameter vector θ, we can learn the model by minimizing the cost function $C(\theta)$:

$$\theta^* = \underset{\theta}{\text{Arg Min}} C(\theta)$$

For minima, the gradient of the cost function $C(\theta)$ with respect to θ (i.e., $\nabla C(\theta)$) should be zero. The minima can be reached through gradient-descent methods. The update rule for gradient descent is $\theta^{(t+1)} = \theta^{(t)} - \eta \nabla C(\theta^{(t)})$, where η is the learning rate and $\theta^{(t+1)}$ and $\theta^{(t)}$ are the parameter vectors at iterations $t + 1$ and t respectively.

If we consider individual weight within the parameter vector, the gradient-descent update rule becomes the following:

$$w_k^{(t+1)} = w_k^{(t)} - \eta \frac{\partial C\left(w_k^{(t)}\right)}{\partial w_k} \quad \forall w_k \in \theta$$

The gradient vector would not be as easy to compute as it would be in linear or logistic regression since in neural networks the weights follow a hierarchical order. However, the chain rule of derivatives provides for some simplification to methodically compute the partial derivatives with respect to the weights (including biases).

The method is called *backpropagation* and it provides simplification in gradient computation.

Backpropagation for Gradient Computation

Backpropagation is a useful method of propagating the error at the output layer backward so that the gradients at the preceding layers can be computed easily using the chain rule of derivatives.

Let us consider one training example and work through the backpropagation, taking the XOR network structure into account (see Figure 2-8). Let the input be $x = [x_1\ x_2]^T$ and the corresponding class be y. So, the cost function for the single record becomes the following:

$$C = -y log z_3 - (1-y)\log(1-z_3)$$

$$\frac{\partial C}{\partial w_1} = \frac{dC}{dz_3}\frac{dz_3}{di_3}\frac{\partial i_3}{\partial w_1}$$

$$\frac{dC}{dz_3} = \frac{(z_3 - y)}{z_3(1-z_3)}$$

Now $z_3 = 1/(1+e^{-z_3})$

$$\frac{dz_3}{di_3} = z_3(1-z_3)$$

$$\frac{dC}{di_3} = \frac{dC}{dz_3}\frac{dz_3}{di_3} = \frac{(z_3 - y)}{z_3(1-z_3)} z_3(1-z_3) = (z_3 - y)$$

As we can see, the derivative of the cost function with respect to the net input in the final layer is nothing but the error in estimating the output *($z_3 - y$)* :

$$\frac{\partial i_3}{\partial w_1} = z_1$$

$$\frac{\partial C}{\partial w_1} = \frac{dC}{dz_3}\frac{dz_3}{di_3}\frac{\partial i_3}{\partial w_1} = (z_3 - y) z_1$$

Similarly,

$$\frac{\partial C}{\partial w_2} = \frac{dC}{dz_3}\frac{dz_3}{di_3}\frac{\partial i_3}{\partial w_2} = (z_3 - y) z_2$$

$$\frac{\partial C}{\partial b_3} = \frac{dC}{dz_3}\frac{dz_3}{di_3}\frac{\partial i_3}{\partial b_3} = (z_3 - y)$$

Now, let's compute the partial derivatives of the cost function with respect to the weights in the previous layer:

$$\frac{\partial C}{\partial z_1} = \frac{dC}{dz_3}\frac{dz_3}{di_3}\frac{\partial i_3}{\partial z_1} = (z_3 - y) w_1$$

$\frac{\partial C}{\partial z_1}$ can be treated as the error with respect to the output of the hidden layer unit h_1. The error is propagated in proportion to the weight that is joining the output unit to the hidden layer unit. If there were multiple output units then $\frac{\partial C}{\partial z_1}$ would have a contribution from each of the output units. We will see this in detail in the next section.

Similarly,

$$\frac{\partial C}{\partial i_1} = \frac{dC}{dz_3}\frac{dz_3}{di_3}\frac{\partial i_3}{\partial z_1}\frac{dz_1}{di_1} = (z_3 - y) w_1 z_1 (1 - z_1)$$

$\frac{\partial C}{\partial i_1}$ can be considered the error with respect to the net input of the hidden layer unit h_1. It can be computed by just multiplying the z_1 (1 – z_1) factor by $\frac{\partial C}{\partial z_1}$:

$$\frac{\partial C}{\partial w_{11}} = \frac{dC}{dz_3}\frac{dz_3}{di_3}\frac{\partial i_3}{\partial z_1}\frac{dz_1}{di_1}\frac{\partial i_i}{\partial w_{11}} = (z_3 - y)w_1 z_1(1 - z_1)x_1$$

$$\frac{\partial C}{\partial w_{21}} = \frac{dC}{dz_3}\frac{dz_3}{di_3}\frac{\partial i_3}{\partial z_1}\frac{dz_1}{di_1}\frac{\partial i_i}{\partial w_{21}} = (z_3 - y)w_1 z_1(1 - z_1)x_2$$

$$\frac{\partial C}{\partial b_1} = \frac{dC}{dz_3}\frac{dz_3}{di_3}\frac{\partial i_3}{\partial z_1}\frac{dz_1}{di_1}\frac{\partial i_i}{\partial w_{21}} = (z_3 - y)w_1 z_1(1 - z_1)$$

Once we have the partial derivative of the cost function with respect to the input in each neuron unit, we can compute the partial derivative of the cost function with respect to the weight contributing to the input—we just need to multiply the input coming through that weight.

Generalizing the Backpropagation Method for Gradient Computation

In this section, we try to generalize the backpropagation method through a more complicated network. We assume the final output layer is composed of three independent sigmoid output units, as depicted in Figure 2-15. Also, we assume that the network has a single record for ease of notation and for simplification in learning.

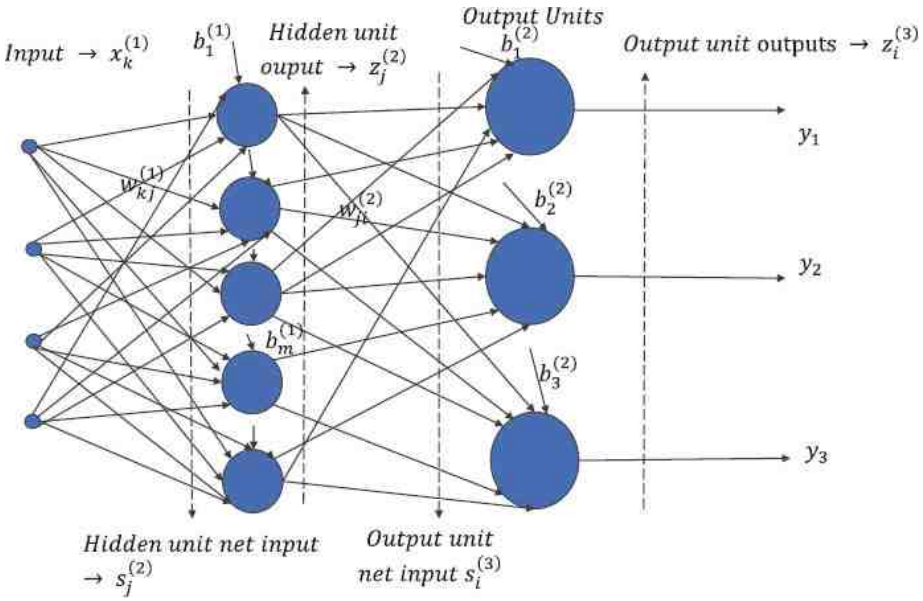

Figure 2-15: Network to illustrate backpropagation for independent sigmoid output layers

The cost function for a single input record is given by the following:

$$C = \sum_{i=1}^{3} -y_i \log P(y_i = 1) - (1 - y_i)\log\left(1 - P(y_i = 1)\right)$$

$$= \sum_{i=1}^{3} -y_i \log z_i^{(3)} - (1 - y_i)\log\left(1 - z_i^{(3)}\right)$$

In the preceding expression, $y_i \in \{0,1\}$, depending on whether the event specific to y_i is active or not. $P(y_i = 1) = z_i^{(3)}$ denotes the predicted probability of the *ith* class.

Let's compute the partial derivative of the cost function with respect to the weight $w_{ji}^{(2)}$. The weight would only impact the output of the *ith* output unit of the network.

$$\frac{\partial C}{\partial w_{ji}^{(2)}}=\frac{\partial C}{\partial z_i^{(3)}}\frac{\partial z_i^{(3)}}{\partial s_i^{(3)}}\frac{\partial s_i^{(3)}}{\partial w_{ji}^{(2)}}$$

$$\frac{\partial C}{\partial z_i^{(3)}}=\frac{\left(z_i^{(3)}-y_i\right)}{z_i^{(3)}\left(1-z_i^{(3)}\right)}$$

$$P\left(y_i=1\right)=z_i^{(3)}=1/\left(1+e^{-s_i^{(3)}}\right)$$

$$\frac{\partial z_i^{(3)}}{\partial s_i^{(3)}}=z_i^{(3)}\left(1-z_i^{(3)}\right)$$

$$\frac{\partial C}{\partial s_i^{(3)}}=\frac{\partial C}{\partial z_i^{(3)}}\frac{\partial z_i^{(3)}}{\partial s_i^{(3)}}=\frac{\left(z_i^{(3)}-y_i\right)}{z_i^{(3)}\left(1-z_i^{(3)}\right)}z_i^{(3)}\left(1-z_i^{(3)}\right)=\left(z_i^{(3)}-y_i\right)$$

So, as before, the partial derivative of the cost function with respect to the net input for the *ith* output unit is $\left(z_i^{(3)}-y_i\right)$, which is nothing but the error in prediction at the *ith* output unit.

$$\frac{\partial s_i^{(3)}}{\partial w_{ji}^{(2)}}=z_j^{(2)}$$

Combining $\frac{\partial C}{\partial s_i^{(3)}}$ and $\frac{\partial s_i^{(3)}}{\partial w_{ji}^{(2)}}$, we get,

$$\frac{\partial C}{\partial w_{ji}^{(2)}}=\left(z_i^{(3)}-y_i\right)z_j^{(2)}$$

$$\frac{\partial C}{\partial b_i^{(2)}}=\left(z_i^{(3)}-y_i\right)$$

The preceding gives a generalized expression for partial derivatives of the cost function with respect to weights and biases in the last layer of the network. Next, let's compute the partial derivative of the weights and biases in the lower layers. Things get a little more involved but still follow a generalized trend. Let's compute the partial derivative of the cost function with respect to the weight $w_{kj}^{(1)}$. The weight would be impacted by the errors at all three output units. Basically, the error at the output of the *jth* unit in the hidden layer would have an error contribution from all output units, scaled by the weights connecting the output layers to the *jth* hidden unit. Let's compute the partial derivative just by the chain rule and see if it lives up to what we have proclaimed:

$$\frac{\partial C}{\partial w_{kj}^{(1)}}=\frac{\partial C}{\partial z_j^{(2)}}\frac{\partial z_j^{(2)}}{\partial s_j^{(2)}}\frac{\partial s_j^{(2)}}{\partial w_{kj}^{(1)}}$$

$$\frac{\partial s_j^{(2)}}{\partial w_{kj}^{(1)}}=z_k^{(1)}$$

$$\frac{\partial z_j^{(2)}}{\partial s_j^{(2)}}=z_j^{(2)}\left(1-z_j^{(2)}\right)$$

Now, $\frac{\partial C}{\partial z_j^{(2)}}$ is the tricky computation since $z_j^{(2)}$ influences all three output units:

$$\frac{\partial C}{\partial z_j^{(2)}} = \sum_{i=1}^{3} \frac{\partial C}{\partial z_i^{(3)}} \frac{\partial z_i^{(3)}}{\partial s_i^{(3)}} \frac{\partial s_i^{(3)}}{\partial z_j^{(2)}}$$

$$= \sum_{i=1}^{3} \left(z_i^{(3)} - y_i \right) w_{ji}^{(2)}$$

Combining the expressions for $\frac{\partial s_j^{(2)}}{\partial w_{kj}^{(1)}}, \frac{\partial z_j^{(2)}}{\partial s_j^{(2)}}$ and $\frac{\partial C}{\partial z_j^{(2)}}$, we have

$$\frac{\partial C}{\partial w_{kj}^{(1)}} = \sum_{i=1}^{3} \left(z_i^{(3)} - y_i \right) w_{ji}^{(2)} z_j^{(2)} \left(1 - z_j^{(2)} \right) x_k^{(1)}$$

In general, for a multi-layer neural network to compute the partial derivative of the cost function *C* with respect to a specific weight *w* contributing to the net input *s* in a neuron unit, we need to compute the partial derivative of the cost function with respect to the net input (i.e., $\frac{\partial C}{\partial s}$) and then multiply the input *x* associated with the weight *w*, as follows:

$$\frac{\partial C}{\partial w} = \frac{\partial C}{\partial s} \frac{\partial s}{\partial w} = \frac{\partial C}{\partial s} x$$

$\frac{\partial C}{\partial s}$ can be thought of as the error at the neural unit and can be computed iteratively by passing the error at the output layer to the neural units in the lower layers. Another point to note is that an error in a higher-layer neural unit is distributed to the output of the preceding layer's neural units in proportion to the weight connections between them. Also, the partial derivative of the cost function with respect to the net input to a sigmoid activation neuron $\frac{\partial C}{\partial s}$ can be computed from the partial derivative of the cost function with respect to the output *z* of a neuron (i.e., $\frac{\partial C}{\partial z}$) by multiplying $\frac{\partial C}{\partial z}$ by $z(1 - z)$. For linear neurons, this multiplying factor becomes 1.

