Deep neural networks and fraud detection

Yifei Lu
Deep neural networks and fraud detection

Yifei Lu
## Contents

1 Introduction 1

2 Deep Neural network 1
   2.1 The artificial neuron ........................................ 2
   2.2 Activation Function ........................................... 3
   2.3 Single-layer feedforward network ............................. 5
   2.4 Perceptron ..................................................... 6
   2.5 Deep neural network ............................................ 7

3 Estimation of deep neural network 8

4 Implementation 15
   4.1 Deep learning commercial softwares ............................. 16
   4.2 The TensorFlow programming model ............................ 16
       4.2.1 Computational graph structure .......................... 17
       4.2.2 Execution model ......................................... 18
       4.2.3 Optimizations ............................................. 19
       4.2.4 Gradient Computation .................................... 20
       4.2.5 Programming interface ................................... 21
       4.2.6 TensorBoard .............................................. 22
   4.3 Comparison with other deep learning libraries ................ 22

5 Experiments 23
   5.1 Logistic regression model ...................................... 24
   5.2 Neural network using Scikit-learn ............................ 25
   5.3 TensorFlow implementation .................................... 27

6 Conclusion 30

A Scikit-learn code 31

B TensorFlow code 32
1 Introduction

Machine learning is one subfield of artificial intelligence, which presents human intelligence by machines. Deep learning as one subset of machine learning becomes most popular research hotspot currently. It employs artificial neural network (ANN) that is an information process machine modeled on the structure and action of biological neural network in the brain [1][2]. ANN is flexible and self-adaptive to solve complex problems that are difficult described by mathematical model, such as pattern recognition and classification, function approximation and control [3]. Recent years the increasing interest in deep neural networks (DNNs), which employs many layers, has heightened the need for ANN in both industrial and academical areas. Deep neural networks learn experience from data to approximate any nonlinear relations between the input information and the final output. A well-trained deep neural network has the ability to capture abstract features over the entire data set.

The purpose of this thesis is to detect credit card fraud transactions by applying deep neural networks. Applying back-propagation algorithm to find optimal parameters, the network is trained to reach stability and be optimal so that a appropriate model can be found to detect whether the transaction made is normal or fraud. This problem of detecting fraud transactions can be regarded as a classification problem. The optimal network to perform classification on credit card data set is explored and implemented in two open source machine learning libraries, namely TensorFlow released by Google and Scikit-learn.

This thesis is organized in following way: the second chapter presents theory of neural network’s architecture, from basic processing unit to the deep neural network with many layers. The third chapter introduces the theory of training the network and related concepts. The fourth chapter presents the theory of implementation tool TensorFlow, from what is TensorFlow to how it works. In addition, we compare TensorFlow with other open source machine learning libraries. The fifth chapter describes the experiments conducted on the data set and implemented in two machine learning libraries as well as experiment results.

2 Deep Neural network

The mechanism of artificial neuron is simulated on the neuron in the brain [4]. Neuron is the basic element to process information in the brain. The brain consists of about 10 billions neurons that are interconnected to form the network. The biological neuron receives signals from other neurons and process on the information [5][6]. If the processed input signal exceeds a threshold value, the neuron fires and produces electric pulse to transfer the signals to other neurons; if the input signal is below the threshold value the neuron does not fire and no output signal is produced [7].

The following of this chapter based on the book Simon Haykin (2009) [4], presents the structure of artificial neurons and building blocks of neural networks. Then we introduce the theory of different topologies of neural networks,
starting with single-layer neural networks then perceptron. The chapter ends with theory of deep neural network.

2.1 The artificial neuron

The topologies of artificial neuron network are determined by the interconnected manner between the basic process units called neurons. An artificial neuron is the basic computational unit which performs a nonlinear function on the input signal[4]. Figure 1 shows the model of a neuron. The neuron is fed by input signal \(x_1, x_2, ..., x_m\) through a set of modifiable weights that can either strength or dampen the specific input. A signal \(x_j\) at the input of the link \(j\) connecting to the neuron \(k\) is multiplied by the weight \(w_{kj}\). The input-weight products are summed as a net input of the activation function that produces output which are restricted into some finite value. In addition, a bias denoted by \(b_k\) is also added to the neuron as a component of the net input of the activation function. The bias plays an important role when designing the network. It allows the parallel movement of activation function which increases the possibility of solving problems.

![Figure 1. Nonlinear model of a neuron](image)

The mathematical expression for the model of a neuron \(k\) is given by

\[
u_k = \sum_{j=1}^{m} w_{kj} x_j\] 

(1)

and

\[
y_k = \varphi(u_k + b_k)\] 

(2)

where \(w_{k1}, w_{k2}, ..., w_{km}\) are the connecting weights of neuron \(k\); \(u_k\) is the summation of input-weights products; \(b_k\) is the bias; \(\varphi(\cdot)\) is the activation function; and \(y_k\) is the output signal of the neuron \(k\).

The bias \(b_k\) also can be regarded as a modifiable weight equal to \(w_{k0}\) with a fixed input of \(x_0 = +1\). Thus Eq.(1) can be written as follows:

\[
u_k = \sum_{j=0}^{m} w_{kj} x_j\] 

(3)
then
\[ y_k = \varphi(v_k) \quad (4) \]

### 2.2 Activation Function

The activation function performs nonlinear transformation on the input signal for the sake of controlling the activation to the output, hence an infinite range of input can be transferred into specific range of output value[8]. Several most common activation functions are presented below:

1. **Threshold Function**  
   A threshold function describing the output of neuron \( k \) is given by
   \[
   y_k = \begin{cases} 
   1, & \text{if } v_k \geq 0 \\
   0, & \text{if } v_k < 0.
   \end{cases}
   \]
   where \( v_k \) is the net input of the neuron \( k \):
   \[
   v_k = \sum_{j=1}^{m} w_{kj}x_j + b_k
   \]
   For threshold function, shown by Fig.2, the output of the neuron takes value of 1 if that neuron’s input is positive and 0 otherwise.

![Figure 2. Threshold function](https://en.wikibooks.org)

2. **Sigmoid Function**  
   Sigmoid function is the commonly used activation function which has a “S” shape. It is a strictly monotonic increasing function which shows good balance between linear and nonlinear property[4]. One example of the sigmoid function is the logistic function shown as Fig.3, defined by
   \[
   f(v) = \frac{1}{1 + e^{-av}}
   \]
   where \( a \) is the slope parameter of the sigmoid function and it equals \( \frac{4}{a} \) at the origin. A logistic sigmoid function restrict the value of output to a continuous range from 0 to 1 representing the probability of binary classification. It is
an acceptable mathematical representation of a biological neuron model which shows if a neuron fires or not. As $a$ approaches to infinity, the sigmoid function becomes a threshold function. Moreover, sigmoid function is differentiable which are essential properties for the theory of training the neural network. One fatal flaw of sigmoid function, however, is its saturation to "kill" the gradient. When the output is at the tail of 0 or 1(saturation) the gradient of neuron approaches to zero, which makes the network hard for training. The details are shown in next chapter.

3. **Hyperbolic tangent function** Hyperbolic tangent is more like to logistic sigmoid function except that it gives output from -1 to 1. As shown in Fig.4, it is bounded and differentiable. The mathematical definition of hyperbolic tangent function is given by

$$f(v) = \tanh(v) = \frac{e^v - e^{-v}}{e^v + e^{-v}} = \frac{e^{2v} - 1}{e^{2v} + 1}$$

4. **Rectified Linear Units (ReLU)** Rectified Linear Units are the new popular trend used for training the deep neural networks in recent years.
The rectifier is linear when the input is positive and zero otherwise, defined as

\[ f(v) = \max(0, v) \]

where \( v \) is the input to a neuron. Compared with sigmoid activation function, ReLU solves training procedure faster due to its non-saturating nonlinearity[11]. Furthermore, ReLU can be implemented by simply thresholding a matrix of activations at zero compared with sigmoid function which causes expensive computations.\(^1\)

Another type of ReLU is softplus function, a smooth approximation to the rectifier, defined by \( f(v) = \ln(1 + e^v) \).

![Figure 5. Rectified Linear Units (ReLU) activation function](https://en.wikipedia.org)

2.3 Single-layer feedforward network

According to the topological structure of the connections between neurons, the neural network can be divided into different types. One is feedforward network with data entering from input layer transferred in one direction to output layer and no cycle between the neurons. Single-layer neural network is the simplest feedforward network consisting of a single output layer meaning data going from input layer directly to output layer through weights. Figure 6 shows a single-layer feedforward network.

