Neural Networks

Contents

1 Introduction

This section looks at neural networks. The main contribution of neural networks is processing in parallel, taking an analogy from how the brain works. A neural network is essentially a mathematical model of the brain and can take the form of a software or hardware implementation. As can be seen from section 4 a number of neural networks have been developed over time. This section will look at three of the commonly used neural networks and learning algorithms, namely, the perceptron learning algorithm, the backpropagation algorithm and the Hopfield neural network. The section ends by examining deep learning and convolution neural networks.

2 The Computational Neuron

Neural networks takes an analogy from the structure of the human brain which is composed of neurons. Neural networks are essentially a network of neurons. A neural network is comprised of two or more layers, with each layer comprised of a number of neurons. Figure 1 illustrates a biological neuron.

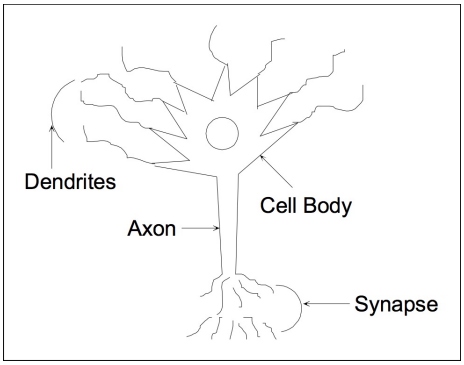

Figure 1: A biological neuron

A neuron receives input via its dendrites. The cell body of the neuron than processes the inputs and may produce an output signal or not. If it produces an output signal the neuron is said to *fire*. The output of the neuron is transmitted via the axon and transferred to the next neuron via the synapse. Our brains are made up of networks of neurons transmitting signals to perform different functions. The computational equivalent of a neuron is depicted in Figure 2. The vector $p = p_1, ..., p_m$ represents input to the neuron. The vector $w = w_1, ..., w_m$ represent the weights. In addition to the weights some neural network architectures include a *bias*, which is indicated by b in the figure. The bias always has an input of 1. The processing in the computational neuron involves taking a sum of the weighted inputs and the bias. Hence $n = \sum_{i=1}^{m} = w_i p_i + b$.

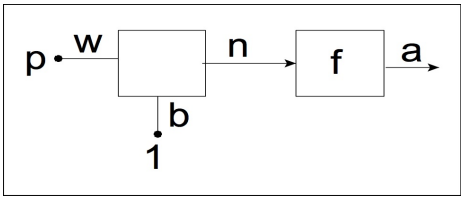

Figure 2: A computational neuron [1]

The weighted sum n is input to an activation function. The activation or transfer function is used to determine the output of the neuron. The simplest activation will produce a 1 (indicating that the neuron will fire) or 0 (indicating that the neuron will not fire). The bias has on impact on the activation function. There are a number of activation functions, e.g. hard limit, linear, sigmoid [2]. Different learning algorithms can be used to determine the weights and biases for a neural network. This processing of determining the weights and biases is referred to as *training* the neural network. A *training set*, providing the inputs to the neural network, is used to train the neural network. A learning algorithm, e.g. perceptron learning algorithm, backpropagation algorithm, is used to determine the weights and biases. Different learning algorithms use different activation functions.

Each neuron forms part of of neural network. The following section looks at different neural network architectures.

3 Neural Network Architectures

This section looks at different neural network architectures. Figure 3 depicts a single neuron in a single layer. The neuron has multiple inputs.

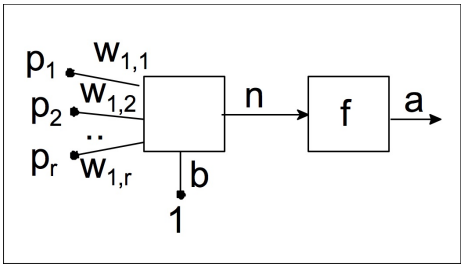

Figure 3: A single neuron single layer neural network

Figure 4 [2] depicts a single layer neural network with s neurons in the network.

Figure 5 depicts a two layer neural network. The first layer consists of two neurons and the second layer one neuron.