All these properties of neural networks make computing gradients easy. This is how a neural network learns in each iteration through backpropagation.

Each iteration is composed of a forward pass and a backward pass, or backpropagation. In the forward pass, the net input and output at each neuron unit in each layer are computed. Based on the predicted output and the actual target values, the error is computed in the output layers. The error is backpropagated by combining it with the neuron outputs computed in the forward pass and with existing weights. Through backpropagation the gradients get computed iteratively. Once the gradients are computed, the weights are updated by gradient-descent methods.

Please note that the deductions shown are valid for sigmoid activation functions. For other activation functions, while the approach remains the same, changes specific to the activation functions are required in the implementation.

The cost function for a SoftMax function is different from that for the independent multi-class classification.

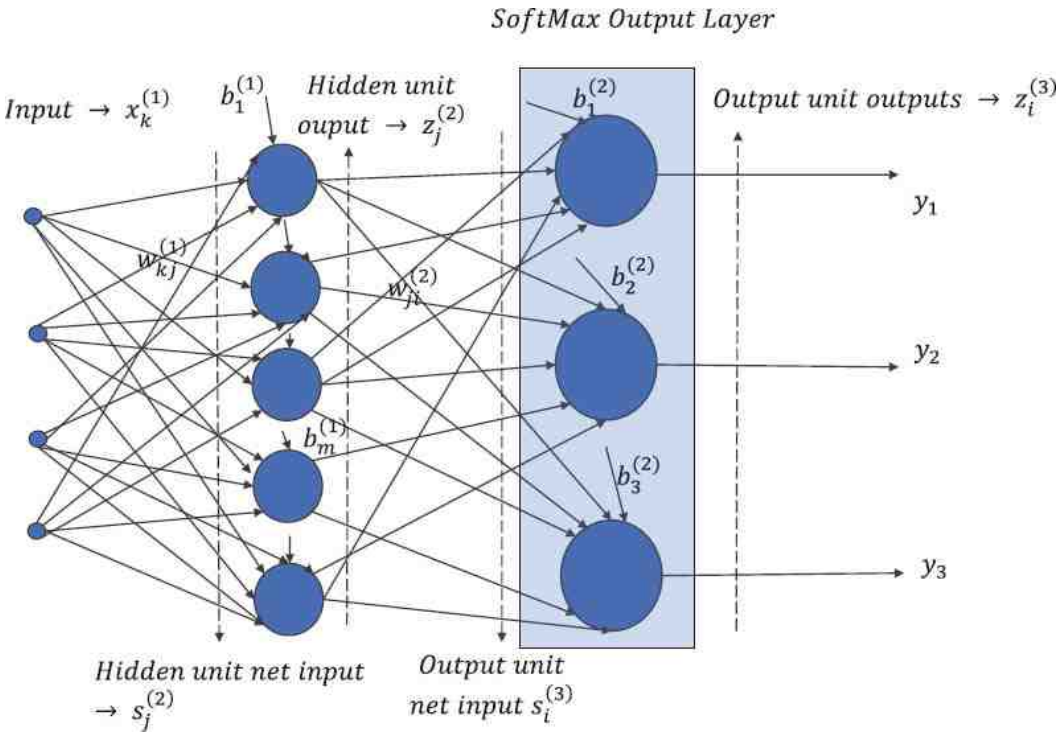

Figure 2-16: Network to illustrate backpropagation for Softmax output layer

The cross-entropy cost for the SoftMax activation layer in the network represented in Figure 2-16 is given by

$$C=\sum_{i=1}^{3}-y_i\log P(y_i=1)=\sum_{i=1}^{3}-y_i\log z_i^{(3)}$$

Let's compute the partial derivative of the cost function with respect to the weight $w_{ji}^{(2)}$. Now, the weight would impact the net input $s_i^{(3)}$ to the *ith* SoftMax unit. However, unlike the independent binary activations in the earlier network, here all three SoftMax output units $z_k^{(3)}\ \forall k\in\{1,,2,,3\}$ would be influenced by $s_i^{(3)}$ since

$$z_k^{(3)}=\frac{e^{s_k^{(3)}}}{\sum_{l=1}^{3}e^{s_l^{(3)}}}=\frac{e^{s_k^{(3)}}}{\sum_{l\neq i}e^{s_l^{(3)}}+e^{s_i^{(3)}}}$$

Hence, the derivative $\frac{\partial C}{\partial w_{ji}^{(2)}}$ can be written as follows:

$$\frac{\partial C}{\partial w_{ji}^{(2)}}=\frac{\partial C}{\partial s_i^{(3)}}\frac{\partial s_i^{(3)}}{\partial w_{ji}^{(2)}}$$

Now, as just stated, since $s_i^{(3)}$ influences all the outputs $z_k^{(3)}$ in the SoftMax layer,

$$\frac{\partial C}{\partial s_i^{(3)}}=\sum_{k=1}^{3}\frac{\partial C}{\partial z_k^{(3)}}\frac{\partial z_k^{(3)}}{\partial s_i^{(3)}}$$

The individual components of the partial derivative are as follows:

$$\frac{\partial C}{\partial z_k^{(3)}}=\frac{-y_k}{z_k^{(3)}}$$

For *K* = *i*, $\frac{\partial z_k^{(3)}}{\partial s_i^{(3)}}=z_i^{(3)}\left(1-z_i^{(3)}\right)$

For $K \neq i$, $\frac{\partial z_k^{(3)}}{\partial s_i^{(3)}} = -z_i^{(3)} z_k^{(3)}$

$$\frac{\partial s_i^{(3)}}{\partial w_{ji}^{(2)}} = z_j^{(2)}$$

$$\frac{\partial C}{\partial s_i^{(3)}} = \sum_{k=1}^{3} \frac{\partial C}{\partial z_k^{(3)}} \frac{\partial z_k^{(3)}}{\partial s_i^{(3)}} = \sum_{k=i} \frac{\partial C}{\partial z_k^{(3)}} \frac{\partial z_k^{(3)}}{\partial s_i^{(3)}} + \sum_{k \neq i} \frac{\partial C}{\partial z_k^{(3)}} \frac{\partial z_k^{(3)}}{\partial s_i^{(3)}}$$
$$= \frac{-y_i}{z_i^{(3)}} z_i^{(3)} \left(1 - z_i^{(3)}\right) + \sum_{k \neq i} \frac{-y_k}{z_k^{(3)}} \left(-z_i^{(3)} z_k^{(3)}\right)$$

$$= -y_i \left(1 - z_i^{(3)}\right) + z_i^{(3)} \sum_{k \neq i} y_k$$

$$= -y_i + y_i z_i^{(3)} + z_i^{(3)} \sum_{k \neq i} y_k$$

$$= -y_i + z_i^{(3)} \sum_{k} y_k$$

$$= -y_i + z_i^{(3)} \; Since\ only\ one\ of\ the\ y_k\ can\ be\ 1 \therefore \sum_k y_k = 1$$

$$= \left(z_i^{(3)} - y_i\right)$$

As it turns out, the cost derivative with respect to the net input to the *ith* SoftMax unit is the error in predicting the output at the *ith* SoftMax output unit. Combining $\frac{\partial C}{\partial s_i^{(3)}}$ and $\frac{\partial s_i^{(3)}}{\partial w_{ji}^{(2)}}$, we get the following:

$$\frac{\partial C}{\partial w_{ji}^{(2)}} = \frac{\partial C}{\partial s_i^{(3)}} \frac{\partial s_i^{(3)}}{\partial w_{ji}^{(2)}} = \left(z_i^{(3)} - y_i\right) z_j^{(2)}$$

Similarly, for the bias term to the *ith* SoftMax output unit we have the following:

$$\frac{\partial C}{\partial b_i^{(2)}} = \left(z_i^{(3)} - y_i\right)$$

Computing the partial derivative of the cost function with respect to the weight $w_{kj}^{(1)}$ in the previous layer, i.e., $\frac{\partial C}{\partial w_{kj}^{(1)}}$, would have the same form as that in the case of a network with independent binary classes.

This is obvious since the networks only differ in terms of the output units' activation functions, and even then the expressions that we get for $\frac{\partial C}{\partial s_i^{(3)}}$ and $\frac{\partial s_i^{(3)}}{\partial w_{ji}^{(2)}}$ remain the same. As an exercise, interested readers can verify whether $\frac{\partial C}{\partial w_{kj}^{(1)}} = \sum_{i=1}^{3} \left(z_i^{(3)} - y_i\right) w_{ji}^{(2)} z_j^{(2)} \left(1 - z_j^{(2)}\right) x_k^{(1)}$ still holds true.

Deep Learning Versus Traditional Methods

In this book, we will use TensorFlow from Google as the deep-learning library since it has several advantages. Before moving on to TensorFlow, let's look at some of the key advantages of deep learning and a few of its shortcomings if it is not used in the right place.

- Deep learning outperforms traditional machine-learning methods by a huge margin in several domains, especially in the fields of computer vision, speech recognition, natural language processing, and time series.

- With deep learning, more and more complex features can be learned as the layers in the deep-learning neural network increase. Because of this automatic feature-learning property, deep learning reduces the feature-engineering time, which is a time-consuming activity in traditional machine-learning approaches.
- Deep learning works best for unstructured data, and there is a plethora of unstructured data in the form of images, text, speech, sensor data, and so forth, which when analyzed would revolutionize different domains, such as health care, manufacturing, banking, aviation, e-commerce, and so on.

A few limitations of deep learning are as follows:

- Deep learning networks generally tend to have a lot of parameters, and for such implementations there should be a sufficiently large volume of data to train. If there are not enough data, deep-learning approaches will not work well since the model will suffer from overfitting.
- The complex features learned by the deep-learning network are often hard to interpret.
- Deep-learning networks require a lot of computational power to train because of the large number of weights in the model as well as the data volume.

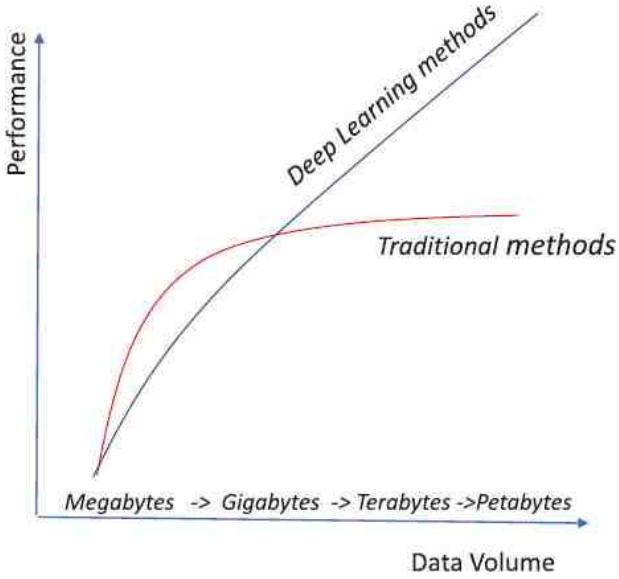

Figure 2-17: Performance comparison of traditional methods versus deep-learning methods

When the data volume is less, traditional methods tend to perform better that deep-learning ones. However, when the data volume is huge, the deep-learning method wins over traditional methods by a huge margin, which has been roughly depicted in Figure 2-17.

TensorFlow

TensorFlow from Google is an open source library that primarily focuses on deep learning. It uses computational data-flow graphs to represent complicated neural-network architecture. The nodes in the graph denote mathematical computations, also called ops (operations), whereas the edges denote the data tensors transferred between them. Also, the relevant gradients are stored at each node of the computational graph, and during backpropagation these are combined to get the gradients with respect to each weight. Tensors are multi-dimensional data arrays used by TensorFlow.