![Figure 6. Single-layer feedforward network](http://lib.csdn.net)

\(^1\)S231n Convolutional Neural Networks for Visual Recognition [http://cs231n.stanford.edu]
2.4 Perceptron

The perceptron proposed by Rosenblatt (1958) is the simplest neural network for linearly separable classification. The perceptron is a single neuron with adjustable weights and bias. Rosenblatt has proved that the learning algorithm of perceptron can be convergent to find out a hyperplane as a decision boundary to separate the training examples that can be classified into two linearly separable classes\[4]. This algorithm is known as perceptron convergence theorem. By including more than one neuron in the output layer of the perceptron, the multiple perceptron can classify more classes which are linear separable.

The structure of the perceptron is shown as Figure 6. The neuron is connected to a set of inputs by corresponding adjustable weights. The summed input-weight products is taken as input of threshold activation function. The introduced bias adds one more free parameter to make the network output more easily to reach the expected target. The output of perceptron is classified according to the net input. If the net input \( v_k = \sum_{i=1}^{m} w_i x_i + b \) is positive the neuron produces an output of +1, and -1 if it is negative. Therefore the decision of classification can be presented by a hyperplane which is defined by

\[
\sum_{i=1}^{m} w_i x_i + b = 0
\] (7)

When the parameters of the network are determined, the trajectory of \( \sum_{i=1}^{m} w_i x_i + b = 0 \) can be drawn as a classification boundary in the space composed of the input vectors \( x_1, x_2, ..., x_m \). For any given input vector through the weights and bias, it is either located above or below the hyperplane to be classified. Figure 7 shows two-dimensional input vector space where the hyperplane is a straight line. The points above the boundary line belong to Class 1 and other points lying below the boundary line belong to class 2. The boundary line can be shifted from the origin by the bias.

![Figure 6. Signal-flow graph of the perceptron](http://sebastianraschka.com)

To produce desired classification of the input vectors, the perceptron’s optimal weights and bias can be learned from training iteratively on the dataset. As mentioned above, Roseblatt has proved that if the training examples are
picked from two linearly separable classes, the algorithm of updating weights is convergent. Denote the time step of performing the algorithm as \( n \). The weights of the perceptron can be adjusted according to the error-correction rule which is defined by

\[
\mathbf{w}(n+1) = \mathbf{w}(n) + \eta(d(n) - y(n))\mathbf{x}(n)
\]  

(8)

where \( \eta \) is called learning rate which is a fixed parameter between 0 and 1, \( d(n) \) is the desired value and \( y(n) \) is the actual value. A single perceptron can only function on simple pattern classification for two classes entirely intersected by a hyperplane[4]. The nonlinearly separable pattern classification is out of the computational ability of perceptron.

![Figure 7. Hyperplane as decision boundary for a two-dimensional, two-class pattern-classification problem](image)

2.5 Deep neural network

Due to practical limitation of single-layer network on the linear separable problem, deep neural network was introduced to solve an arbitrary classification problem[4]. It contains one or more hidden layers whose computational nodes are called hidden nodes. The depth of the model refers to the number of hidden layers. Figure 8 shows the topology of deep neural network with two hidden layers and an output layer. The input information enters the first hidden layer and the outputs of this layer are transferred as inputs to the second hidden layer and so on. Each layer receives outputs from previous layer as inputs, thus the input signal propagates forward on layer-by-layer base until the output layer. An error signal is produced at neurons in the output layer which is propagated backward through the network.

![Figure 8. Structure of deep neural network with two hidden layers](image)

Source: www.mql5.com
Each neuron in hidden and output layers performs two operations. One is to compute the differentiable activation function on the input vectors and weights and spread the output forward through the hidden layers. The second operation is to compute the gradient of error with respect to weights connected to the inputs of that neuron, which flows backward through the network.

Deep neural network can be used for pattern recognition and classification. Several key concepts are introduced below based on the website deeplearning4j.org. The input layer consists of the components of a feature vector to be classified. Each hidden layer of neurons processes on a different set of features that are output of the previous layer. The more hidden layers the network has, the more complicated features that can be detected by neurons since they collect and combine information generated by previous layer. The nonlinearity from training data gives the network greater computational power and can implement more complex functions than network without hidden neurons.

3 Estimation of deep neural network

This chapter introduces estimation of deep neural network based on Simon Haykin(2009). The learning process of neural network is to estimate the optimal weights parameters through training on the data set to obtain expected output. Training begins with initializing randomly small weights. To ensure the network learn normally, the weights need to be initialized with different values. Choosing large weights would result in saturation of the network. In general, an effective way to initialize weights is to randomly pick values from a uniform distribution. Learning is then carried out with feeding input data into the network to produce the output. The performance of neural network is measured by the difference between the actual output and the desired output. This difference is described as loss function which can be minimized by adjusting weights.

When the network learns from a data set whose category label is known, it employs supervised learning. Each training example in the training set has a pair of data, an input vector and corresponding expected output. The network learns the function using labeled training set therefore it can map the unlabeled input into the correct output to perform classification or regression[12][13]. For training on the unlabeled dataset it is referred to unsupervised learning which will be not discussed here.

Consider a training set denoted as \( D = \{(x^1, d^1), \ldots, (x^N, d^N)\} \) is used to train the network with one or more hidden layers. Denote the output of neuron \( j \) as \( y_j \) in the output layer produced by data \( x(n) \) in the input layer, the corresponding loss function produced by the output at neuron \( j \) is calculated by

\[
e_j(n) = d_j(n) - y_j(n)
\]

For convenience of taking derivative of loss function we add a factor of \( \frac{1}{2} \) to \( e \) and summing the loss of all neurons in the output layer, then total loss of the whole network is expressed by

\[
E_n(w) = \frac{1}{2} \sum_{j \in C} e_j^2(n)
\]
where the set \( C \) contains all the neurons in the output layer. With the training set consisting of \( N \) examples, the average loss over the entire training set is defined by

\[
E_{av}(w) = \frac{1}{N} \sum_{n=1}^{N} E(w) = \frac{1}{2N} \sum_{n=1}^{N} \sum_{j \in C} e_j^2(n)
\] (11)

One important point is that \( E \) is a function of all the modifiable weights, i.e., free parameters of the network.

**Batch learning and on-line learning**

There are two different methods to implement supervised learning of neural network, namely batch learning and on-line learning[4]. Batch training, also called standard gradient descent, performs adjustments in weight space after presenting all the examples in the training set that creates one epoch of training. In other words, the loss function of batch learning is given by \( E_{av}(w) \) and the gradient is calculated for the entire training set. In the on-line method of supervised learning, each example is chosen randomly from the training set and a step of adjustment in the weight space is performed for each example presented to the network. Therefore the loss function to be minimized is \( E(w) \).

In practice, batch learning leads to extra computations for the larger data set, since there exist similar examples to compute gradients (described in the following section)[14]. Therefore stochastic gradient descent (SGD) was proposed to compute gradient for each subset of the whole training set. Each step we take \( M \) random samples from \( N \) training examples to compute the gradient then update weights. Therefore the loss function to be optimized becomes:

\[
E_{av}(w) = \frac{1}{2M} \sum_{m=1}^{M} \sum_{j \in C} e_j^2(m)
\] (12)

where \( M \) is an integer from 1 to \( N \). SGD is the most commonly used optimization method, which results in a faster training procedure and can also be used as on-line learning[14].

**Back-propagation algorithm**

Back-propagation algorithm is one of the simplest and most widely used methods for training supervised neural networks when calculating gradient of the loss function. Gradient is the slope of the loss function in weight space showing that how the loss function varies as weights change. The loss (error) function produced by current weights is propagated backward through the network to update the weights. As the network learns from its mistakes for every training iteration, it adjusts weights iteratively to reduce the loss function.

As shown in Fig.9, we start searching on the loss surface from initial weights then move a step along the opposite direction of the gradient and the size of step is determined by both learning rate and the slope of the gradient[14]. In mathematical way, the rule of updating weights is defined by

\[
w = w - \eta \nabla E_{av}(w)
\] (13)
where $\eta$ is the learning rate, a fixed parameter between 0 and 1. The smaller learning rate gives smaller changes in the weight space and longer training time to reach the minimum of loss function. Larger learning rate, however, results in an unstable result. The gradient $\nabla E = (\frac{\partial E}{\partial w_1}, \ldots, \frac{\partial E}{\partial w_n})$ is calculated using back-propagation algorithm.