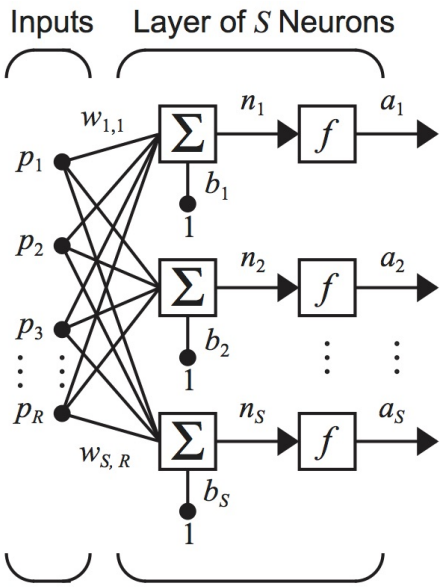

Figure 4: s neurons single layer neural network [2]

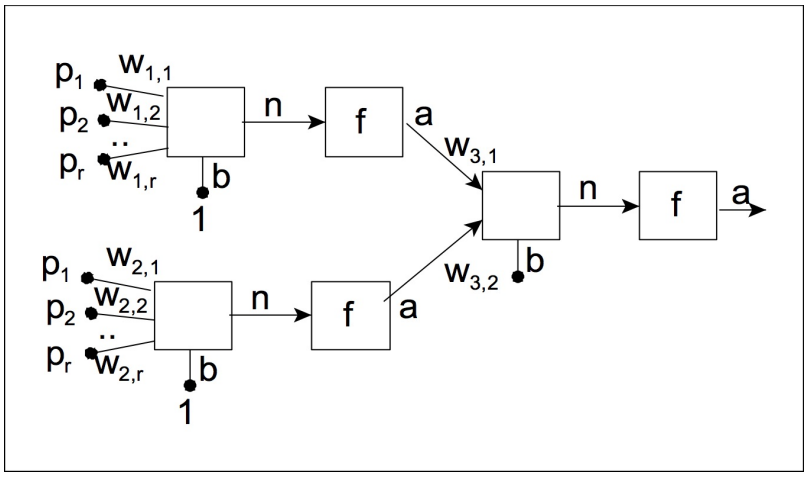

Figure 5: Two layer neural network

Finally, Figure 6 [2] displays a multilayer neural network comprised of 3 layers, with each layer contain S neurons.

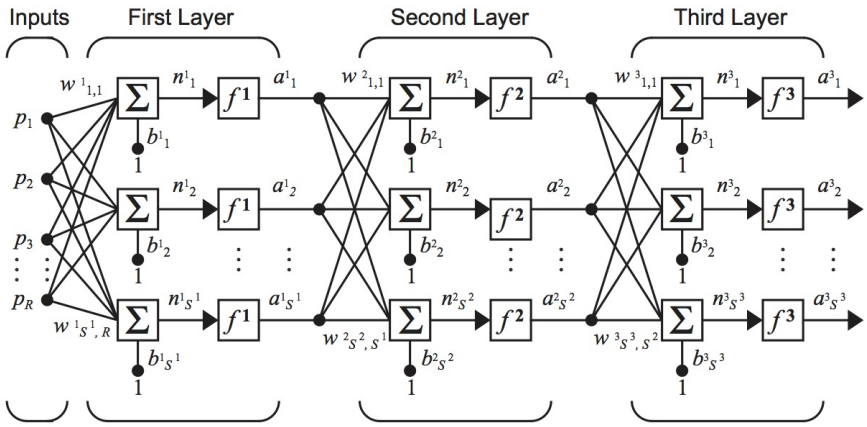

Figure 6: Two layer neural network [2]

4 History of Neural Networks

Different neural network architectures and learning algorithms have been developed over the years since the inception of neural networks in the 1940's with highlights marking various eras [1, 2]. This is depicted below:

The Beginning of Neural Networks (1940's): McCulloch Pitts Neuron, Hebbian Learning
The First Golden Age of Neural Networks (1950's and 1960's): Perceptrons, Adaline
The Quiet Years-1970's: Kohenen, Anderson, Grossberg, Carpenter
Renewed Enthusiasm-1980's: Backpropagation, Hopfield nets, Neocognitron,Boltzman machine, hardware Implementation
Recurrent Neural Networks and Gradient-Based Learning (1990's): Long Short-Term Memory (LSTM), gradient-based learning
Currently: Deep learning and convolutional neural networks

5 McCulloch-Pitts Neuron

The McCulloch-Pitts neuron was the first neuron introduced at the inception of neural networks [1]. The architecture is a very simple architecture compared to the neural net-

works that we use in this day and age. This neuron is a binary neuron which has an output of 1, i.e. the neuron fires, or an output of 0 indicating that the neuron does not fire. A threshold θ is set for each neuron. This is a parameter for the neuron and the best value is problem dependent and is a parameter value that is tuned for the neural network. The activation function for the McCulloch-Pitts neuron is depicted in equation 1:

$$\begin{aligned} f(n) = 1 \; if \; n >= \theta \\ 0 \; if \; n < \theta \end{aligned} \tag{1}$$

An example of a McCulloch-Pitts neuron architecture is illustrated in Figure 7. The neuron takes three inputs, x_1, x_2 and x_3. The weights determined for the neuron are 2, 2, and 1.