Common Deep-Learning Packages

The common deep-learning packages are as follows:

- ***Torch*** – A scientific computing framework with underlying C implementation and LuaJIT as the scripting language. Initial release of Torch was in 2002. Operating systems on which Torch works are Linux, Android, Mac OS X, and iOS. Reputed organizations such as Facebook AI Research and IBM use Torch. Torch can utilize GPU for fast computation.
- ***Theano*** – Is a deep-learning package in Python that is primarily used for computationally intensive research-oriented activities. It is tightly integrated with Numpy array and has efficient symbolic differentiators. It also provides transparent use of GPU for much faster computation.
- ***Caffe*** – Deep-learning framework developed by Berkeley AI Research (BAIR). Speed makes Caffe perfect for research experiments and industry deployment. Caffe implementation can use GPU very efficiently.
- ***CuDNN*** – CuDNN stands for CUDA Deep Neural Network library. It provides a library of primitives for GPU implementation of deep

neural networks.

- ***TensorFlow*** – Open source deep-learning framework from Google inspired by Theano. TensorFlow is slowly becoming the preferred library for deep learning in research-oriented work as well as for production implementation. Also for distributed production implementation over the cloud, TensorFlow is becoming the go-to library.
- ***MxNet*** – Open source deep-learning framework that can scale to multiple GPUs and machines. Supported by major cloud providers such as AWS and Azure. Popular machine-learning library GraphLab has good deep-learning implementation using MxNet.
- ***deeplearning4j*** – Open source distributed deep-learning framework for Java virtual machines.

A few salient features of these deep-learning frameworks are as follows:

- Python is the high-level language of choice for TensorFlow and Theano, whereas Lua is the high-level language of choice for Torch. MxNet also has Python APIs.
- TensorFlow and Theano are very similar in nature. TensorFlow has better support for distributed systems. Theano is an academic project while TensorFlow is funded by Google.
- TensorFlow, Theano, MxNet, and Caffe all use automatic differentiators while Torch uses AutoGrad. Automatic differentiators are different from symbolic differentiation and numeric differentiation. Automatic differentiators are very efficient when used in neural networks because of the backpropagation method of learning that utilizes the chain rule of differentiation.
- For production implementation on the cloud, TensorFlow is on its way to becoming the go-to platform for applications targeting large distributed systems.

TensorFlow Installation

TensorFlow can be installed with ease in Linux-, Mac OS-, and Windows-based machines. It is always preferable to create separate environments for TensorFlow. One of the things to note is that TensorFlow installation in Windows requires your Python version to be greater than or equal to 3.5. Such limitations don't exist for Linux-based machines, or for Mac OS, for that matter. The details of installation for Windows-based machines are documented well on the official website for TensorFlow: https://www.tensorflow.org/install/install_windows. The installation links for Linux-based machines and Mac OS are:

https://www.tensorflow.org/install/install_linux

https://www.tensorflow.org/install/install_mac

TensorFlow Basics for Development

TensorFlow has its own format of commands to define and manipulate tensors. Also, TensorFlow executes the computational graphs within activated sessions. Listings 2-1 to 2-15 are a few of the basic TensorFlow commands used to define tensors and TensorFlow variables and to execute TensorFlow computational graphs within sessions.

Listing 2-1: Import TensorFlow and Numpy Library

```
import tensorflow as tf
import numpy as np
```

Listing 2-2: Activate a TensorFlow Interactive Session

```
tf.InteractiveSession()
```

Listing 2-3: Define Tensors

```
a = tf.zeros((2,2));
b = tf.ones((2,2))
```

Listing 2-4: Sum the Elements of the Matrix (2D Tensor) Across the Horizontal Axis

```
tf.reduce_sum(b,reduction_indices = 1).eval()

-- output --
array([ 2.,  2.], dtype=float32)
```

To run TensorFlow commands in interactive mode, the `Interactive Session()` command can be invoked as in Listing 2-2, and by using the `eval()` method the TensorFlow commands can be run under the activated interactive session as shown in Listing 2-4.

Listing 2-5: Check the Shape of the Tensor

```
a.get_shape()

-- output --

TensorShape([Dimension(2), Dimension(2)])
```

Listing 2-6: Reshape a Tensor

```
tf.reshape(a,(1,4)).eval()

-- output --

array([[ 0.,  0.,  0.,  0.]], dtype=float32)
```

Listing 2-7: Explicit Evaluation in TensorFlow and Difference with Numpy

```
ta = tf.zeros((2,2))
print(ta)

-- output --

Tensor("zeros_1:0", shape=(2, 2), dtype=float32)

print(ta.eval())

-- output --

[[ 0.  0.]
 [ 0.  0.]]

a = np.zeros((2,2))
print(a)

-- output --

[[ 0.  0.]
 [ 0.  0.]]
```

Listing 2-8: Define TensorFlow Constants

```
a = tf.constant(1)
b = tf.constant(5)
c= a*b
```

Listing 2-9: TensorFlow Session for Execution of the Commands Through Run and Eval

```
with tf.Session() as sess:
    print(c.eval())
    print(sess.run(c))

-- output --

5
5
```

Listing 2-10a: Define TensorFlow Variables

```
w = tf.Variable(tf.ones(2,2),name='weights')
```

Listing 2-10b: Initialize the Variables After Invoking the Session

```
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    print(sess.run(w))

-- output --

[[ 1.  1.]
 [ 1.  1.]]
```

A TensorFlow session is generally activated through `tf.Session() as` shown in Listing 2-10b, and the computational graph operations(ops) are executed under the activated session.

Listing 2-11a: Define the TensorFlow Variable with Random Initial Values from Standard Normal Distribution

```
rw = tf.Variable(tf.random_normal((2,2)),name='random_weights')
```

Listing 2-11b: Invoke Session and Display the Initial State of the Variable

```
with tf.Session()as sess:
    sess.run(tf.global_variables_initializer())
    print(sess.run(rw))

-- output --

[[ 0.37590656 -0.11246648]
 [-0.61900514 -0.93398571]]
```

As shown in Listing 2-11b, the `run` method is used to execute the computational operations (ops) within an activated session, and `tf.global_variables_initializer()` when `run` initializes the TensorFlow variables defined. The random variable `rw` defined in 2-11a got initialized in Listing 2-11b.

Listing 2-12: TensorFlow Variable State Update

```
var_1 = tf.Variable(0,name='var_1')
add_op =  tf.add(var_1,tf.constant(1))
upd_op = tf.assign(var_1,add_op)
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    for i in xrange(5):
        print(sess.run(upd_op))

-- output --

1
2
3
4
5
```

Listing 2-13: Display the TensorFlow Variable State

```
x = tf.constant(1)
y = tf.constant(5)
z = tf.constant(7)

mul_x_y = x*y
final_op = mul_x_y + z

with tf.Session() as sess:
    print(sess.run([mul_x_y,final_op]))

-- output --

      5 12
```

Listing 2-14: Convert a Numpy Array to Tensor

```
a = np.ones((3,3))
b = tf.convert_to_tensor(a)
with tf.Session() as sess:
    print(sess.run(b))

-- output --

[[ 1.  1.  1.]
 [ 1.  1.  1.]
 [ 1.  1.  1.]]
```

Listing 2-15: Placeholders and Feed Dictionary

```
inp1 = tf.placeholder(tf.float32,shape=(1,2))
inp2 = tf.placeholder(tf.float32,shape=(2,1))
output = tf.matmul(inp1,inp2)
with tf.Session() as sess:
    print(sess.run([output],feed_dict={inp1:[[1.,3.]],inp2:[[1],[3]]}))

-- output --
```

```
[array([[ 10.]], dtype=float32)]
```

A TensorFlow placeholder defines a variable, the data for which would be assigned at a later point in time. The data is generally passed to the placeholder through the `feed_dict` while running the ops involving the TensorFlow placeholder. This has been illustrated in Listing 2-15.

Gradient-Descent Optimization Methods from a Deep-Learning Perspective

Before we dive into the TensorFlow optimizers, it's important to understand a few key points regarding full-batch gradient descent and stochastic gradient descent, including their shortcomings, so that one can appreciate the need to come up with variants of these gradient-based optimizers.

Elliptical Contours

The cost function for a linear neuron with a least square error is quadratic. When the cost function is quadratic, the direction of the gradient resulting from the full-batch gradient-descent method gives the best direction for cost reduction in a linear sense, but it doesn't point to the minimum unless the different elliptical contours of the cost function are circles. In cases of long elliptical contours, the gradient components might be large in directions where less change is required and small in directions where more change is required to move to the minimum point.

As we can see in Figure 2-18, the gradient at *S* doesn't point to the direction of the minimum; i.e., point *M*. The problem with this condition is that if we take small steps by making the learning rate small then the gradient descent would take a while to converge, whereas if we were to use a big learning rate, the gradients would change direction rapidly in directions where the cost function had curvature, leading to oscillations. The cost function for a multi-layer neural network is not quadratic but rather is mostly a smooth function. Locally, such non-quadratic cost functions can be approximated by quadratic functions, and so the problems of gradient descent inherent to elliptical contours still prevail for non-quadratic cost functions.

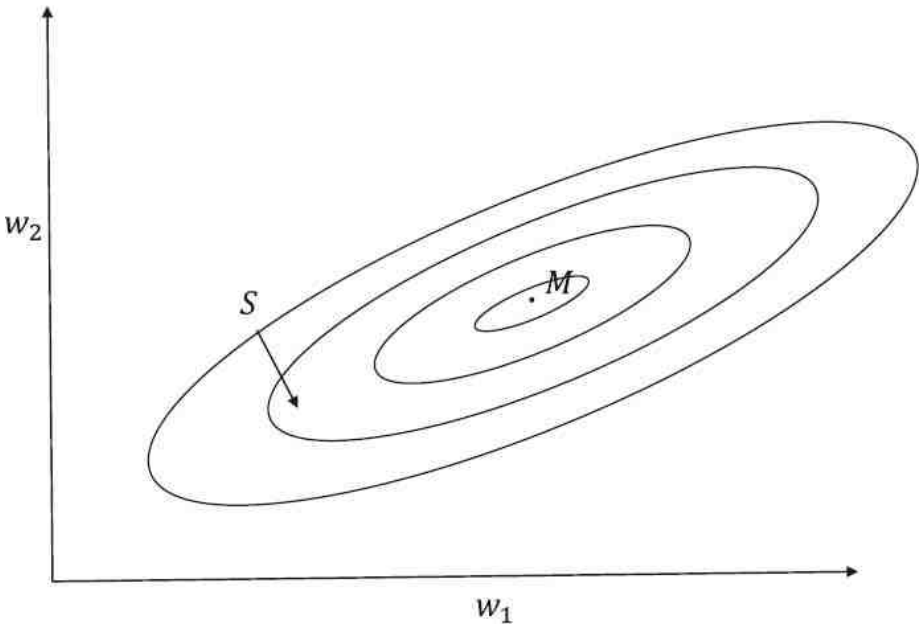

Figure 2-18: Contour plot for a quadratic cost function with elliptical contours

The best way to get around this problem is to take larger steps in those directions in which the gradients are small but consistent and take smaller steps in those directions that have big but inconsistent gradients. This can be achieved if, instead of having a fixed learning rate for all dimensions, we have a separate learning rate for each dimension.

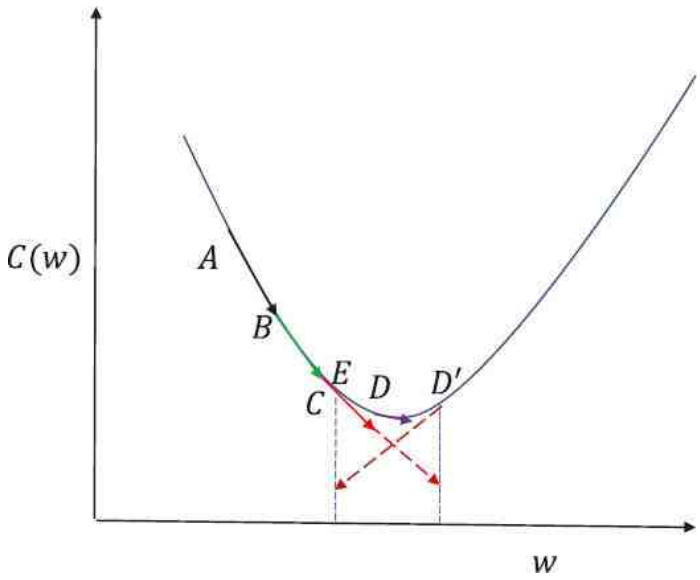

Figure 2-19: Gradient descent for a cost function with one variable

In Figure 2-19, the cost function between *A* and *C* is almost linear, and so gradient descent works well. However, from point *C* the curvature of the cost function takes over, and so the gradient at *C* is not able to keep up with the direction of the change in cost function. Based on the gradient, if we take a small learning rate at *C* we will end up at D, which is reasonable enough since it doesn't overshoot the point of minima. However, a larger step size at *C* will get us to *D'*, which is not desirable, because it's on the other side of the minima. Again, a large step size at *D'* would get us to E, and if the learning rate is not reduced the algorithm tends to toggle between points on either side of the minima, leading to oscillations. When this happens, one way to stop it and achieve convergence is to look at the sign of the gradient $\frac{\partial C}{\partial w}$ or $\frac{dC}{dw}$ in successive iterations, and if they have opposite signs, reduce the learning rate so that the oscillations are reduced. Similarly, if the successive gradients have the same sign then the learning rate can be increased accordingly. When the cost function is a function of multiple weights, the cost function might have curvatures in some dimensions of the weights while it might be linear along other dimensions. Hence, for multivariate cost functions, the partial derivative of the cost function with respect to each weight $\left(\frac{\partial C}{\partial w_i}\right)$ can be similarly analyzed to update the learning rate for each weight or dimension of the cost function.