Consider neuron $j$ fed by a set of inputs $y_1(n), y_2(n), \ldots, y_m(n)$ and the net input of activation function produced at neuron $j$ is

$$v_j = \sum_{i=0}^{m} w_{ji}(n)y_i(n) \tag{14}$$

where $w_{j0}$ corresponding to the fixed input $y_0 = +1$ equals the bias $b_j$ connected to neuron $j$. Therefore the output of activation function $f$ produced at neuron $j$ is

$$y_j(n) = f_j(v_j(n)) \tag{15}$$

According to the chain rule we present this gradient as

$$\frac{\partial E(n)}{\partial w_{ji}(n)} = \frac{\partial E(n)}{\partial e_j(n)} \frac{\partial e_j(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)} \frac{\partial v_j(n)}{\partial w_{ji}(n)} = -e_j(n)f_j'(v_j(n))y_i(n) \tag{16}$$

where we multiply the derivatives of loss of the whole network $E(n) = \frac{1}{2}e_j^2$, the loss signal $e_j(n) = d_j(n) - y_j(n)$, the active function $y_j(n)$ and the net input $v_j(n)$.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{gradient.png}
\caption{Schematic of gradient descent including two weights}
\end{figure}

Source: http://www.webpages.ttu.edu

The adjustment $\Delta w_{ji}(n)$ to $w_{ji}(n)$ is defined by the delta rule:

$$\Delta w_{ji}(n) = -\eta \frac{\partial E(n)}{\partial w_{ji}(n)} \tag{17}$$

where again $\eta$ presents the learning rate, and the minus sigh shows that the weights are changed in a direction to reduce the loss. Plug Eq.(16) into Eq.(17) gives

$$\Delta w_{ji}(n) = \eta \delta_j(n)y_i(n) \tag{18}$$
where the local gradient $\delta_j(n)$ is given by

$$\delta_j(n) = e_j(n)f'_j(v_j(n)) \quad (19)$$

To calculate the weight adjustment $\Delta w_{ji}(n)$ the error signal $e_j(n)$ at the output of neuron $j$ is needed. There are two different cases based on whether neuron $j$ is on the output layer or the hidden layer.

**Case 1** When neuron $j$ is an output node, the error $e_j(n)$ of neuron $j$ calculated by Eq.(9) gives an explicit computation of local gradient $\delta_j(n)$ using Eq.(19).

**Case 2** When neuron $j$ is in the hidden layer, the local gradient for output at neuron $j$ is defined by

$$\delta_j(n) = -\frac{\partial E(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)} f'_j(v_j(n)) \quad (20)$$

where the second line uses Eq.(15). The error signal for a hidden neuron $j$ contributes to the errors of all neurons connected to $j$ on the next layer, thus the loss is given by $E(n) = \frac{1}{2} \sum_{k \in C} e_k^2(n)$ where $k$ is an output neuron. Differentiating the loss function with respect to $y_j(n)$ gives

$$\frac{\partial E(n)}{\partial y_j(n)} = \sum_k e_k \frac{\partial e_k}{\partial y_j} \frac{\partial y_j}{\partial v_k} \frac{\partial v_k}{\partial y_j} \quad (21)$$

The error at neuron $k$ is

$$e_k(n) = d_k(n) - y_k(n) = d_k(n) - f_k(v_k(n)) \quad (22)$$

Thus,

$$\frac{\partial e_k}{\partial v_k} = -f'_k(v_k(n)) \quad (23)$$

The net input at neuron $k$ is

$$v_k(n) = \sum_{j=0}^m w_{kj}(n)y_j(n) \quad (24)$$

where $m$ is the number of inputs excluding the bias entering into neuron $k$ and $w_{k0} = b_k$ with the fixed input of value +1. Thus

$$\frac{\partial v_k}{\partial y_j} = w_{kj}(n) \quad (25)$$

By using Eq.(23),Eq.(25) in Eq.(21), we get

$$\frac{\partial E}{\partial y_j(n)} = -\sum_k e_k f'_k(v_k(n)) w_{kj}(n) \quad (26)$$
Using Eq.(26) in Eq.(20) we get the back-propagation formula for the local gradient $\delta_j(n)$ at hidden neuron $j$ is given by

$$\delta_j(n) = f'_j(v_j(n)) \sum_k \delta_k(n)w_{kj}(n)$$  \hspace{1cm} (27)$$

To summarize the relationships derived from the back-propagation algorithm, first we note that the adjustment for weight $\Delta w_{ji}(n)$ is defined by the delta rule:

$$\Delta w_{ji}(n) = \eta \delta_j(n)y_i(n)$$  \hspace{1cm} (28)$$

The local gradient $\delta_j(n)$ is different depending on whether neuron $j$ is in the output layer or hidden layer. If neuron $j$ is an output node, the local gradient is calculated by $\delta_j(n) = e_j(n)f'_j(v_j(n))$; if neuron $j$ is a hidden node, $\delta_j(n)$ is given by $\delta_j(n) = f'_j(v_j(n)) \sum_k \delta_k(n)w_{kj}(n)$.

**Forward and backward passes**

Haykin(2009) claimed that there are two different information transmission when applying back-propagation algorithm, namely the forward and backward passes. In the forward pass, the input vector combined with the weights enters the first hidden layer; then the output is transfered to the second layer as input. The output for neuron $j$ is calculated as $y_j(n) = f(\sum_{i=0}^{m} w_{ji}(n)y_i(n))$ where $m$ represents the number of neurons in previous layer; $f$ is activation function of neuron $j$; $w_{ji}$ is the weight connecting neuron $i$ to neuron $j$; $y_i$ is the input of neuron $j$. If neuron $j$ is in the first hidden layer, then $y_i$ represents the $i$th element of input vector; if $j$ is on the output layer, then $y_j$ is the $j$th output neuron in the output layer. The input information is propagated layer by layer until the output layer to produce an error signal for each neuron at output layer calculated by the difference of output value and actual value. Note that the weights are fixed under the forward propagation.

The backward pass starts from the output layer by propagating the loss backward through the network and calculating the local gradient $\delta$ for each neuron recursively. The weights will be adjusted according to the delta rule of Eq.(28) recursively in the backward pass. If neuron $j$ is on the output layer the local gradient equals the product of the derivative of activation function corresponding to neuron $j$ and the error signal at neuron $j$. Then updates for weights connecting to neurons in the output layer can be calculated directly by Eq(28). Next we determine the local gradients for neurons in the second last layer according to Eq.(27) then update all weights fed into that neuron. The changes in weights are computed recursively and propagated backward until all weights are updated in the network. It is noted that the original weights are used to perform back-propagation before updating all weights in the network.

**Numerical example of back-propagation**

We use a numerical example shown in Figure 10 with two layers to illustrate back-propagation algorithm. The input vector is $x = [0.1, 0.5]$ with desired output $d = 1$. The sigmoid activation function is applied on hidden and output layers and assume that learning rate $\eta$ is 0.1. Denote $w_{ji}^{(l)}$ is the weight connecting neuron $i$ to neuron $j$ in layer $l$. 


The initial weight values is as follows:

<table>
<thead>
<tr>
<th>$w_{i}^{(0)}$</th>
<th>$w_{j}^{(0)}$</th>
<th>$w_{k}^{(0)}$</th>
<th>$w_{l}^{(0)}$</th>
<th>$w_{m}^{(0)}$</th>
<th>$w_{n}^{(0)}$</th>
<th>$w_{o}^{(0)}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>0.2</td>
<td>0.3</td>
<td>-0.1</td>
<td>-0.5</td>
<td>0.4</td>
<td>0.3</td>
</tr>
</tbody>
</table>

The net input for the first neuron in the hidden layer:

$net_1^{(1)} = w_{11}^{(1)} * x_1 + w_{12}^{(1)} * x_2 + w_{10}^{(1)} * b_1$

$net_1^{(1)} = 0.2 * 0.1 + 0.3 * 0.5 + 0.1 * 1 = 0.27$

We then apply sigmoid function to get the output:

$out_1^{(1)} = \frac{1}{1 + e^{-net_1^{(1)}}} = 0.567$

Carrying out the same process for the second neuron in the hidden layer we get:

$out_2^{(1)} = \frac{1}{1 + e^{-net_2^{(1)}}} = 0.512$

We repeat the process to the output layer neuron using outputs at hidden layer as input:

$net_2^{(2)} = w_{11}^{(2)} * out_1^{(1)} + w_{12}^{(2)} * out_2^{(1)} + w_{10}^{(2)} * b_2$

$net_2^{(2)} = -0.2 * 0.567 + 0.1 * 0.512 + 0.3 * 1 = 0.2378$

$out_2^{(2)} = \frac{1}{1 + e^{-net_2^{(2)}}} = 0.56$

The error at output layer neuron is $d - out_2^{(2)} = 0.44$. We know the derivative of sigmoid function $\sigma$ is $\sigma * (1 - \sigma)$. The gradient for output layer neuron is

$\delta_2^{(2)} = \sigma'(net_2^{(2)}) * (d - out_2^{(2)}) = \sigma(net_2^{(2)}) * (1 - \sigma(net_2^{(2)})) * (d - \sigma(net_2^{(2)}))$

$\delta_2^{(2)} = 0.56 * (1 - 0.56) * 0.44 = 0.108$

The error derivative of neuron in the output layer is $\frac{\partial E}{\partial w_{kj}} = \delta_2^{(2)} * out_1^{(1)}$ with $k = 1$. Thus by the delta rule the updated weights connecting to the output layer neuron are

$w_{11}^{(2)} = -0.2 - (0.1)(0.108)(0.567) = -0.206$

$w_{12}^{(2)} = 0.1 - (0.1)(0.108)(0.512) = 0.094$

$w_{10}^{(2)} = 0.3 - (0.1)(0.108) = 0.289$

The gradient for hidden layer neurons are given by

$\delta_j^{(1)} = \sigma'(net_j^{(1)}) \sum_k w_{kj}^{(2)} \delta_k = [\sigma(net_j^{(1)})(1 - \sigma(net_j^{(1)}))] \sum_k w_{kj}^{(2)} \delta_k$

$\delta_1^{(1)} = (0.567)(1 - 0.567)(-0.2)(0.108) = -0.0053$

$\delta_2^{(1)} = (0.512)(1 - 0.512)(0.1)(0.108) = 0.0027$

The error gradient for neurons in the hidden layer is given by $\frac{\partial E}{\partial w_{ij}} = \delta_j^{(1)} * x_i$.