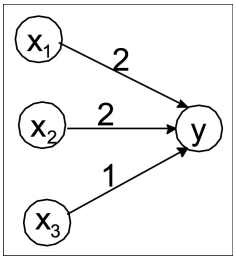

Figure 7: McCulloch-Pitts neuron example [1]

Suppose that you are required to design a McColloch-Pitts neuron to emulate the logical *or* function. In this case the neuron will have two inputs, x_1 and x_2. The values for both the inputs and the corresponding outputs are illustrated in Table 1.

Table 1: OR logical function

x_1	x_2	Output
0	0	0
0	1	1
1	0	1
1	1	1

Training the neural network will require determining the weights, w_1 and w_2, and the θ value so that given the input values x_1 and x_2 in Table 1 the neuron produces the corresponding output. So for inputs 0 and 0 the output must be 0 and for inputs 1 and 1 the output must be 1. For the McCulloch-Pitts neuron, due to its simplicity, the weights are determined manually instead of using a learning algorithm. If a value of 2 is used for θ

and 2 and 2 are used for the weights this will give the required output. This is illustrated in Table 2.

Table 2: Training for the OR logical function

x_1	x_2	n	$f(n)$
0	0	0	0
0	1	2	1
1	0	2	1
1	1	4	1

To solve real-world problems neural networks consists of a number of layers of a number of neurons and hence the weights to train the neural network cannot be determined manually, a learning algorithm is used to determine the weights.

6 Linear Separability

If you tried to find the weights and θ value to train a single McCulloch-Pitts neuron to emulate the XOR function you would spend a lot of time doing this and would not be able to come up with a solution. The reason for this is that the XOR function is not a linear separable function and hence more than one layer will be required in a neural network in order to emulate the function. The concept of linear separability is illustrated in Figure 8.

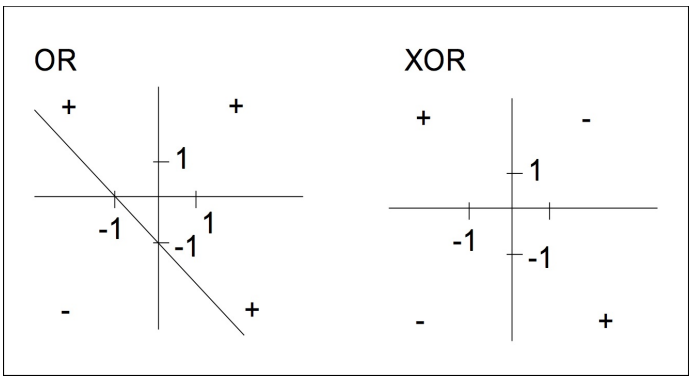

Figure 8: Linear separability [1]

If you look at the graph for the OR function you will see that the area where the neuron fires, i.e. outputs a value of 1, can clearly be divided by a line from the area of the graph in which the neuron does not fire, i.e. produces a value of 0. However, if you look at the graph

for the XOR function the area in which the neuron fires cannot be divide by a line from the area in which the neuron cannot fire. Hence, XOR is not a linearly separable problem. In instances where the problem is not linearly separable a multilayer neural network is needed to solve the problem.

7 The Single Layer Perceptron

Neural networks can be classified into two categories based on their pattern recognition function, namely, pattern classification and pattern association. The perceptron is a *feed-forward* neural network that performs pattern classification [3]. The perceptron can be single layer or multilayer [2]. This section examines the single layer perceptron learning algorithm. Training a perceptron neural network involves determining the weights and bias for each layer. A learning algorithm is used to determine the weights and bias. The learning algorithm performs a number iterations, called *epochs*, over which the weights and bias are determined. The algorithm must *converge* to a set of weights and bias. The algorithm has converged when the weights and bias can produce the target output for all instances in the training set.

The perceptron performs classification. The inputs represent the attributes for the classification problem and the output the class. If the classification is binary, i.e. the input vector represents an entity either belonging to the class or not, the target output is 0 (or -1) or 1, i.e. the output vector has length one. In the case of multiclass classification where an entity can belong to one of n classes, where $n > 2$, the output vector has length greater than one. Input and output vectors can be specified as binary or bipolar. In bipolar a -1 is used instead of a 1.