Non-convexity of Cost Functions

The other big problem with neural networks is that the cost functions are mostly non-convex and so the gradient-descent method might get stuck at local minimum points, leading to a sub-optimal solution. The non-convex nature of the neural network is the result of the hidden layer units that have non-linear activation functions, such as sigmoid. Full-batch gradient descent uses the full dataset for the gradient computation. While this is good for convex cost surfaces, it has its own problems in cases of non-convex cost functions. For non-convex cost surfaces with full-batch gradients, the model is going to end up with the minima in its basin of attraction. If the initialized parameters are in the basin of attraction of a local minima that doesn't provide good generalization, a full-batch gradient would give a sub-optimal solution.

With stochastic gradient descent, the noisy gradients computed may force the model out of the basin of attraction of the bad local minima —one that doesn't provide good generalization—and place it in a more optimal region. Stochastic gradient descent with single data points produces very random and noisy gradients. Gradients with mini batches tend to produce much more stable estimates of gradients when compared to gradients of single data points, but they are still noisier than those produced by the full batches. Ideally, the mini-batch size should be carefully chosen such that the gradients are noisy enough to avoid or escape bad local minima points but stable enough to converge at global minima or a local minimum that provides good generalization.

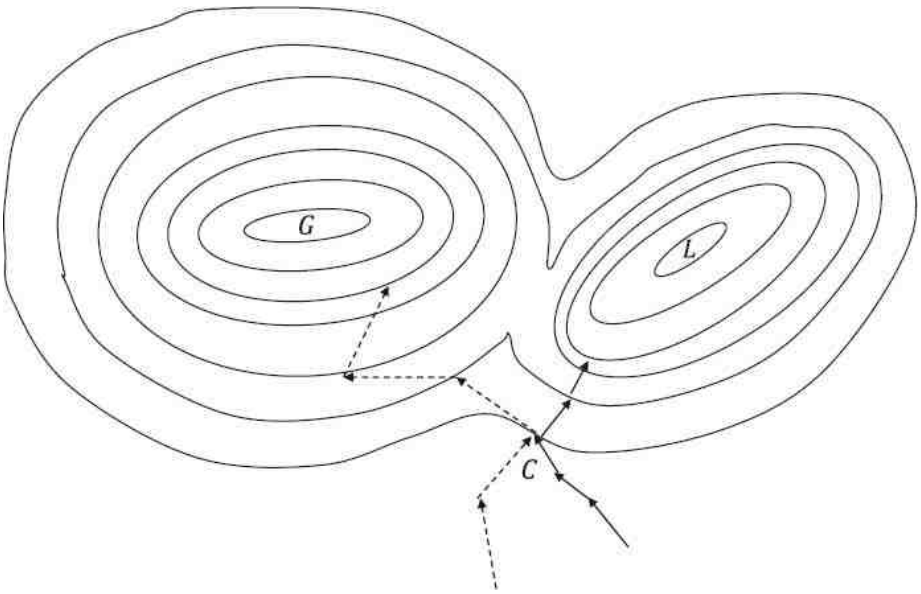

Figure 2-20: Contour plot showing basins of attraction for global and local minima and traversal of paths for gradient descent and stochastic gradient descent

In Figure 2-20, the dotted arrows correspond to the path taken by stochastic gradient descent (SGD), while the continuous arrows correspond to the path taken by full-batch gradient descent. Full-batch gradient descent computes the actual gradient at a point, and if it is in the basin of attraction of a poor local minimum, gradient descent almost surely ensures that the local minima *L* is reached. However, in the case of stochastic gradient descent, because the gradient is based on only portion of the data and not on the full batch, the gradient direction is only a rough estimate. Since the noisy rough estimate doesn't always point to the actual gradient at the point C, stochastic gradient descent may escape the basin of attraction of the local minima and fortunately land in the basin of a global minima. Stochastic gradient descent may escape the global minima basin of attraction too, but generally if the basin of attraction is large and the mini-batch size is carefully chosen so that the gradients it produces are moderately noisy, stochastic gradient descent is most likely to reach the global minima *G* (as in this case) or some other optimal minima that has a large basin of attraction. For non-convex optimization, there are other heuristics as well, such as momentum, which when adopted along with stochastic gradient descent increases the chances of the SGD's avoiding shallow local minima. Momentum generally keeps track of the previous gradients through the velocity component. So, if the gradients are steadily pointing toward a good local minimum that has a large basin of attraction, the velocity component would be high in the direction of the good local minimum. If the new gradient is noisy and points toward a bad local minimum, the velocity component would provide momentum to continue in the same direction and not get influenced by the new gradient too much.

Saddle Points in the High-Dimensional Cost Functions

Another impediment to optimizing non-convex cost functions is the presence of saddle points. The number of saddle points increases exponentially with the dimensionality increase of the parameter space of a cost function. Saddle points are stationary points (i.e., points where the gradient is zero) but are neither a local minimum nor a local maximum point. Since the saddle points are associated with a long plateau of points with the same cost as that of the saddle point, the gradient in the plateau region is either zero or very close to zero. Because of this near-zero gradient in all directions, gradient-based optimizers have a hard time coming out of these saddle points. Mathematically, to determine whether a point is a saddle point the Eigen values of the Hessian matrix of the cost function must be computed at the given point. If there are both positive and negative Eigen values, then it is a saddle point. Just to refresh our memory of local and global minima tests, if all the Eigen values of the Hessian matrix are positive at a stationary point then the point is a global minimum, whereas if all the Eigen values of the Hessian matrix are negative at the stationary point then the point is a global maximum. The Eigen vectors of the Hessian matrix for a cost function give the direction of change in the curvature of the cost function, whereas the Eigen values denote the magnitude of the curvature changes along those directions. Also, for cost functions with continuous second derivatives, the Hessian matrix is symmetrical and hence would always produce an orthogonal set of Eigen vectors, thus giving mutually orthogonal directions for cost curvature changes. If in all such directions given by Eigen vectors the values of the curvature changes (Eigen values) are positive, then the point must be a local minimum, whereas if all the values of curvature changes are negative, then the point is a local maximum. This generalization works for cost functions with any input dimensionality, whereas the determinant rules for determining extremum points varies with the dimensionality of the input to the cost function. Coming back to saddle points, since the Eigen values are positive for some directions but negative for other directions, the curvature of the cost function increases in the direction of positive Eigen values while decreasing in the direction of Eigen vectors with negative co-efficients. This nature of the cost surface around a saddle point generally leads to a region of long plateau with a near-to-zero gradient and makes it tough for gradient-descent methods to escape the plateau of this low gradient. The point (0, 0) is a saddle point for the function $f(x,y) = x^2 - y^2$ as we can see from the following evaluation below:

$$\nabla f(x,y) = 0 \Rightarrow \frac{\partial f}{\partial x} = 0 \text{ and } \frac{\partial f}{\partial y} = 0$$

$$\frac{\partial f}{\partial x} = 2x = 0 \Rightarrow x = 0$$

$$\frac{\partial f}{\partial y} = -2y = 0 \Rightarrow y = 0$$

So, *(x,y)* = (0,0) is a stationary point. The next thing to do is to compute the Hessian matrix and evaluate its Eigen values at *(x,y)* = (0,0). The Hessian matrix *Hf(x, y*) is as follows:

$$Hf(x,y) = \begin{bmatrix} \frac{\partial^2 f}{\partial x^2} & \frac{\partial^2 f}{\partial x \partial y} \\ \frac{\partial^2 f}{\partial x \partial y} & \frac{\partial^2 f}{\partial y^2} \end{bmatrix} = \begin{bmatrix} 2 & 0 \\ 0 & -2 \end{bmatrix}$$

So, the Hessian *Hf(x,y)* at all points including *(x,y*) = (0,0) is $\begin{bmatrix} 2 & 0 \\ 0 & -2 \end{bmatrix}$.

The two eigen values of the *Hf(x,y)* are 2 and -2, corresponding to the Eigen vectors $\begin{bmatrix} 1 \\ 0 \end{bmatrix}$ and $\begin{bmatrix} 0 \\ 1 \end{bmatrix}$, which are nothing but the directions along the *X* and *Y* axes. Since one Eigen value is positive and the other negative, *(x,y)* = (0,0) is a saddle point.

The non-convex function $f(x,y) = x^2 - y^2$ is plotted in Figure 2-21, where *S* is the saddle point at *x, y* =(0,0)

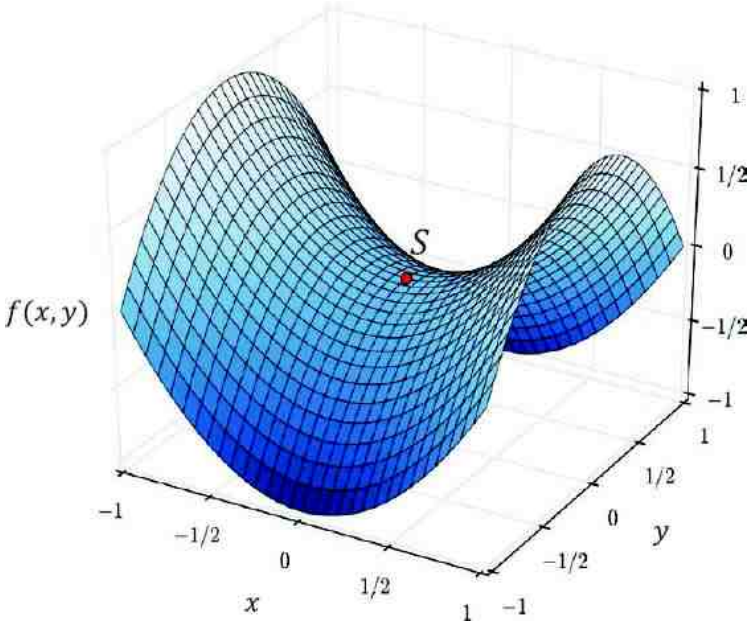

Figure 2-21: Plot of $f(x,y) = x^2 - y^2$

Learning Rate in Mini-batch Approach to Stochastic Gradient Descent

When there is high redundancy in the dataset, the gradient computed on a mini-batch of data points is almost the same as the gradient computed on the whole dataset, provided the mini batch is a good representation of the entire dataset. In such cases, computing the gradient on the whole dataset can be avoided, and instead the gradient on the mini batch of data points can be used as the approximate gradient for the whole dataset. This is the mini-batch approach to gradient descent, which is also called mini-batch stochastic gradient descent. When, instead of using a mini batch, the gradients are approximated by one data point, it is called online learning or stochastic gradient descent. However, it is always better to use the mini-batch version of stochastic gradient descent over online learning since the gradients for the mini-batch method are less noisy compared to the online mode of learning. Learning rate plays a vital role in the convergence of mini-batch stochastic gradient descent. The following approach tends to provide good convergence:

- Start with an initial learning rate.
- Increase the learning rate if the error reduces.

- Decrease the learning rate if the error increases.
- Stop the learning process if the error ceases to reduce.

As we will see in the next section, the different optimizers adopt an adaptive learning-rate approach in their implementations.

Optimizers in TensorFlow

TensorFlow has a rich inventory of optimizers for optimizing cost functions. The optimizers are all gradient based, along with some special optimizers to handle local minima problems. Since we dealt with the most common gradient-based optimizers used in machine learning and deep learning in the first chapter, here we will stress the customizations added in TensorFlow to the base algorithms.

GradientDescentOptimizer

`GradientDescentOptimizer` implements the fundamental full-batch gradient-descent algorithm and takes the learning rate as an input. The gradient-descent algorithm will not loop over the iterations automatically, so such logic must be specified in the implementation, as we will see later.