The updated weights of connections fed into hidden layer are

}\]
\[ w_{11}^{(1)} = 0.2 - (0.1)(-0.0053)(0.1) = 0.20005 \]
\[ w_{12}^{(1)} = 0.3 - (0.1)(-0.0053)(0.5) = 0.30027 \]
\[ w_{13}^{(1)} = -0.5 - (0.1)(-0.0027)(0.1) = -0.49997 \]
\[ w_{21}^{(1)} = 0.4 - (0.1)(-0.0027)(0.5) = 0.40014 \]
\[ w_{10}^{(1)} = 0.1 - (0.1)(-0.0053) = 0.10053 \]
\[ w_{20}^{(1)} = -0.1 - (0.1)(-0.0027) = -0.09973 \]

**Rate of learning**

As seen from the delta rule, the learning rate reflects the size of changes in weights. Smaller learning rate gives smaller change for weights between iterations which results in a smoother path of movement in the weights space. This, however, gives a longer training time to reach the local minimum on the error surface. The larger learning rate gives an unstable result of the network. In order to avoid instability with increasing learning rate, the delta rule for updating weights is modified by the generalized delta rule as follows:

\[ \Delta w_{ji}(n) = \alpha \Delta w_{ji}(n - 1) + \eta \delta_j(n) y_i(n) \]  

(29)

where \( \alpha \) is a positive number called the momentum constant, which describes the adjustment of weight \( w_{ji} \) at iteration \( n \) and it at iteration \( n-1 \). Eq.(29) can be seen as a time series with index \( t \) from the initial time to the current time \( n \). By solving this equation we get:

\[ \Delta w_{ji}(n) = \eta \sum_{t=0}^{n} \alpha^{n-t} \delta_j(t) y_i(t) \]  

(30)

With the help of error derivatives \( \frac{\partial E(n)}{\partial w_{ji}(n)} \) from previous equations we get

\[ \Delta w_{ji}(n) = -\eta \sum_{t=0}^{n} \alpha^{n-t} \frac{\partial E(t)}{\partial w_{ji}(t)} \]  

(31)

**Early-Stopping of Training**

The goal of training a neural network for classification problem is to obtain a good performance on the unseen data which has the same distribution as the training data in the same data set[15][16]. When a neural network learns training data excessively, the network will concentrate on finding features that only exit in the training data rather than a true underlying pattern[4]. This lead to a poor generalization which is referred to overfitting[17].

To improve generalization and avoid overfitting, an optimal stopping time of training is required. Because the weights are initialized with random values, the error is large on the training set. As the amount of training iteration increases, the error decreases dramatically then continues to decrease slowly before the network reaching the required minimum error of stopping training. However as the training error gradually decreasing to a stable value the error on the unseen data increases. Therefore the optimal generalization usually appears before training error reaching the local or global minimum. To find the
optimal stopping time, the data are divided into disjoint sets: a training set, an validation set not used for training and a test set. After every period, for instance every three epochs, of training is done with fixed weights and bias, the network is then tested on the validation set. The error of each example in the validation set is evaluated as validation error. Training is stopped when a minimum of validation error is reached then weights in the network are used as solutions. This procedure is called the early-stopping method of training which is widely used in practice. Figure 11 shows estimation(training) learning curve and validation learning curve presenting average error of validation set.

\[ L(w) = \frac{1}{2} \sum_{i=0}^{m} (d^i - y^i)^2 + \frac{\lambda}{2m} \sum w^2 \]  \hspace{1cm} (32)

where \( w \) is neural network parameters, \( m \) is the number of training examples contained in the training set, and \( \lambda \) is a regularization parameter to control the tradeoff between two terms in the loss function. This regularization approach is referred to \( L_2 \) regularization(weight decay). Adding regularization term can limit the range of parameters not too large therefore it can reduce overfitting to a certain extent.

## 4 Implementation

The neural network learning algorithms and model structure expressed in mathematics need to be realized by computer programs for the real word use[20].
Therefore a series of open source commercial machine learning software libraries for particularly deep learning which employs deep neural network with huge numbers of hidden layers are invented. We will present several commercial softwares such as Torch, Theano, Caffe and Tensorflow, based on the article written by Peter Goldsborough(2016).

4.1 Deep learning commercial softwares

**Torch** released in 2002[21], is the oldest machine learning library for training deep neural network. It can implement a variety range of advanced learning algorithms in one integrated framework. **Torch** was originally implemented in C++, today it uses Lua language as frontend. It includes four basic classes: dataset, trainer, machine and measure. The Trainer sends input produced by Dataset to Machine that can produce an output, which is used to modify the Machine. During the training process Measures can evaluate the performance of the model.

**Theano** was released under the BSD license in 2008 and is developed by the LISA Group (now MILA) at the University of Montreal, Canada[22]. It is specifically designed for large-scale computations required for deep neural network. **Theano** is a mathematical compiler written in Python library. It defines mathematical expression as symbolic representation, such as computational graph, which allows symbolic differentiation of intricate expressions. The expressions are optimized and translated into C++, then compiled and computed on CPU or GPU devices efficiently. It is one of the pioneers and most popular among deep learning libraries.

**Caffe** released in 2014 under a BSD-License is an open source framework for deep learning algorithm and maintained by the Berkeley Vision and Learning Center[23]. The code is implemented in C++ with CUDA computed on GPU device. The computation task is designed on the basis of the network layers. **Caffe** especially is applied on training and deploying convolutional neural networks(CNNs), and widely used for image recognition.

4.2 The TensorFlow programming model

One emerging open source deep learning software library is TensorFlow released by Google in 2015 which is used for defining, training and deploying deep neural network. In this section we present the basic structure of TensorFlow and explain the way how the machine learning algorithm is expressed by the computational graph in TensorFlow. Next we discuss the execution model for how to realize the computational graph on the basic process devices. Then we investigate several algorithm optimizations built into TensorFlow according to both software and hardware. Subsequently we discuss extension to the basic programming model. Lastly the programming interface and visualization tool of TensorFlow are introduced.
4.2.1 Computational graph structure

TensorFlow is a programming system using the computational or dataflow graph to represent computation task (learning algorithms). The computational graph consists of nodes and directed edges connecting to nodes. The nodes represent operations and edges represent data flow between operations. The principle components of computational graph, namely operations, tensors, variables and sessions are presented below.

1. Operations:
   In TensorFlow, nodes represent the operations. More precisely, nodes describe how the input data flow through them in the directed graph. An operation can obtain zero or many inputs then produce zero or many outputs. Such an operation can be a mathematical equation, a constant or variable. Figure 3.1 shows examples of operations in TensorFlow. The constant is obtained by an operation taking no inputs and producing the output the same as corresponding constant. Similarly, a variable is an operation taking no inputs and producing the current value of that variable. Any operation needs to be implemented by its kernel that can be executed on hardware device such as a CPU or GPU.

<table>
<thead>
<tr>
<th>Category</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Element-wise operations</td>
<td>Add, Mul, Exp</td>
</tr>
<tr>
<td>Matrix operations</td>
<td>MatrixInv, MatrixInverse</td>
</tr>
<tr>
<td>Value-producing operations</td>
<td>Constant, Variable</td>
</tr>
<tr>
<td>Neural network axes</td>
<td>SoftMax, Relu, Conv2D</td>
</tr>
<tr>
<td>Checkpoint operations</td>
<td>Save, Restore</td>
</tr>
</tbody>
</table>

*Figure 3.1 Examples of operation in TensorFlow*

Source: Peter Goldsborough (2016)

2. Tensors:
   In TensorFlow, the data are represented by tensors flowing between nodes in the computational graph. A tensor is a multi-dimensional array with a static type and dynamical dimensions. The number of dimensions of a tensor is referred as its rank. A tensor’s shape describes the number of components in each dimension. In the computational graph, when creating an operation, a tensor is returned which will be sent by a directed edge as input to the connected operation.