The most important method is the `minimize` method in which one needs to specify the cost function to minimize (denoted by `loss`) and the variable list (denoted by `var_list`) with respect to which the cost function must be minimized. The `minimize` method internally invokes the `compute_gradients()` and `apply_gradients()` methods. Declaring the variable list is optional, and if not specified the gradients are computed with respect to the variables defined as TensorFlow variables (i.e., ones declared as `tensorflow.Variable()`)

Usage

```
train_op = tf.train.GradientDescentOptimizer(learning_rate).minimize(cost)
```

where `learning_rate` is the constant learning rate and `cost` is the cost function that needs to be minimized through gradient descent. The cost function is minimized with respect to the TensorFlow variables associated with the cost function.

AdagradOptimizer

`AdagradOptimizer` is a first-order optimizer like gradient descent but with some modifications. Instead of having a global learning rate, the learning rate is normalized for each dimension on which the cost function is dependent. The learning rate in each iteration is the global learning rate divided by the l^2 norm of the prior gradients up to the current iteration for each dimension.

If we have a cost function $C(\theta)$ where $\theta = \left[\theta_1\theta_2\theta_3 \ldots \theta_n\right]^T \in \mathbb{R}^{n\times 1}$, then the update rule for θ_i is as follows:

$$\theta_i^{(t+1)} = \theta_i^{(t)} - \frac{\eta}{\sqrt{\sum_{\tau=1}^{t}\theta_i^{(\tau)2} + \epsilon}} \frac{\partial C^{(t)}}{\partial \theta_i}$$

Where η is the learning rate and $\theta_i^{(t)}$ and $\theta_i^{(t+1)}$ are the values for the *ith* parameter at iterations t and $t+1$ respectively.

In a matrix format, the parameter update for the vector θ can be represented by the following:

$$\theta^{(t+1)} = \theta^{(t)} - \eta G_{(t)}^{-1} \nabla C\left(\theta^{(t)}\right)$$

where $G_{(t)}$ is the diagonal matrix containing the l^2 norm of the past gradients till iteration t for each dimension. The matrix $G_{(t)}$ would be of the following form:

$$G_{(t)} = \begin{bmatrix} \sqrt{\sum_{\tau=1}^{t}\theta_1^{(\tau)2} + \epsilon} & \cdots & 0 \\ \vdots & \sqrt{\sum_{\tau=1}^{t}\theta_i^{(\tau)2} + \epsilon} & \vdots \\ 0 & \cdots & \sqrt{\sum_{\tau=1}^{t}\theta_n^{(\tau)2} + \epsilon} \end{bmatrix}$$

Sometimes sparse features that don't show up much in the data can be very useful to an optimization problem. However, with basic

gradient descent or stochastic gradient descent the learning rate gives equal importance to all the features in each iteration. Since the learning rate is same, the overall contribution of non-sparse features would be much more than that of sparse features. Hence, we end up losing critical information from the sparse features. With `Adagrad`, each parameter is updated with a different learning rate. The sparser the feature is, the higher its parameter update would be in an iteration. This is because for sparse features the quantity $\sqrt{\sum_{\tau=1}^{t}\theta_i^{(\tau)2}+\epsilon}$ would be less and hence the overall learning rate would be high.

This is a good optimizer to use in applications with natural language processing and image processing where the data is sparse.

Usage

```
train_op = tf.train.AdagradOptimizer.(learning_rate=0.001, initial_accumulator_value=0.1)
```

where `learning_rate` represents η and `initial_accumulator_value` represents the initial non-zero normalizing factor for each weight.

RMSprop

`RMSprop` is the mini-batch version of the resilient backpropagation `(Rprop)` optimization technique that works best for full-batch learning. `Rprop` solves the issue of gradients' not pointing to the minimum in cases where the cost function contours are elliptical. As we discussed earlier, in such cases, instead of a global learning rule a separate adaptive update rule for each weight would lead to better convergence. The special thing with `Rprop` is that it doesn't use the magnitude of the gradients of the weight but only the signs in determining how to update each weight. The following is the logic by which `Rprop` works:

- Start with the same magnitude of weight update for all weights; i.e., $\Delta_{ij}^{(t=0)}=\Delta_{ij}^{(0)}=\Delta$.

  Also, set the maximum and minimum allowable weight updates to Δ_{max} and Δ_{min} respectively.

- At each iteration, check the sign of both the previous and the current gradient components; i.e., the partial derivatives of the cost function with respect to the different weights.

- If the signs of the current and previous gradient components for a weight connection are the same–i.e., *sign* $\left(\frac{\partial C^{(t)}}{\partial w_{ij}}\frac{\partial C^{(t-1)}}{\partial w_{ij}}\right)=+ve$ —then increase the learning by a factor η_+ = 1.2. The update rule becomes

$$\Delta_{ij}^{(t+1)}=\min\left(\eta_+\Delta_{ij}^{(t)},\Delta_{\max}\right)$$

$$w_{ij}^{(t+1)}=w_{ij}^{(t)}-sign\left(\frac{\partial C^{(t)}}{\partial w_{ij}}\right).\Delta_{ij}^{(t+1)}$$

- If the signs of the current and previous gradient components for a dimension are different–i.e., *sign* $\left(\frac{\partial C^{(t)}}{\partial w_{ij}}\frac{\partial C^{(t-1)}}{\partial w_{ij}}\right)=-ve$ –then reduce the learning rate by a factor η_- = 0.5. The update rule becomes

$$\Delta_{ij}^{(t+1)}=\max\left(\eta_-\Delta_{ij}^{(t)},\Delta_{\min}\right)$$

$$w_{ij}^{(t+1)}=w_{ij}^{(t)}-sign\left(\frac{\partial C^{(t)}}{\partial w_{ij}}\right).\Delta_{ij}^{(t+1)}$$

- If $\frac{\partial C^{(t)}}{\partial w_{ij}}\frac{\partial C^{(t-1)}}{\partial w_{ij}}=0$, the update rule is as follows:

$$\Delta_{ij}^{(t+1)} = \Delta_{ij}^{(t)}$$

$$w_{ij}^{(t+1)} = w_{ij}^{(t)} - sign\left(\frac{\partial C^{(t)}}{\partial w_{ij}}\right).\Delta_{ij}^{(t+1)}$$

The dimensions along which the gradients are not changing sign at a specific interval during gradient descent are the dimensions along which the weight changes are consistent. Hence, increasing the learning rate would lead to faster convergence of those weights to their final value.

The dimensions along which the gradients are changing sign indicate that along those dimensions the weight changes are inconsistent, and so by decreasing the learning rate one would avoid oscillations and better catch up with the curvatures. For a convex function, gradient sign change generally occurs when there is curvature in the cost function surface and the learning rate is set high. Since the gradient doesn't have the curvature information, a large learning rate takes the updated parameter value beyond the minima point, and the phenomena keeps on repeating on either side of the minima point.

`Rprop` works well with full batches but doesn't do well when stochastic gradient descent is involved. When the learning rate is very small, gradients from different mini batches average out in cases of stochastic gradient descent. If through stochastic gradient descent for a cost function the gradients for a weight are +0.2 each for nine mini-batches and -0.18 for the tenth mini batch when the learning rate is small, then the effective gradient effect for stochastic gradient descent is almost zero and the weight remains almost at the same position, which is the desired outcome.

However, with `Rprop` the learning rate will increase about nine times and decrease only once, and hence the effective weight would be much larger than zero. This is undesirable.

To combine the qualities of `Rprop`'s adaptive learning rule for each weight with the efficiency of stochastic gradient descent, `RMSprop` came into the picture. In `Rprop` we don't use the magnitude but rather just the sign of the gradient for each weight. The sign of the gradient for each weight can be thought of as dividing the gradient for the weight by its magnitude. The problem with stochastic gradient descent is that with each mini batch the cost function keeps on changing and hence so do the gradients. So, the idea is to get a magnitude of gradient for a weight that would not fluctuate much over nearby mini batches. What would work well is a root mean of the squared gradients for each weight over the recent mini batches to normalize the gradient.

$$g_{ij}^{(t)} = \alpha g_{ij}^{(t-1)} + (1-\alpha)\left(\frac{\partial C^{(t)}}{\partial w_{ij}}\right)^2$$

$$w_{ij}^{(t+1)} = w_{ij}^{(t)} - \frac{\eta}{\sqrt{g_{ij}^{(t)} + \epsilon}} \frac{\partial C^{(t)}}{\partial w_{ij}}$$

where $g^{(t)}$ is the root mean square of the gradients for the weight w_{ij} at iteration t and α is the decay rate for the root mean square gradient for each weight w_{ij}.

Usage

```
train_op = tf.train.RMSPropOptimizer(learning_rate=0.001, decay =0.9,
momentum=0.0, epsilon=1e-10)
```

where `decay` represents α, epsilon represents ϵ, and η represents the learning rate.

AdadeltaOptimizer

`AdadeltaOptimizer` is a variant of `AdagradOptimizer` that is less aggressive in reducing the learning rate. For each weight connection, `AdagradOptimizer` scales the learning rate constant in an iteration by dividing it by the root mean square of all past gradients for that weight till that iteration. So, the effective learning rate for each weight is a monotonically decreasing function of the iteration number, and after a considerable number of iterations the learning rate becomes infinitesimally small. `AdagradOptimizer` overcomes this problem by taking the mean of the exponentially decaying squared gradients for each weight or dimension. Hence, the effective learning rate in `AdadeltaOptimizer` remains more of a local estimate of its current gradients and doesn't shrink as fast as the `AdagradOptimizer` method. This ensures that learning continues even after a considerable number of iterations or epochs. The learning rule for `Adadelta` can be summed up as follows:

$$g_{ij}^{(t)} = \gamma\, g_{ij}^{(t-1)} + (1-\gamma)\left(\frac{\partial C^{(t)}}{\partial w_{ij}}\right)^2$$

$$w_{ij}^{(t+1)} = w_{ij}^{(t)} - \frac{\eta}{\sqrt{g_{ij}^{(t)} + \epsilon}} \frac{\partial C^{(t)}}{\partial w_{ij}}$$

Where γ is the exponential decay constant, η is a learning-rate constant and $g_{ij}^{(t)}$ represents the effective mean square gradient at iteration *t*. We can denote the term $\sqrt{g_{ij}^{(t)} + \epsilon}$ as $RMS(g_{ij}^{(t)})$, which gives the update rule as follows:

$$w_{ij}^{(t+1)} = w_{ij}^{(t)} - \frac{\eta}{RMS\left(g_{ij}^{(t)}\right)} \frac{\partial C^{(t)}}{\partial w_{ij}}$$

If we observe carefully, the unit for the change in weight doesn't have the unit of the weight. The units for $\frac{\partial C^{(t)}}{\partial w_{ij}}$ and $RMS(g_{ij}^{(t)})$ are the same—i.e., the unit of gradient (cost function change/per unit weight change)—and hence they cancel each other out. Therefore, the unit of the weight change is the unit of the learning-rate constant. `Adadelta` solves this problem by replacing the learning-rate constant η with a square root of the mean of the exponentially decaying squared-weight updates up to the current iteration. Let $h_{ij}^{(t)}$ be the mean of the square of the weight updates up to iteration *t*, β be the decaying constant, and $\Delta w_{ij}^{(t)}$ be the weight update in iteration *t*. Then, the update rule for $h_{ij}^{(t)}$ and the final weight update rule for `Adadelta` can be expressed as follows:

$$h_{ij}^{(t)} = \beta h_{ij}^{(t-1)} + (1-\beta)(\Delta w_{ij}^{(t)})^2$$

$$w_{ij}^{(t+1)} = w_{ij}^{(t)} - \frac{\sqrt{h_{ij}^{(t)} + \epsilon}}{RMS(g_{ij}^{(t)})} \frac{\partial C^{(t)}}{\partial w_{ij}}$$

If we denote $\sqrt{h_{ij}^{(t)} + \epsilon}$ as $RMS(h_{ij}^{(t)})$ then the update rule becomes -

$$w_{ij}^{(t+1)} = w_{ij}^{(t)} - \frac{RMS(h_{ij}^{(t)})}{RMS(g_{ij}^{(t)})} \frac{\partial C^{(t)}}{\partial w_{ij}}$$

Usage

```
train_op = tf.train.AdadeltaOptimizer(learning_rate=0.001, rho=0.95, epsilon=1e-08)
```

where `rho` represents γ, `epsilon` represents ϵ, and η represents the learning rate.

One significant advantage of `Adadelta` is that it eliminates the learning-rate constant altogether. If we compare `Adadelta` and `RMSprop`, both are the same if we leave aside the learning-rate constant elimination. `Adadelta` and `RMSprop` were both developed independently around the same time to resolve the fast learning-rate decay problem of `Adagrad`.