3. Variables:
   When performing stochastic gradient descent, the computational graph of the neural network is executed iteratively for single experiment. Across evaluations of training, most tensors do not survive whereas the state of the model such as weights and bias need to be maintained. Hence variables are added to the computational graph as special operations. The variables saving tensors are persistently stored in in-memory buffers. The value of variables can be loaded when training and evaluating the model. When creating a variable, it needs to be supplied with a tensor as its initial value upon execution. The shape and data type of that tensor automatically becomes the variable’s shape and type. The initialization of variables must be executed before training. This can be done with adding an operation to initialize all variables and executing it before training the network.

4. Sessions:
The execution of operations and evaluation of tensors are performed in the context of *session*. The Session uses a *Run* routine as entry for executing the computational graph. With invocation of run, the input is taken into the computational process of entire graph to return output according to the graph definition. The computation graph will be executed repeatedly for training the network with invoking *Run* routine. The session distributes the operations of the graph to devices such as CPU or GPU on different machines according to TensorFlow’s *placement algorithm* which will be presented later. In addition, the ordering of node executions are defined explicitly, namely control dependencies. Evaluating the model ensures that the control dependencies are maintained.

### 4.2.2 Execution model

The task of executing the computational graph is divided into four groups: the client, the master, workers and a set of devices. The client sends the request of executing the graph via run routine to the master process, which is responsible for assigning tasks to a set of workers. Each worker is responsible for monitoring one or more devices which are physical entities to implement the kernels of an operation. Based on the numbers of machine, there are two ”version” of TensorFlow, namely local and distributed implementation. Local system is that the nodes are executed on a single machine; distributed system is to implement nodes on many machines with many devices.

*Devices* are the basic and smallest physical units in TensorFlow to execute the operations. The kernel of nodes will be assigned to available devices such as CPU or GPU to be executed. Furthermore, TensorFlow allows new physical implementation units registered by users. To monitor the operations on a device, each worker process is responsible for one or more devices on a single machine. The device’s name is determined by its type and the index of the worker group where it is located.

*Placement Algorithm* is to determine which nodes will be assigned to which device. The algorithm simulates execution of the graph from the input tensor to the output tensor. The algorithm adopts a *cost model* $C_v(d)$ to determine on which device $D = \{d_1, \ldots, d_n\}$ to execute a particular operation. The optimal device is determined by the minimum of cost model $d = \arg \min_{d \in D} C_v(d)$ to place a given node during the training.

*Cross-Device execution*: Tensorflow usually assigns nodes to different devices as long as the user’s system provides multiple devices. This process classifies nodes then assigns nodes belonging to the same class to one device. Therefore it is necessary to deal with the problem of node dependencies allocated across devices. Consider two such devices where nodes $A$ is on device $A$. If the tensor $v$ produced by operation $A$ is transmitted as inputs to two different operations $\alpha$ and $\beta$ which are on device $B$, then there exit edges $v \rightarrow \alpha$ and $v \rightarrow \beta$ across from device $A$ to $B$, as shown in figure 2(a):
In practice, the process of transmitting output tensor of $v$ from device $A$, such as GPU, to device $B$ such as CPU are completed by producing two kinds of nodes, namely send and recv, shown in figure 2(b). At last TensorFlow uses "canonicalization" to optimize (send,recv) pairs which is an equivalent but more effective approach compared with the example shown above. The two recv nodes that connect to $\alpha$ and $\beta$ on device $B$ are replaced by only one recv node receiving the output from $v$ transmitted to two dependent nodes $\alpha$ and $\beta$.

4.2.3 Optimizations

To ensure the efficiency and performance of TensorFlow execution model, some optimizations are constructed in the library. Common subgraph elimination performed in TensorFlow is to canonicalize the same type of operations on an identical input tensor to a single operation when traversing the computational graph. The output tensor is then transferred to all dependent nodes. Another optimization, namely scheduling, is to perform nodes as late as possible which ensures that the result of operations remains for the required minimum time in memory. This effectively reduces the memory consumption and improves the performance of the model. Loss compression optimization refers to add conversion nodes to the graph. An optimized robust model does not change the response output due to the variation in noise signals. Therefore, the requirement for the precision arithmetic of the algorithm is reduced. Based on this principle, when the data communicate across the devices or machines, a 32-bit floating-point representation at the sender is truncated into a 16-bit representation by the conversion node, and then converted to 32-bit at the receiving end by simply adding zeros.
4.2.4 Gradient Computation

In this section we describe one advanced feature that is extent to the basic TensorFlow programming model which was presented in section 3.2.1.

The machine learning algorithms require to calculate the gradient of specific nodes with respect to one or many nodes in the computational graph. In the neural network, the gradient of cost with respect to weights should be computed given the training example which is fed into the network. The back-propagation algorithm which is discussed in Chapter 2 is used to compute the gradient reversely starting from the end of the graph.

There are two methods calculating back-propagating gradients through a computational graph in [24]. The first is referred to symbol-to-number differentiation. Inputs are propagated forward through the graph to compute cost function, then gradient is explicitly calculated via chain rule in opposite direction through the graph. Another method more adopted in TensorFlow is namely symbol-to-symbol derivatives which computes gradient automatically rather than explicitly applying back-propagation algorithm. In such a way, special nodes are added to the computational graph to compute the gradient of each operation included in the chain. To realize back-propagation algorithm, the execution of gradient nodes are just like other nodes by launching the graph evaluation engine. This method provides a symbol handle to compute derivatives instead of calculating derivatives as numerical values. It can be explained specifically as follows:

The gradient of a particular node \( v \) with respect to other tensor \( \alpha \) is computed backward from \( v \) to \( \alpha \) through the graph. This extends the graph by adding the gradient node to each operation \( o \) which is a function of \( \alpha \) encountered in the chain \( (v \circ \cdots \circ o \circ \cdots)(\alpha) \) producing the output tensor. Hence TensorFlow adds the gradient node for each such operation \( o \) by multiplying the derivative of its outer function with its own derivative. The procedure is reversely calculated to the end node that producing a symbolic handle to the desired gradient \( \frac{dv}{d\alpha} \) which implicitly performs back-propagation method. Note that the symbol-to-symbol differentiation is just another operation without any exception.

In [25], the symbol-to-symbol differentiation may produce considerable computational cost as well as increasing memory overhead. The reason is that there exits two different equations for applying chain rule. The first equation reuses the previous computations which requires longer time to be stored than needed for the forward propagation. The chain rule is applied shown below:

\[
\frac{df}{dw} = f'(y) \cdot g'(x) \cdot h'(w) \tag{33}
\]

with \( y = g(x), x = h(x) \). The alternative way to express chain rule is given by equation (2):

\[
\frac{df}{dw} = f'(g(h(w))) \cdot g'(h(w)) \cdot h'(w) \tag{34}
\]

It shows that each function needs to recalculate all of its arguments and invokes
every inner function it depends on. Currently TensorFlow applies the first approach based on the article [25]. Considering a chain that has thousands of nodes, then recomputing the inner-most function for almost every operation in the link seems not sensible. On the other hand, it is not optimal to store tensors for long time on the device, especially such as GPU which memory resource is scarce. For equation (2), theoretically the tensor stored on the device is freed as soon as it is processed by the graph dependencies. Therefore according to the development team of TensorFlow [25], recomputing some tensor rather than store them on the device may be improved in the future work.

4.2.5 Programming interface

After discussing TensorFlow's computational model, we focus on more practical programming interface. We will discuss the available language interface and summarize the high-level abstraction of Tensorflow API and how to quickly create machine learning algorithm prototypes.

TensorFlow supports C++ and Python programming languages that allow users to call backend functionality. Currently, TensorFlow’s Python programming interface is easier to use, since it provides a variety of functions to simplify and complete the construction and execution of computational graph that have not yet been supported by C++. It is important to note that Tensorflow’s API and NumPy which is a library for scientific computing with Python are well integrated. Thus, we can see that TensorFlow tensors and NumPy ndarrays are interchangeable in many situations.

TensorFlow program consists of two distinct phases: a construction phase and an execution phase. In the construction phase, the operations in computational graph is created to represent the structure and learning algorithms of the neural network. Then in the execution phase the operations in the graph are executed repeatedly to train the network.