AdamOptimizer

`Adam`, or Adaptive Moment Estimator, is another optimization technique that, much like `RMSprop` or `Adagrad`, has an adaptive learning rate for each parameter or weight. `Adam` not only keeps a running mean of squared gradients but also keeps a running mean of past gradients.

Let the decay rate of the mean of gradients $m_{ij}^{(t)}$ and the mean of the square of gradients $v_{ij}^{(t)}$ for each weight w_{ij} be β_1 and β_2 respectively. Also, let η be the constant learning-rate factor. Then, the update rules for `Adam` are as follows:

$$m_{ij}^{(t)} = \beta_1 m_{ij}^{(t-1)} + (1-\beta_1)\frac{\partial C^{(t)}}{\partial w_{ij}}$$

$$v_{ij}^{(t)} = \beta_2 v_{ij}^{(t-1)} + (1-\beta_2)\left(\frac{\partial C^{(t)}}{\partial w_{ij}}\right)^2$$

The normalized mean of the gradients $\hat{m}_{ij}^{(t)}$ and the mean of the square gradients $\hat{v}_{ij}^{(t)}$ are computed as follows:

$$\hat{m}_{ij}^{(t)} = \frac{m_{ij}^{(t)}}{\left(1-\beta_1^{\,t}\right)}$$

$$\hat{v}_{ij}^{(t)} = \frac{v_{ij}^{(t)}}{\left(1-\beta_2^{\,t}\right)}$$

The final update rule for each weight w_{ij} is as follows:

$$w_{ij}^{(t+1)} = w_{ij}^{(t)} - \frac{\eta}{\sqrt{\hat{v}_{ij}^{(t)} + \epsilon}}\hat{m}_{ij}^{(t)}$$

Usage

```
train_op = tf.train.AdamOptimizer(learning_rate=0.001,beta1=0.9,beta2=0.999,epsilon=1e-08).
minimize(cost)
```

where `learning_rate` is the constant learning rate η and `cost` C is the cost function that needs to be minimized through `AdamOptimizer`. The parameters `beta1` and `beta2` correspond to β_1 and β_2 respectively, whereas `epsilon` represents ϵ.

The cost function is minimized with respect to the TensorFlow variables associated with the cost function.

MomentumOptimizer and Nesterov Algorithm

The momentum-based optimizers have evolved to take care of non-convex optimizations. Whenever we are working with neural networks, the cost functions that we generally get are non-convex in nature, and thus the gradient-based optimization methods might get caught up in bad local minima. As discussed earlier, this is highly undesirable since in such cases we get a sub-optimal solution to the optimization problem—and likely a sub-optimal model. Also, gradient descent follows the slope at each point and makes small advances toward the local minima, but it can be terribly slow. Momentum-based methods introduce a component called velocity v that dampens the parameter update when the gradient computed changes sign, whereas it accelerates the parameter update when the gradient is in the same direction of velocity. This introduces faster convergence as well as fewer oscillations around the global minima, or around a local minimum that provides good generalization. The update rule for momentum-based optimizers is as follows:

$$v_i^{(t+1)} = \alpha v_i^{(t)} - \eta\frac{\partial C}{\partial w_i}\left(w_i^{(t)}\right)$$

$$w_i^{(t+1)} = w_i^{(t)} + v_i^{(t+1)}$$

Where α is the momentum parameter and η is the learning rate. The terms $v_i^{(t)}$ and $v_i^{(t+1)}$ represent the velocity at iterations t and (t +1) respectively for the *ith* parameter. Similarly, $w_i^{(t)}$ and $w_i^{(t+1)}$ represent the weight of the *ith* parameter at iterations t and (t + 1) respectively.

Imagine that while optimizing a cost function the optimization algorithm reaches a local minimum where $\frac{\partial C}{\partial w_i}\left(w_i^{(t)}\right) \to 0 \; \forall i \in \mathbb{R}^{m\times 1}$. In normal gradient-descent methods that don't take momentum into consideration, the parameter update would stop at that local minimum or the saddle point. However, in momentum-based optimization, the prior velocity would drive the algorithm out of the local minima, considering the local minima has a small basin of attraction, as $v_i^{(t+1)}$ would be non-zero because of the non-zero velocity from prior gradients. Also, if the prior gradients consistently pointed toward a global minimum or a local minimum with good generalization and a

reasonably large basin of attraction, the velocity or the momentum of gradient descent would be in that direction. So, even if there were a bad local minimum with a small basin of attraction, the momentum component would not only drive the algorithm out of the bad local minima but also would continue the gradient descent toward the global minima or the good local minima.

If the weights are part of the parameter vector θ, the vectorized update rule for momentum-based optimizers would be as follows (refer to Figure 2-22 for the vector-based illustration):

$$v^{(t+1)} = \alpha v^{(t)} - \eta \nabla C\left(\theta = \theta^{(t)}\right)$$

$$\theta^{(t+1)} = \theta^{(t)} + v^{(t+1)}$$

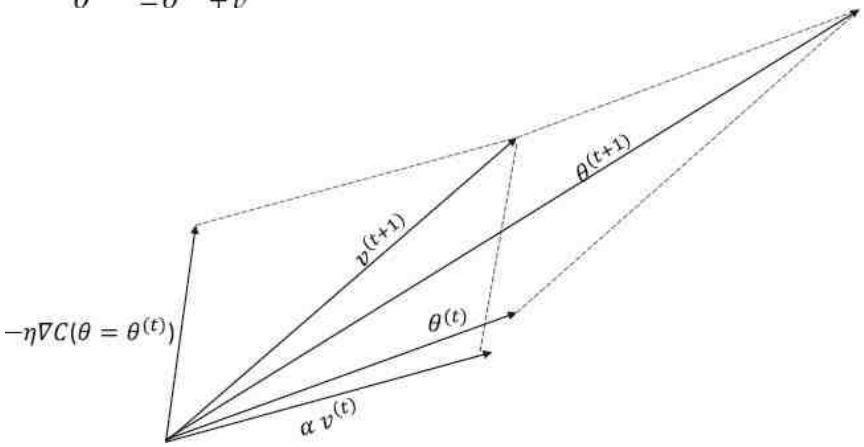

Figure 2-22: Parameter vector update in momentum-based gradlen'-descen' optimizer

A specific variant of momentum-based optimizers is the Nesterov accelerated gradient technique. This method utilizes the existing velocity $v^{(t)}$ to make an update to the parameter vector. Since it's an intermediate update to the parameter vector, it's convenient to denote it by $\theta^{\left(t+\frac{1}{2}\right)}$. The gradient of the cost function is evaluated at $\theta^{\left(t+\frac{1}{2}\right)}$, and the same is used to update the new velocity. Finally, the new parameter vector is the sum of the parameter vector at the previous iteration and the new velocity.

$$\theta^{\left(t+\frac{1}{2}\right)} = \theta^{(t)} + \alpha v^{(t)}$$

$$v^{(t+1)} = \alpha v^{(t)} - \eta \nabla C\left(\theta = \theta^{\left(t+\frac{1}{2}\right)}\right)$$

$$\theta^{(t+1)} = \theta^{(t)} + v^{(t+1)}$$

Usage

```
train_op = tf.train.MomentumOptimizer.(learning_rate=0.001, momentum=0.9,use_nesterov=False)
```

where `learning_rate` represents η, `momentum` represents α, and `use_nesterov` determines whether to use the Nesterov version of momentum.

Epoch, Number of Batches, and Batch Size

Deep-learning networks, as mentioned earlier, are generally trained through mini-batch stochastic gradient descent. A few of the terms with which we need to familiarize ourselves are as follows:

- ***Batch size*** – The batch size determines the number of training data points in each mini batch. The batch size should be chosen such that it gives a good enough estimate of the gradient for the full training dataset and at the same time noisy enough to escape bad local minima that don't provide good generalization.
- ***Number of batches*** – The number of batches gives the total number of mini batches in the entire training dataset. It can be computed by dividing the count of the total training data points by the batch size. Please note that the last mini batch might have a smaller number of data points than the batch size.

- ***Epochs*** – One epoch consists of one full pass of training over the entire dataset. To be more specific, one epoch is equivalent to a forward pass plus one backpropagation over the entire training dataset. So, one epoch would consist of *n* number of (forward pass + backpropagation) where *n* denotes the number of batches.

XOR Implementation Using TensorFlow

Now that we have a decent idea of the components and training methods involved with an artificial neural network, we will implement a XOR network using sigmoid activation functions in the hidden layers as well as in the output. The detailed implementation has been outlined in Listing 2-16.

Listing 2-16: XOR Implementation with Hidden Layers That Have Sigmoid Activation Functions

```
#-------------------------------------------------------------------------------------------
#XOR  implementation in Tensorflow with hidden layers being sigmoid to
# introduce Non-Linearity
#-------------------------------------------------------------------------------------------
import tensorflow as tf
#-------------------------------------------------------------------------------------------
# Create placeholders for training input and output labels
#-------------------------------------------------------------------------------------------
x_ = tf.placeholder(tf.float32, shape=[4,2], name="x-input")
y_ = tf.placeholder(tf.float32, shape=[4,1], name="y-input")
#-------------------------------------------------------------------------------------------
#Define the weights to the hidden and output layer respectively.
#-------------------------------------------------------------------------------------------
w1 = tf.Variable(tf.random_uniform([2,2], -1, 1), name="Weights1")
w2 = tf.Variable(tf.random_uniform([2,1], -1, 1), name="Weights2")
#-------------------------------------------------------------------------------------------
# Define the bias to the hidden and output layers respectively
#-------------------------------------------------------------------------------------------
b1 = tf.Variable(tf.zeros([2]), name="Bias1")
b2 = tf.Variable(tf.zeros([1]), name="Bias2")
#-------------------------------------------------------------------------------------------
# Define the final output through forward pass
#-------------------------------------------------------------------------------------------
z2 = tf.sigmoid(tf.matmul(x_, w1) + b1)
pred = tf.sigmoid(tf.matmul(z2,w2) + b2)
#-------------------------------------------------------------------------------------------
#Define the Cross-entropy/Log-loss Cost function based on the output label y and
# the predicted probability by the forward pass
#-------------------------------------------------------------------------------------------
cost = tf.reduce_mean(( (y_ * tf.log(pred)) +
        ((1 - y_) * tf.log(1.0 - pred)) ) * -1)
learning_rate = 0.01
train_step = tf.train.GradientDescentOptimizer(learning_rate).minimize(cost)
#-------------------------------------------------------------------------------------------
#Now that we have all that we need set up we will start the training
#-------------------------------------------------------------------------------------------
XOR_X = [[0,0],[0,1],[1,0],[1,1]]
XOR_Y = [[0],[1],[1],[0]]
#-------------------------------------------------------------------------------------------
# Initialize the variables
#-------------------------------------------------------------------------------------------
init = tf.initialize_all_variables()
sess = tf.Session()
writer = tf.summary.FileWriter("./Downloads/XOR_logs", sess.graph_def)

sess.run(init)
for i in range(100000):
        sess.run(train_step, feed_dict={x_: XOR_X, y_: XOR_Y})

#-------------------------------------------------------------------------------------------
print('Final Prediction', sess.run(pred, feed_dict={x_: XOR_X, y_: XOR_Y}))
#-------------------------------------------------------------------------------------------

--output --

('Final Prediction', array([[ 0.06764214],
       [ 0.93982035],
       [ 0.95572311],
       [ 0.05693595]], dtype=float32))
```

In Listing 2-16, the XOR logic is implemented using TensorFlow. The hidden layer units have sigmoid activation functions to introduce non-linearity. The output activation function has a sigmoid activation function to give probability outputs. We are using the gradient-descent optimizer with a learning rate of 0.01 and total iterations of around 100,000. If we see the final prediction, the first and fourth training samples have a near-zero value for probabilities while the second and fourth training samples have probabilities near 1. So, the network can predict the classes accurately and with high precision. Any reasonable threshold would classify the data points correctly.

TensorFlow Computation Graph for XOR network

In Figure 2-23, the computation graph for the preceding implemented XOR network is illustrated. The computation graph summary is written to the log files by including the following line of code. The phrase `". /Downloads/XOR_logs"` indicates the location where the summary log files have been stored. It can be any location you choose, however.

```
writer = tf.summary.FileWriter("./Downloads/XOR_logs", sess.graph_def)
```

Once the summary has been written to the log files on the terminal, we need to execute the following command to activate Tensorboard:

```
tensorboard --logdir=./Downloads/XOR_logs
```

This would start the Tensorboard session and prompt us to access the Tensorboard at http://localhost:6006, where the computation graph could be visualized.