For the deep neural network with huge number of hidden layers, the steps of creating weights and bias, computing matrix multiplication and addition, applying nonlinear activation function are not efficient. Therefore a number of open source libraries are proposed to abstract and package these steps as well as building high-level blocks such as entire layers at one time. One abstraction library is PrettyTensor developed by Google which can provide high-level interface to the TensorFlow API. It allows the user to wrap the Tensorflow operations and tensors into a clean version, and then quickly connect to any layers in series. TFLearn is another abstraction library built based on the TensorFlow, which can be used mixed with TensorFlow code. It provides highly packaged network layers to quickly construct computational graph and allows for chaining network layers rapidly. Moreover, compared with PrettyTensor using tf.Session to train the model, TFLearn can directly add training examples and corresponding labels to train a model easily.
4.2.6 TensorBoard

Deep learning usually employs complex networks. In order to apply and debug such a complicated network, a strong visualization tool is required. TensorBoard is a web interface that visualizes and manipulates the computational graph built in TensorFlow. The main feature of TensorBoard is to construct a clear and organized visual interface for the computational graph with complicated structure and many layers to make it easily to understand how the data flows through the graph. One important visual classification method in Tensorflow is Name scopes. It can group the operations, the input, output and relationships belonging to one name scope into one block such as a single network layer, which can be expanded interactively to show details of the block.

Furthermore TensorBoard can trace changes of a single tensor during the training. The summary operations are added to the nodes of the computational graph to produce the report of the tensor’s values. TensorBoard web interface can be interacted in a way where once the computational graph is uploaded one can observe the model as well as monitor the operations.

4.3 Comparison with other deep learning libraries

In this section we compare TensorFlow with other machine learning frameworks which are discussed in section 3.1.

**Theano**: Among the three alternative libraries, Theano is the most similar as TensorFlow. Theano’s programming model is declarative instead of imperative based on the computational graph. Also it adopts symbolic differentiation. As it applies more advanced optimization algorithms on the graph whereas TensorFlow only executes common subgraph elimination, Theano has longer graph compile time. Moreover Theano’s visualization interface is poor than TensorBoard. Lastly, Theano is lack of the ability of distributed execution on the computational graph which is supported by TensorFlow.

**Torch**: The main difference between Torch and TensorFlow is that Torch has a C/CUDA backen and uses Lua as its frontend which is not a mainstream programming language compared to Python. This leads to a harder application in the industry than Python-based TensorFlow. In addition Torch’s programming model is different from TensorFlow in a way that Torch adopts imperative instead of declarative, which implies that Torch performs symbol-to-number differentiation to compute gradients for optimizing the model.

**Caffe**: Caffe is significantly different in many aspects compared with TensorFlow. It uses MATLAB and Python as frontend to construct model. The basic building unit of Caffe is entire network layers rather than operations in TensorFlow. Similarly as Torch, Caffe has no focus on th construction of the computational graph which results in computing gradients using symbol-to-number method. Moreover, Caffe is especially suitable for the convolutional neural networks and image recognition, whereas it is not robustness as TensorFlow in the area of neural networks. Lastly Caffe can not support distributed execution.
5 Experiments

In this section we conduct experiments for detecting credit card fraud transactions and present the results. The data set of credit card transactions was made by European cardholders occurred in two days in September 2013. It contains 284,807 transactions where 492 of them are fraud transactions, which accounts for 0.172% of all transactions.

The description of the credit card dataset is given on www.kaggle.com. The dataset consists of thirty features that are numerical values already transformed by principle component analysis (PCA) to reduce the dimension of feature space. For the sake of information security, the original features of consumers are not given. Features V1, V2, ... V28 are the principal components obtained with PCA. Two features 'Time' and 'Amount' are not transformed from PCA. Feature 'Time' represents the time elapsed between each transaction and the first transaction in the dataset. Feature 'Amount' is the amount of the transaction. Label 'Class' is the target label with value 1 representing the fraud case and 0 representing the normal case.

Standardization of the feature space is a common requirement for training the network effectively since the neural network is sensitive to the way how the input vectors are scaled [26]. In many dataset the features are different scaled and have different range of values which results in longer training time for the network to converge. Therefore it is helpful to standardize all the features in a way that each feature is transformed to standard normally distribution by removing the mean value of each feature, then scaling it by dividing non-constant features by their standard deviations.

In the data set we consider fraud transaction as positive class and normal transaction as negative class. There are four predictive outcomes which can be formulated in $2 \times 2$ confusion matrix as shown in Table 5.1. The column represents predicted label and row represents the true label. $TN$ is the number of normal transactions correctly classified (True Negative), $FP$ is the number of normal transactions incorrectly classified as fraud (False Positive), $FN$ is the number of fraud transactions incorrectly classified as normal (False Negative) and $TP$ is the number of fraud transactions correctly classified (True Positive).

The performance of the neural network learning algorithm is normally evaluated using predictive accuracy [27], which is defined as $\text{Accuracy} = \frac{TP + TN}{TP + FP + TN + FN}$. Unfortunately such a simple predictive accuracy is not an appropriate measure since the distribution of normal and fraud transactions in the data set is extremely imbalanced. A simple default strategy of classifying a transaction as normal transaction yields a very high accuracy. Despite the high accuracy, the model is lack of the ability to detect any fraud transactions within all transactions. Our goal of training the network is to obtain a fairly high rate of detecting fraud transactions correctly which allows a lower error rate in the normal transactions to achieve this [27]. Therefore simple predictive accuracy
is obviously not an effective metric in such case. The Receiver Operating Characteristic (ROC) curve is a standard technique to evaluate the performance of model for binary classification.

<table>
<thead>
<tr>
<th></th>
<th>Predicted Negative</th>
<th>Predicted Positive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actual Negative</td>
<td>TN</td>
<td>FP</td>
</tr>
<tr>
<td>Actual Positive</td>
<td>FN</td>
<td>TP</td>
</tr>
</tbody>
</table>

*Table 5.1* Confusion matrix of classifying normal and fraud transactions

A ROC curve describes performances of the model, which is represented by the relationship between true positive rate (TPR) and false positive rate (FPR) over a range of decision thresholds [28]. On ROC space the X-axis is defined as \( \%FP = \frac{FP}{TN+FP} \) (sensitivity) and Y-axis is defined as \( \%TP = \frac{TP}{TP+FN} \) (1-specificity). The best prediction occurs at the point (0,1) in the ROC space, representing all positive classes are classified correctly and no negative classes are classified as positive classes. A random prediction produces a point along the diagonal line \( y = x \). The closer ROC curve is to the upper left corner, the higher accuracy the model obtains. Therefore ROC curves can easily present comparison of performances between different models. Area Under the ROC Curve (AUC) is an effective metric to summarize the performance of a ROC curve. Generally the ROC curve with larger AUC reveals a better performance model.

There exits other evaluation metrics derived based on the confusion matrix shown in Table 5.1. Here we choose to present one particular evaluation metric which is called recall, defined as \( \%TP = \frac{TP}{TP+FN} \). Recall is the percentage of true positive classes that are detected correctly by the model among total positives classes. It is used to compare different models when performance of detecting positive(minority) class is preferred [29].

Logistic regression is chosen as the benchmark model, which can be regarded as the simplest neural network if sigmoid activation function is used [30]. Then we construct neural networks with complex structure to investigate if we can obtain better performance than using logistic regression model. The experiments are implemented in Scikit-learn and TensorFlow respectively.

### 5.1 Logistic regression model

Logistic regression is a regression method for predicting a binary dependent variable which takes value 0 or 1. Considering an input vector with \( n \) independent variables \( x = \{x_1, x_2, ... x_n\} \), the conditional probability of dependent variable being class 1 is defined as

\[
P(y = 1|x) = \pi(x) = \frac{1}{1 + e^{-g(x)}}
\]

where \( \frac{1}{1 + e^{-g(x)}} \) is the logistic function or sigmoid function, \( g(x) = w_0 + w_1x_1 + \cdots + w_nx_n \). Assume there are \( m \) observed dependent variables \( y_1, y_2, ... y_m \), the
probability distribution of dependent variables is given by

\[
P(Y_i = y_i) = \begin{cases} 
\pi(x_i)^{y_i}(1 - \pi(x_i))^{1-y_i}, & y_i=0 \text{ or } 1 \\
0, & \text{otherwise.} 
\end{cases}
\]

The likelihood function is the product of these probabilities as shown below:

\[
L(w) = \prod_{i=1}^{m} \pi(x_i)^{y_i}(1 - \pi(x_i))^{1-y_i} \tag{36}
\]

The optimal parameters \(w_0, w_1, \ldots, w_n\) are estimated via maximizing the likelihood function which takes logarithm of \(L(w)\). The loss function of logistic regression to be minimized is simplified as following:

\[
C(w) = -\frac{1}{m} \sum_{i=1}^{m} (y_i \log[\pi(x_i)] + (1 - y_i) \log[1 - \pi(x_i)]) \tag{37}
\]

In logistic regression, the model complexity is already low which makes less overfitting for training\([30]\). The result implemented in Scikit-learn is shown below:

```
[[54925 1936]
 [ 4 971]]
Accuracy: 96.59%
Recall: 96.84%
```

### 5.2 Neural network using Scikit-learn

In this section deep neural networks are constructed and implemented in Scikit-learn, which is an open source machine learning commercial library. The data set is split into a training set and a test set. The learning of network is performed on the training set and the performance of the network is evaluated on the test set. Due to the imbalanced data, we train the network without re-sampling and with re-sampling on the training set to investigate the impact of re-sampling on performance of the network.

The designing of neural network topology is the critical factor of affecting the accuracy of classification system. Adding hidden nodes can increase the accuracy of the network, however, the excessive hidden nodes will cause overfitting problem, which has negative impact on the generalization leading to deviations in prediction; therefore improving the accuracy and generalization requires suitable numbers of hidden nodes \([31]\). There has not been formal theory in determination of numbers of hidden nodes. The recommendation is based on previous experience and repeated experiments.

In Larochelle et al. 2009 \([32]\), it states that the network with the best performance is the one with same number of nodes in each hidden layer. In the experiments we test with different number of nodes in the hidden layers and also find some structures give worse results or as good as the structure with equal number of nodes in the hidden layers. Therefore we adopt the same number of nodes, such as 5, 15 and 20, in the hidden layers and do experiments starting from small network with one hidden layer then expand the network layer by
layer until 5 hidden layers. We performed test and found the network having 4 hidden layers with 20 nodes in each hidden layer produced better result. This was trained with learning rate 0.001 for 400 iterations and L2 regularization parameter 0.1.

The result for prediction on the test set without oversampling is shown below:

\[
\begin{bmatrix}
15651 & 10 \\
23 & 781
\end{bmatrix}
\]

Accuracy: 99.94%
Recall: 77.23%

To balance the data, oversampling was performed on the training set to increase the minority class to make it equal to the majority class. In [33], one approach is Synthetic Minority Over-sampling Technique (SMOTE), which oversample the minority class by creating “synthetic” examples. For each fraud sample \(x\), synthetic examples are introduced along the lines joining to the \(k\) nearest fraud samples neighbors. According to the unbalanced ratio of the data set we decide the amount of required over-sampling \(N\). Therefore only \(N\) neighbors \(\tilde{x}\) from the \(k\) nearest neighbors are randomly chosen and samples are generated in the direction of each. Each \(\tilde{x}\) is respectively combined with original sample to generate new sample as following formula:

\[
\begin{align*}
x_{new} &= x + \text{rand}(0, 1) \times (\tilde{x} - x) \\
\end{align*}
\]

This creates a new sample along the line segment between two features \(x\) and \(\tilde{x}\). The prediction result with oversampling is given below:

\[
\begin{bmatrix}
156748 & 113 \\
14 & 811
\end{bmatrix}
\]

Accuracy: 99.78%
Recall: 86.14%

We also found another neural network with 5 hidden layers and 20 nodes in each layer obtain good results, given learning rate 0.1 and maximum iteration 400. The result without oversampling is shown below:

\[
\begin{bmatrix}
156851 & 0 \\
191 & 0
\end{bmatrix}
\]

Accuracy: 99.82%
Recall: 0.0%

The result with oversampling is shown below:

\[
\begin{bmatrix}
154783 & 2076 \\
6 & 951
\end{bmatrix}
\]

Accuracy: 96.34%
Recall: 94.06%

As seen from above two networks’ results, we found that the results using oversampling on the training set are better than the results without using oversampling method. Following this, it is concluded that oversampling using SMOTE on the imbalanced training set can improve the overall performance of the network to detect the fraud transactions correctly. Comparing two neural networks’ results using oversampling method, the second network obtains higher recall on the test set, whereas the number of normal transactions been classified as fraud transactions are larger than the result of the first network.
Since the cost of processing fraud transactions are considerable, the situation of classifying normal transactions as fraud transactions will lead to unnecessary cost. Therefore we choose the first neural network with four hidden layers to conduct the following experiments.

After deciding the network topology we then inspect how learning rate affects the performance of the network. We found that $L_2=0.1$ gave the best results with respect to recall; therefore we kept this value constant throughout the experiments. We tried different learning rates with maximum iterations 80. The results are given as below:

![Figure 4.1 The impact of learning rates on results](image)

In Scikit-learn when the training loss did not improve more than by at least tolerance for two consecutive epochs, the network is considered to be convergent and training stops. This is the reason that some curves are shorter in Figure 4.1. As seen from the graph, larger learning rates increase the speed of network convergence, whereas cause instability of the network. Small learning rates lead to slow convergence which make the convergent path more smoothly. Furthermore the loss is smaller when using small learning rates.

### 5.3 TensorFlow implementation

The TensorFlow code was inspired by the code developer from [www.kaggle.com][34]. I used part of the original code and modified it.

The dataset is split into training set, validation set and test set. The training set is balanced by duplicating fraud transactions to make the number of two classes equality. The structure of the network is the same as in Scikit-learn implementation which has 4 hidden layers with 20 nodes in each layer. We choose learning parameters by testing against validation accuracy. We found that the network converges at epoch=200 and learning rate=0.005 with respect to the validation accuracy. Figure 4.2 shows that the result of training accuracy, validation accuracy, training cost and validation cost. The average training accuracy= 0.99765, average validation accuracy= 0.99695, average training cost=...
18226.50195, average validation cost= 850.31555. Then we predicted on the test
set and obtained recall of 86.89%.\textsuperscript{2} Compared with Scikit-learn implementation,
both implementations obtained recall in the same range.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{accuracy_cost}
\caption{Results for training accuracy (blue), validation accuracy (green), training cost and validation cost obtained from the neural network given epoch=1000, learning rate=0.005}
\end{figure}

Next we try different learning rates to investigate its impact on the experiment results given epoch=500 throughout the experiments. The results are shown as Figure 4.3 and Table 4.1.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{learning_rates}
\caption{Validation accuracy for different learning rates}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{learning_rates_table}
\caption{Table 4.1 Average validation accuracy for different learning rates}
\end{figure}

As seen from Figure 4.3, larger learning rate can speed up the convergence; however it is easily to overshoot the local minimum of loss function which leads to oscillations. This instability of the network will have negative impact on the accuracy performance. In practice when approaching the optimal solution we

\textsuperscript{2}This is the result of one calibration. In fact each time the optimization routine can find different local minimums to obtain different weights which leads to changes in results.
can use small learning rates since it makes the convergent path more smoothly to ensure convergence to local minimum.

Next we try different weight initializations to investigate how it affects the results. The weights should be random values which are small enough closed to zero but not identically zero. In previous experiments the weights were initialized using `tf.truncated_normal` which returns values from a normal distribution with specified mean and standard deviation except those values that are larger than two standard deviation from the mean are dropped to regenerate. The purpose for using truncated normal is to avoid saturation of neuron adopting sigmoid active function.

Another similar option used in the experiment is `tf.random_normal` which generates random values from a normal distribution. The weight vector of each neuron is initialized to a random vector, and these random vectors obey a multivariate Gaussian distribution, so that all neurons in the input space are random small. With this formula, we chose standard deviation equal to 0.01 and 1 for experiments.

Based on the article Xavier Glorot, Yoshua Bengio(2010), one of popular methods for initializing weights is called normalized initialization:

\[
W \sim U\left(-\frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}, \frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}\right)
\]

(39)

where \(n_j\) is the number of neurons in \(j\)th layer. This method ensures the requirement of largest gradient and activation variance.

The results of validation accuracy with different weight initialization is shown as below:

![Figure 4.4 Results with different weight initialization given learning rate=0.005.](image)

As seen from Figure 4.4, good initialization methods are usually only to speed up the learning or convergent speed.