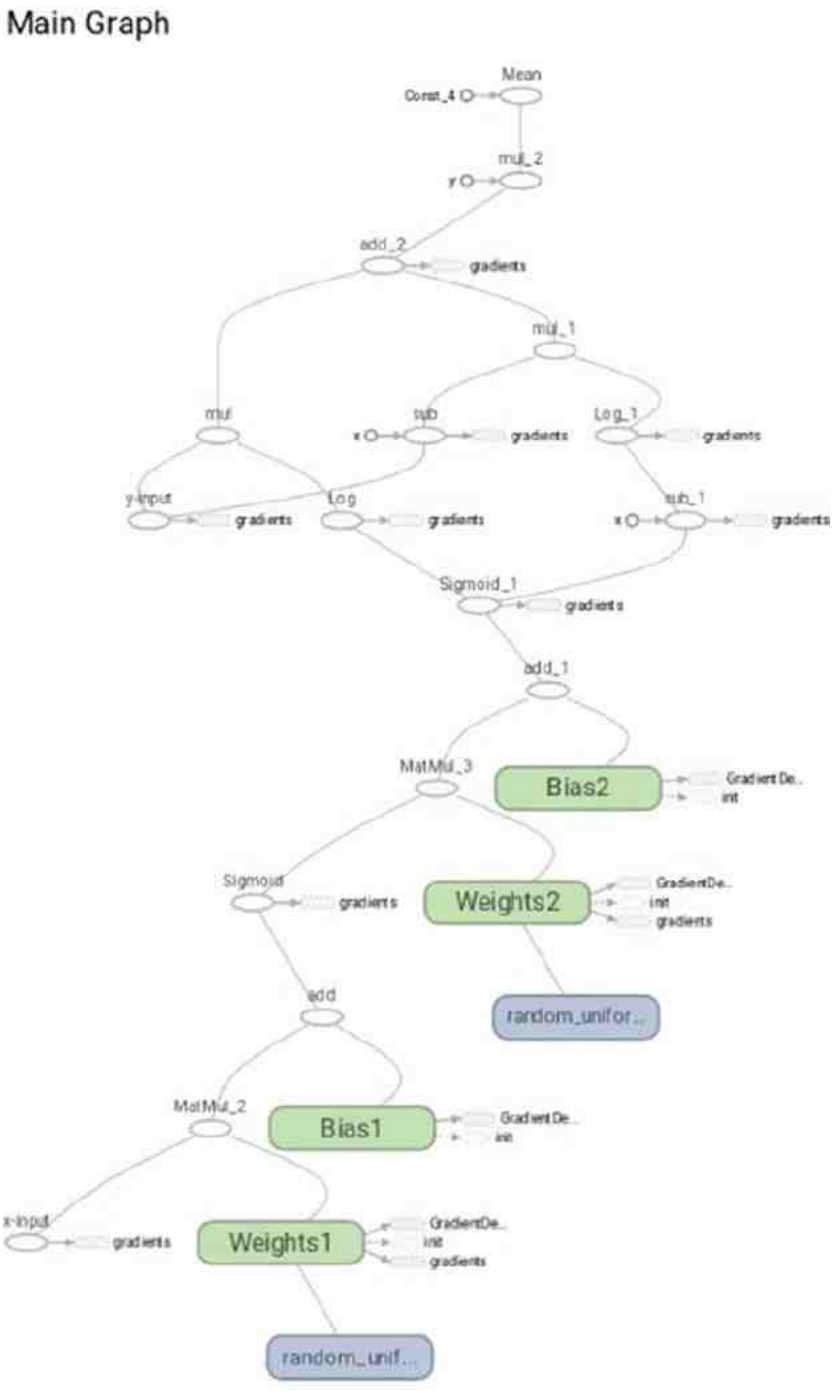

Figure 2-23: Computation graph for the XOR network

Now, we implement the XOR logic again, using linear activation functions in the hidden layer and keeping the rest of the network as it is. Listing 2-17 shows the TensorFlow implementation.

Listing 2-17: XOR Implementation with Linear Activation Functions in Hidden Layer

```
#-------------------------------------------------------------------------------------------
#XOR  implementation in TensorFlow with linear activation for hidden layers
#-------------------------------------------------------------------------------------------
import tensorflow as tf
#-------------------------------------------------------------------------------------------
# Create placeholders for training input and output labels
#-------------------------------------------------------------------------------------------
x_ = tf.placeholder(tf.float32, shape=[4,2], name="x-input")
y_ = tf.placeholder(tf.float32, shape=[4,1], name="y-input")
#-------------------------------------------------------------------------------------------
#Define the weights to the hidden and output layer respectively.
#-------------------------------------------------------------------------------------------
w1 = tf.Variable(tf.random_uniform([2,2], -1, 1), name="Weights1")
w2 = tf.Variable(tf.random_uniform([2,1], -1, 1), name="Weights2")
#-------------------------------------------------------------------------------------------
# Define the bias to the hidden and output layers respectively
#-------------------------------------------------------------------------------------------
b1 = tf.Variable(tf.zeros([2]), name="Bias1")
b2 = tf.Variable(tf.zeros([1]), name="Bias2")
#-------------------------------------------------------------------------------------------
# Define the final output through forward pass
#-------------------------------------------------------------------------------------------
z2 = tf.matmul(x_, w1) + b1
pred = tf.sigmoid(tf.matmul(z2,w2) + b2)
#-------------------------------------------------------------------------------------------
#Define the Cross-entropy/Log-loss Cost function based on the output label y and the
predicted
#probability by the forward pass
#-------------------------------------------------------------------------------------------
cost = tf.reduce_mean(( (y_ * tf.log(pred)) +
        ((1 - y_) * tf.log(1.0 - pred)) ) * -1)
learning_rate = 0.01
train_step = tf.train.GradientDescentOptimizer(learning_rate).minimize(cost)
#-------------------------------------------------------------------------------------------
#Now that we have all that we need, start the training
#-------------------------------------------------------------------------------------------
XOR_X = [[0,0],[0,1],[1,0],[1,1]]
XOR_Y = [[0],[1],[1],[0]]

init = tf.initialize_all_variables()
sess = tf.Session()
sess.run(init)
for i in range(100000):
        sess.run(train_step, feed_dict={x_: XOR_X, y_: XOR_Y})

#-------------------------------------------------------------------------------------------
print('Final Prediction', sess.run(pred, feed_dict={x_: XOR_X, y_: XOR_Y}))
#-------------------------------------------------------------------------------------------

-- output --

('Final Prediction', array([[ 0.5000003 ],
       [ 0.50001115],
       [ 0.49998885],
       [ 0.4999997 ]], dtype=float32))
```

The final predictions as shown in Listing 2-2 are all near 0.5, which means that the implemented XOR logic is not able to do a good job in discriminating the positive class from the negative one. When we have linear activation functions in the hidden layer, the network primarily remains linear, as we have seen previously, and hence the model is not able to do well where non-linear decision boundaries are required to separate classes.

Linear Regression in TensorFlow

Linear regression can be expressed as a single-neuron regression problem. The mean of the square of the errors in prediction is taken as the cost function to be optimized with respect to the co-efficient of the model. Listing 2-18 shows a TensorFlow implementation of linear regression with the Boston housing price dataset.

Listing 2-18: Linear Regression Implementation in TensorFlow

```
#-------------------------------------------------------------------------------------------
# Importing TensorFlow, Numpy, and the Boston Housing price dataset
#-------------------------------------------------------------------------------------------

import tensorflow as tf
import numpy as np
from sklearn.datasets import load_boston
```

```
#-------------------------------------------------------------------------------------------
# Function to load the Boston data set
#-------------------------------------------------------------------------------------------

def read_infile():
    data = load_boston()
    features = np.array(data.data)
    target = np.array(data.target)
    return features,target

#-------------------------------------------------------------------------------------------
# Normalize the features by Z scaling; i.e., subtract from each feature value its mean and
then divide by its #standard deviation. Accelerates gradient descent.
#-------------------------------------------------------------------------------------------

def feature_normalize(data):
    mu = np.mean(data,axis=0)
    std = np.std(data,axis=0)
    return (data - mu)/std

#-------------------------------------------------------------------------------------------
# Append the feature for the bias term.
#-------------------------------------------------------------------------------------------

def append_bias(features,target):
    n_samples = features.shape[0]
    n_features = features.shape[1]
    intercept_feature  = np.ones((n_samples,1))
    X = np.concatenate((features,intercept_feature),axis=1)
    X = np.reshape(X,[n_samples,n_features +1])
    Y = np.reshape(target,[n_samples,1])
    return X,Y
#-------------------------------------------------------------------------------------------
#  Execute the functions to read, normalize, and add append bias term to the data
#-------------------------------------------------------------------------------------------

features,target = read_infile()
z_features = feature_normalize(features)
X_input,Y_input = append_bias(z_features,target)
num_features = X_input.shape[1]

#-------------------------------------------------------------------------------------------
# Create TensorFlow ops for placeholders, weights, and weight initialization
#-------------------------------------------------------------------------------------------

X = tf.placeholder(tf.float32,[None,num_features])
Y = tf.placeholder(tf.float32,[None,1])
w = tf.Variable(tf.random_normal((num_features,1)),name='weights')
init = tf.global_variables_initializer()

#-------------------------------------------------------------------------------------------
# Define the different TensorFlow ops and input parameters for Cost and Optimization.
#-------------------------------------------------------------------------------------------

learning_rate = 0.01
num_epochs = 1000
cost_trace = []
pred = tf.matmul(X,w)
error = pred - Y
cost = tf.reduce_mean(tf.square(error))
train_op = tf.train.GradientDescentOptimizer(learning_rate).minimize(cost)

#-------------------------------------------------------------------------------------------
# Execute the gradient-descent learning
#-------------------------------------------------------------------------------------------

with tf.Session() as sess:
    sess.run(init)
    for i in xrange(num_epochs):
        sess.run(train_op,feed_dict={X:X_input,Y:Y_input})
        cost_trace.append(sess.run(cost,feed_dict={X:X_input,Y:Y_input}))
    error_ = sess.run(error,{X:X_input,Y:Y_input})
    pred_ = sess.run(pred,{X:X_input})

print 'MSE in training:',cost_trace[-1]

-- output --

MSE in training: 21.9711
```

Listing 2-18a: Linear Regression Cost Plot over Epochs or Iterations

```
#-------------------------------------------------------------------------------------------
# Plot the reduction in cost over iterations or epochs
```

```
#-------------------------------------------------------------------------------------------

import matplotlib.pyplot as plt
%matplotlib inline
plt.plot(cost_trace)
```

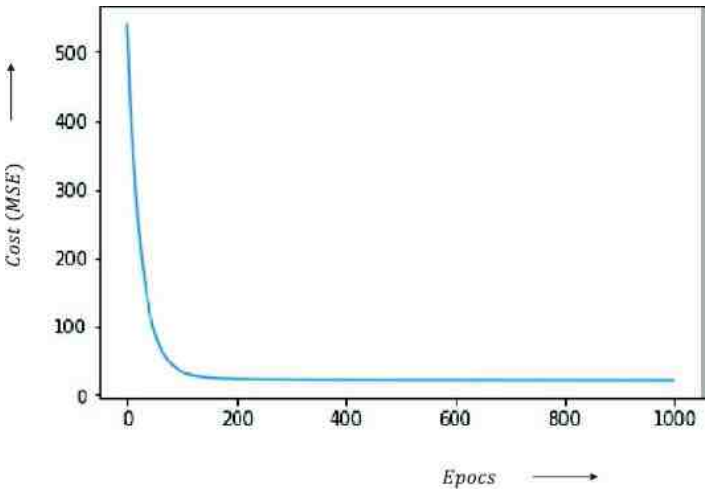

Figure 2-24: Cost (MSE) versus epochs while training

Listing 2-18b: Linear Regression Actual House Price Versus Predicted House Price

```
#-------------------------------------------------------------------------------------------
# Plot the Predicted House Prices vs the Actual House Prices
#-------------------------------------------------------------------------------------------

fig, ax = plt.subplots()
plt.scatter(Y_input,pred_)
ax.set_xlabel('Actual House price')
ax.set_ylabel('Predicted House price')
```

Figure 2-25: Actual house price versus predicted house price

Figure 2-24 illustrates the cost progression against the epochs and Figure 2-25 illustrates the predicted house price versus the actual house price after training.

Multi-class Classification with SoftMax Function Using Full-Batch Gradient Descent

In this section, we illustrate a multi-class classification problem using full-batch gradient descent. The MNIST dataset has been used because there are 10 output classes corresponding to the 10 integers. The detailed implementation is provided in Listing 2-19. A SoftMax has been used as the output layer.