---

6 Conclusion

In this thesis we discussed the theory of neural network’s structure and backpropagation method for learning optimal parameters. Then we introduced the machine learning software TensorFlow and its working mechanism. Next we applied neural network to conduct experiments on credit card fraud data set and compare different models based on the evaluation metric recall. We choose logistic regression as the benchmark model, which yields 96.04% recall demonstrating a better performance result on the test set than neural network. Then we tested different neural networks with different number of hidden layers. Both implementations in Scikit-learn and TensorFlow obtain consistent results. The experiments demonstrated that increasing hidden layers did not improve the classification performance significantly. The network having one hidden layer with several neurons can obtain similar predictive result as the network with several hidden layers. Furthermore the results show that the use of re-sampling imbalanced training set can increase the performance of the network on the test set.
A Scikit-learn code

```python
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import confusion_matrix
from imblearn.over_sampling import SMOTE
from sklearn import preprocessing

data = pd.read_csv('creditcard.csv')

columns = data.columns
# The labels are in the last column ('Class'). Simply remove it to obtain features columns
features_columns = columns.delete(len(columns) - 1)
features = data[features_columns]
labels = data['Class']

X_scaled = preprocessing.scale(features)
features_train, features_test, labels_train, labels_test = train_test_split(X_scaled, labels, test_size=0.2, random_state=0)

oversampler = SMOTE(random_state=0)
os_features, os_labels = oversampler.fit_sample(features_train, labels_train)

clf = MLPClassifier(solver='adam', alpha=1,
                    hidden_layer_sizes=(20, 20, 20, 20),
                    learning_rate_init=0.05,
                    verbose=10, max_iter=400, random_state=1)

# clf.fit(features_train, labels_train)
clf.fit(os_features, os_labels)

print("Training set score: %f" % clf.score(features_train, labels_train))
print("Training set score: %f" % clf.score(os_features, os_labels))

y_pred = clf.predict(features_test)
c = confusion_matrix(labels_test, y_pred)

print(c)
```
print ('Accuracy: ' + str(np.round(100*float((c[0][0]+c[1][1])+c[0][0]+c[1][1])+(c[0][0]+c[1][1]+c[1][0]+c[0][1])),2))+'%')
print ('Recall: ' + str(np.round(100*float((c[1][1])+c[1][1])/float((c[1][0]+c[1][1])),2))+'%')

B TensorFlow code

import pandas as pd
import numpy as np
import tensorflow as tf
from sklearn.cross_validation import train_test_split
import matplotlib.pyplot as plt
from sklearn.utils import shuffle
import seaborn as sns
import matplotlib.gridspec as gridspec
import math
from sklearn.metrics import confusion_matrix
from sklearn.metrics import recall_score
data= pd.read_csv("./creditcard.csv")
# Create a label to mark cases for normal (non-fraudulent) transactions.
data.loc[data.Class == 0, 'Normal'] = 1
data.loc[data.Class == 1, 'Normal'] = 0
# Rename 'Class' to 'Fraud'.
data= data.rename(columns={'Class': 'Fraud'})
# Create dataframes of only Fraud and Normal transactions.
Fraud = data[data.Fraud == 1]
Normal = data[data.Normal == 1]
# Set X_train equal to 80% of the fraudulent transactions.
X_train = Fraud.sample(frac=0.8)
count_Frauds = len(X_train)
# Add 80% of the normal transactions to X_train.
X_train = pd.concat([X_train, Normal.sample(frac=0.8)], axis=0)
# X_test contains all the transactions not in X_train.
X_test = data.loc[~data.index.isin(X_train.index)]
# Shuffle the dataframes so that the network is trained in a random order.
X_train = shuffle(X_train)
X_test = shuffle(X_test)
# Add our target values to y_train and y_test.
y_train = X_train.Fraud
y_train = pd.concat([y_train, X_train.Normal], axis=1)
\[ y \text{ test} = X \text{ test.Fraud} \]
\[ y \text{ test} = \text{pd.concat}([y \text{ test}, X \text{ test.Nomal}], \text{axis}=1) \]

# Delete target values from X_train and X_test.
X_train = X_train.drop(['Fraud', 'Normal'], axis=1)
X_test = X_test.drop(['Fraud', 'Normal'], axis=1)

# The dataset needs to be balanced. Ratio is defined by dividing the number of transactions by fraud transactions.
# Thus the number of fraud transactions multiplied with the ratio will equal the number of normal transactions.
ratio = len(X_train)/count_Frauds
y_train.Fraud *= ratio

# Names of all of the features in X_train.
features = X_train.columns.values

# Transform each feature in feature space so that it has a mean of 0 and standard deviation of 1;
# this helps with training the neural network.
for feature in features:
    mean, std = data[feature].mean(), data[feature].std()
    X_train.loc[:, feature] = (X_train[feature] - mean) / std
    X_test.loc[:, feature] = (X_test[feature] - mean) / std

# Train the Neural Network. Split the testing data into validation and testing sets.
split = int(len(y_test) / 2)
inputX = X_train.as_matrix()
inputY = y_train.as_matrix()
inputX_valid = X_test.as_matrix()[:split]
inputY_valid = y_test.as_matrix()[:split]
inputX_test = X_test.as_matrix()[split:]
inputY_test = y_test.as_matrix()[split:]

# Number of input nodes.
n_input=X_train.shape[1]
# Number of nodes in each hidden layer
n_hidden_1 = 20
n_hidden_2 = 20
n_hidden_3 = 20
n_hidden_4 = 20

# input
x = tf.placeholder(tf.float32, [None, n_input])
# layer 1
W1 = tf.Variable(tf.truncated_normal([n_input, n_hidden_1], stddev=0.01))
b1 = tf.Variable(tf.zeros([n_hidden_1]))
y1 = tf.nn.sigmoid(tf.matmul(x, W1) + b1)

# layer 2
W2 = tf.Variable(tf.truncated_normal([n_hidden_1, n_hidden_2], stddev=0.01))
b2 = tf.Variable(tf.zeros([n_hidden_2]))
y2 = tf.nn.sigmoid(tf.matmul(y1, W2) + b2)

# layer 3
W3 = tf.Variable(tf.truncated_normal([n_hidden_2, n_hidden_3], stddev=0.01))
b3 = tf.Variable(tf.zeros([n_hidden_3]))
y3 = tf.nn.sigmoid(tf.matmul(y2, W3) + b3)

# layer 4
W4 = tf.Variable(tf.truncated_normal([n_hidden_3, n_hidden_4], stddev=0.01))
b4 = tf.Variable(tf.zeros([n_hidden_4]))
y4 = tf.nn.sigmoid(tf.matmul(y3, W4) + b4)

# output layer
W5 = tf.Variable(tf.truncated_normal([n_hidden_4, 2], stddev=0.01))
b5 = tf.Variable(tf.zeros([2]))
y5 = tf.nn.softmax(tf.matmul(y4, W5) + b5)

# output and target values
y = y5
target = tf.placeholder(tf.float32, [None, 2])

# Parameters of the model
training_epochs = 1000
display_step = 10
n_samples = y.train.shape[0]
batch_size = 2048
learning_rate = 0.005
total_batch=int(n_samples / batch_size)

# Compute the cross-entropy cost function
cost = tf.reduce_mean(-tf.reduce_sum(target * tf.log(y)))

# Update the weights of the model via AdamOptimizer
optimizer = tf.train.AdamOptimizer(learning_rate).
minimize(cost)

# Check if the prediction from the output layer matches
the target label.
correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(target, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

accuracy_summary = []  # Record accuracy values for plot
cost_summary = []  # Record cost values for plot
valid_accuracy_summary = []
valid_cost_summary = []

# Initialize the variables
init=tf.global_variables_initializer()
#Execute the operations in the graph
with tf.Session() as sess:
    sess.run(init)
#Training the network
for epoch in range(training_epochs):
    for i in range(total_batch):
        sess.run([optimizer], feed_dict={x: inputX[i * batch_size: (1 + i) * batch_size],
                                         target: inputY[i * batch_size: (1 + i) * batch_size]})
    # Display logs after every 10 epochs
    if epoch % display_step == 0:
        train_accuracy, newCost = sess.run([accuracy, cost], feed_dict={x: inputX, target: inputY})
        valid_accuracy, valid_newCost = sess.run([accuracy, cost], feed_dict={x: inputX_valid, target: inputY_valid})

        print(epoch,
              "{:5f}".format(train_accuracy),
              "{:5f}".format(newCost),
              "{:5f}".format(valid_accuracy),
              "{:5f}".format(valid_newCost))

        # Record the results of the model
        accuracy_summary.append(train_accuracy)
        cost_summary.append(newCost)
        valid_accuracy_summary.append(valid_accuracy)
        valid_cost_summary.append(valid_newCost)

        #Obtain accuracy and recall on the test set
        y_p=tf.argmax(y,1)
        test_accuracy, y_pred=sess.run([accuracy, y_p],
                                       feed_dict={x: inputX_test, target:inputY_test})
        y_true = np.argmax(inputY_test,1)
c = confusion_matrix(y_true, y_pred)

print ('Test accuracy: ' + str(np.round(100*float((c[0][0] + c[1][1]) / float((c[0][0] + c[1][1] + c[1][0] + c[0][1])),2)) + '%')
print ('Recall: ' + str(np.round(100*float((c[0][0]) / float((c[0][0] + c[0][1])),2)) + '%')

f, (ax1, ax2) = plt.subplots(2, 1, sharex=True, figsize=(10,4))
ax1.plot(accuracy_summary) # blue
ax1.plot(valid_accuracy_summary) # green
ax1.set_title('Accuracy')
ax2.plot(cost_summary)
ax2.plot(valid_cost_summary)
ax2.set_title('Cost')
plt.xlabel('Epochs (x10)')
plt.show()
References


erogeneous distributed systems,” 2015.


[29] S. L. Phung, A. Bouzerdoum, and G. H. Nguyen, “Learning pattern class-

ification tasks with imbalanced data sets,” 2009.