Listing 2-19: Multi-class Classification with Softmax Function Using Full-Batch Gradient Descent

```
#-------------------------------------------------------------------------------------------
# Import the required libraries
#-------------------------------------------------------------------------------------------
```

```
import tensorflow as tf
import numpy as np
from sklearn import datasets
from tensorflow.examples.tutorials.mnist import input_data

#---------------------------------------------------------------------------------------
# Function to read the MNIST dataset along with the labels
#---------------------------------------------------------------------------------------

def read_infile():
    mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
    train_X, train_Y,test_X, test_Y = mnist.train.images, mnist.train.labels, mnist.test.
    images, mnist.test.labels
    return train_X, train_Y,test_X, test_Y

#---------------------------------------------------------------------------------------
#  Define the weights and biases for the neural network
#---------------------------------------------------------------------------------------

def weights_biases_placeholder(n_dim,n_classes):
    X = tf.placeholder(tf.float32,[None,n_dim])
    Y = tf.placeholder(tf.float32,[None,n_classes])
    w = tf.Variable(tf.random_normal([n_dim,n_classes],stddev=0.01),name='weights')
    b = tf.Variable(tf.random_normal([n_classes]),name='weights')
    return X,Y,w,b

#---------------------------------------------------------------------------------------
# Define the forward pass
#---------------------------------------------------------------------------------------

def forward_pass(w,b,X):
    out = tf.matmul(X,w) + b
    return out

#---------------------------------------------------------------------------------------
# Define the cost function for the SoftMax unit
#---------------------------------------------------------------------------------------

def multiclass_cost(out,Y):
    cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=out,labels=Y))
    return cost

#---------------------------------------------------------------------------------------
# Define the initialization op
#---------------------------------------------------------------------------------------

def init():
    return tf.global_variables_initializer()

#---------------------------------------------------------------------------------------
# Define the training op
#---------------------------------------------------------------------------------------

def train_op(learning_rate,cost):
    op_train = tf.train.GradientDescentOptimizer(learning_rate).minimize(cost)
    return op_train

train_X, train_Y,test_X, test_Y = read_infile()
X,Y,w,b = weights_biases_placeholder(train_X.shape[1],train_Y.shape[1])
out = forward_pass(w,b,X)
cost = multiclass_cost(out,Y)
learning_rate,epochs = 0.01,1000
op_train = train_op(learning_rate,cost)
init = init()
loss_trace = []
accuracy_trace = []

#---------------------------------------------------------------------------------------
# Activate the TensorFlow session and execute the stochastic gradient descent
#---------------------------------------------------------------------------------------

with tf.Session() as sess:
    sess.run(init)

    for i in xrange(epochs):
        sess.run(op_train,feed_dict={X:train_X,Y:train_Y})
        loss_ = sess.run(cost,feed_dict={X:train_X,Y:train_Y})
        accuracy_ = np.mean(np.argmax(sess.run(out,feed_dict={X:train_X,Y:train_Y}),axis=1)
== np.argmax(train_Y,axis=1))
        loss_trace.append(loss_)
        accuracy_trace.append(accuracy_)
        if (((i+1) >= 100) and ((i+1) % 100 == 0 )) :
            print 'Epoch:',(i+1),'loss:',loss_,'accuracy:',accuracy_

    print 'Final training result:','loss:',loss_,'accuracy:',accuracy_
```

```
    loss_test = sess.run(cost,feed_dict={X:test_X,Y:test_Y})
    test_pred = np.argmax(sess.run(out,feed_dict={X:test_X,Y:test_Y}),axis=1)
    accuracy_test = np.mean(test_pred == np.argmax(test_Y,axis=1))
    print 'Results on test dataset:','loss:',loss_test,'accuracy:',accuracy_test

-- output --
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
Epoch: 100 loss: 1.56331 accuracy: 0.702781818182
Epoch: 200 loss: 1.20598 accuracy: 0.772127272727
Epoch: 300 loss: 1.0129 accuracy: 0.800363636364
Epoch: 400 loss: 0.893824 accuracy: 0.815618181818
Epoch: 500 loss: 0.81304 accuracy: 0.826618181818
Epoch: 600 loss: 0.754416 accuracy: 0.834309090909
Epoch: 700 loss: 0.709744 accuracy: 0.840236363636
Epoch: 800 loss: 0.674433 accuracy: 0.845
Epoch: 900 loss: 0.645718 accuracy: 0.848945454545
Epoch: 1000 loss: 0.621835 accuracy: 0.852527272727
Final training result: loss: 0.621835 accuracy: 0.852527272727
Results on test dataset: loss: 0.596687 accuracy: 0.8614
```

Listing 2-19a: Display the Actual Digits Versus the Predicted Digits Along with the Images of the Actual Digits

```
import matplotlib.pyplot as plt
%matplotlib inline
f, a = plt.subplots(1, 10, figsize=(10, 2))
print 'Actual digits:   ', np.argmax(test_Y[0:10],axis=1)
print 'Predicted digits:',test_pred[0:10]
print 'Actual images of the digits follow:'
for i in range(10):
        a[i].imshow(np.reshape(test_X[i],(28, 28)))

-- output --
```

```
Actual digits:    [7 2 1 0 4 1 4 9 5 9]
Predicted digits: [7 2 1 0 4 1 4 9 6 9]
Actual images of the digits follow:
```

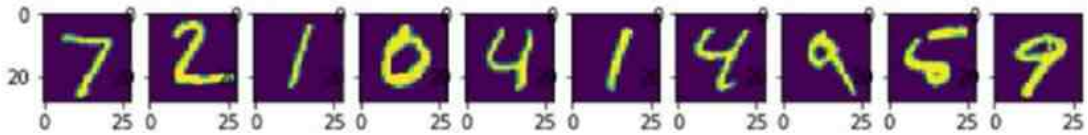

Figure 2-26: Actual digits versus predicted digits for SoftMax classification through gradient descent

Figure 2-26 displays the actual digits versus the predicted digits for SoftMax classification of the validation dataset samples after training through gradient-descent full-batch learning.

Multi-class Classification with SoftMax Function Using Stochastic Gradient Descent

We now perform the same classification task, but instead of using full-batch learning we resort to stochastic gradient descent with a batch size of 1000. The detailed implementation has been outlined in Listing 2-20.

Listing 2-20: Multi-class Classification with Softmax Function Using Stochastic Gradient Descent

```
def read_infile():
    mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
    train_X, train_Y,test_X, test_Y = mnist.train.images, mnist.train.labels, mnist.test.
images, mnist.test.labels
    return train_X, train_Y,test_X, test_Y

def weights_biases_placeholder(n_dim,n_classes):
    X = tf.placeholder(tf.float32,[None,n_dim])
    Y = tf.placeholder(tf.float32,[None,n_classes])
    w = tf.Variable(tf.random_normal([n_dim,n_classes],stddev=0.01),name='weights')
    b = tf.Variable(tf.random_normal([n_classes]),name='weights')
    return X,Y,w,b

def forward_pass(w,b,X):
    out = tf.matmul(X,w) + b
    return out

def multiclass_cost(out,Y):
    cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=out,labels=Y))
    return cost
```

```
def init():
    return tf.global_variables_initializer()

def train_op(learning_rate,cost):
    op_train = tf.train.AdamOptimizer(learning_rate).minimize(cost)
    return op_train

train_X, train_Y,test_X, test_Y = read_infile()
X,Y,w,b = weights_biases_placeholder(train_X.shape[1],train_Y.shape[1])
out = forward_pass(w,b,X)
cost = multiclass_cost(out,Y)
learning_rate,epochs,batch_size = 0.01,1000,1000
num_batches = train_X.shape[0]/batch_size
op_train = train_op(learning_rate,cost)
init = init()
epoch_cost_trace = []
epoch_accuracy_trace = []

with tf.Session() as sess:
    sess.run(init)

    for i in xrange(epochs):
        epoch_cost,epoch_accuracy = 0,0

        for j in xrange(num_batches):
            sess.run(op_train,feed_dict={X:train_X[j*batch_size:(j+1)*batch_
            size],Y:train_Y[j*batch_size:(j+1)*batch_size]})
            actual_batch_size = train_X[j*batch_size:(j+1)*batch_size].shape[0]
            epoch_cost += actual_batch_size*sess.run(cost,feed_dict={X:train_X[j*batch_
            size:(j+1)*batch_size],Y:train_Y[j*batch_size:(j+1)*batch_size]})
        epoch_cost = epoch_cost/float(train_X.shape[0])
        epoch_accuracy = np.mean(np.argmax(sess.run(out,feed_dict={X:train_X,Y:train_Y}),
        axis=1) == np.argmax(train_Y,axis=1))
        epoch_cost_trace.append(epoch_cost)
        epoch_accuracy_trace.append(epoch_accuracy)

        if (((i +1) >= 100) and ((i+1) % 100 == 0 )) :
            print 'Epoch:',(i+1),'Average loss:',epoch_cost,'accuracy:',epoch_accuracy

    print 'Final epoch training results:','Average loss:',epoch_cost,'accuracy:',epoch_
    accuracy
    loss_test = sess.run(cost,feed_dict={X:test_X,Y:test_Y})
    test_pred = np.argmax(sess.run(out,feed_dict={X:test_X,Y:test_Y}),axis=1)
    accuracy_test = np.mean(test_pred == np.argmax(test_Y,axis=1))
    print 'Results on test dataset:','Average loss:',loss_test,'accuracy:',accuracy_test

-- output --
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
Epoch: 100 Average loss: 0.217337096686 accuracy: 0.9388
Epoch: 200 Average loss: 0.212256691131 accuracy: 0.939672727273
Epoch: 300 Average loss: 0.210445133664 accuracy: 0.940054545455
Epoch: 400 Average loss: 0.209570150484 accuracy: 0.940181818182
Epoch: 500 Average loss: 0.209083143689 accuracy: 0.940527272727
Epoch: 600 Average loss: 0.208780818907 accuracy: 0.9406
Epoch: 700 Average loss: 0.208577176387 accuracy: 0.940636363636
Epoch: 800 Average loss: 0.208430663293 accuracy: 0.940636363636
Epoch: 900 Average loss: 0.208319870586 accuracy: 0.940781818182
Epoch: 1000 Average loss: 0.208232710849 accuracy: 0.940872727273
Final epoch training results: Average loss: 0.208232710849 accuracy: 0.940872727273
Results on test dataset: Average loss: 0.459194 accuracy: 0.9155
```

Listing 2-20a: Actual Digits Versus Predicted Digits for SoftMax Classification Through Stochastic Gradient Descent

```
import matplotlib.pyplot as plt
%matplotlib inline
f, a = plt.subplots(1, 10, figsize=(10, 2))
print 'Actual digits:   ', np.argmax(test_Y[0:10],axis=1)
print 'Predicted digits:',test_pred[0:10]
print 'Actual images of the digits follow:'
for i in range(10):
        a[i].imshow(np.reshape(test_X[i],(28, 28)))

--output --
```

Figure 2-27 displays the actual digits versus predicted digits for SoftMax classification of the validation dataset samples after training through stochastic gradient descent.

```
Actual digits:    [7 2 1 0 4 1 4 9 5 9]
Predicted digits: [7 2 1 0 4 1 4 9 6 9]
Actual images of the digits follow:
```

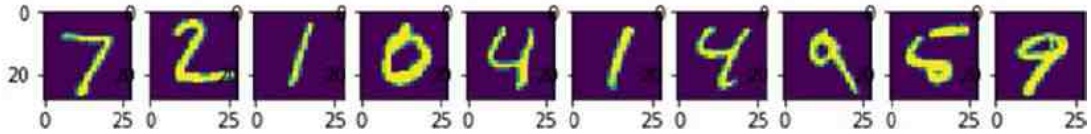

Figure 2-27: Actual digits versus predicted digits for SoftMax classification through stochastic gradient descent

GPU

Before we end this chapter, we want to talk a little about GPU, which has revolutionized the deep-learning world. GPU stands for graphical processing unit, which was initially used for gaming purposes to display more screens per second for better gaming resolution. Deep-learning networks use a lot of matrix multiplication, especially convolution, for both the forward pass and for backpropagation. GPUs are good at matrix-to-matrix multiplication; hence, several thousand cores of GPU are utilized to process data in parallel. This speeds up the deep-learning training. Common GPUs available in the market are

- NVIDIA GTX TITAN XGE
- NVIDIA GTX TITAN X
- NVIDIA GeForce GTX 1080
- NVIDIA GeForce GTX 1070

Summary

In this chapter, we have covered how deep learning has evolved from artificial neural networks over the years. Also, we discussed the Perceptron method of learning, its limitations, and the current method of training neural networks. Problems pertaining to non-convex cost functions, elliptical localized cost contours, and saddle points were discussed in some detail, along with the need for different optimizers to tackle such problems. Also, in the second half of the chapter we caught up on TensorFlow basics and how to execute simple models pertaining to linear regression, multi-class SoftMax, and XOR classification through TensorFlow. In the next chapter, the emphasis is going to be on convolutional neural networks for images.