Binary classification example

Suppose that a warehouse has a conveyor belt for sorting fruit according to the shape, texture and weight of the fruit [2]. The belt contains a sensor for shape, a sensor for texture and a sensor for weight. A fruit can be round or elliptical. A value of 1 represents the shape round and -1 the shape elliptical. The texture of the fruit can be smooth or rough. A value of 1 indicates a smooth texture and -1 a rough texture. The sensor outputs a 1 if the weight of the fruit is greater than 0.5kg and -1 if the fruit is less than or equal to 0.5kg. Suppose that the two fruit that have to be separated into different bins from the conveyor belt are apples and oranges. The input vector corresponding to an orange is $p = [1\ -1\ -1]$ with the 1 representing round, the first -1 representing a rough texture and the second -1 representing a weight less than 0.5 kg. Similarly, the input vector corresponding to an apple is $p = [1\ \ 1 - 1]$ with the first 1 representing round, the second smooth and the -1 representing a weight less than 0.5 kg. The class of apples can be represented by 1 and oranges as -1. This the output will have a dimension of one. Table 3 lists the input vectors

and corresponding output vector.

Table 3: Binary classification example

p			t
p_1	p_2	p_3	t
1	-1	-1	-1
1	1	-1	1

Figure 9 depicts the perceptron architecture with three inputs and one output.

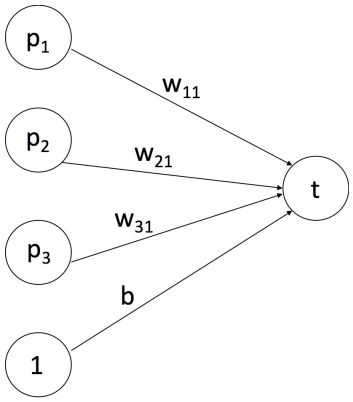

Figure 9: Perceptron architecture for binary classification

The corresponding weight matrix and bias vector that must be calculated are:

$$\mathbf{W}=\begin{pmatrix} \mathrm{w}_{11} \\ \mathrm{w}_{21} \\ \mathrm{w}_{11} \end{pmatrix} \mathbf{b} = [b]$$

Multiclass classification example

Suppose that in addition to apples and oranges, the conveyor belt also has to separate grapefruit from these two fruit. The input vector corresponding to grapefruit will be $p = [1 \quad -1 \; 1]$ with the first 1 representing round, the -1 representing a rough texture and the second 1 representing a weight greater than 0.5kg. Table 4 displays the corresponding input and output vectors.

Figure 10 depicts the perceptron architecture with three inputs and two outputs.

Table 4: Multiclass classification example

p			t	
p_1	p_2	p_3	$t1$	$t2$
1	-1	-1	-1	1
1	1	-1	1	-1
1	-1	1	1	1

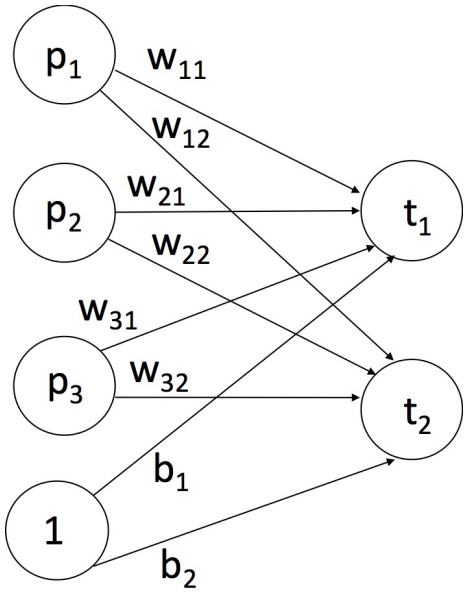

Figure 10: Perceptron architecture for multiclass classification

The corresponding weight matrix and bias vector that must be calculated are:

$$\mathbf{W}=\begin{pmatrix} \mathrm{w}_{11} & \mathrm{w}_{12} \\ \mathrm{w}_{21} & \mathrm{w}_{22} \\ \mathrm{w}_{31} & \mathrm{w}_{32} \end{pmatrix} \mathbf{b} = [\mathrm{b}_1 \quad b_2]$$

Before you apply the perceptron learning algorithm it is a good idea to draw a diagram of the neural network architecture to determine the weights and the biases. The perceptron learning algorithm uses the following activation function in the case of binary training data:

$$\begin{aligned} f(n) = 1 \; if \; n >= 0 \\ 0 \; if \; n < 0 \end{aligned} \tag{2}$$

In the case of bipolar data the following activation function is used:

$$\begin{aligned} f(n) = 1 \; if \; n >= 0 \\ -1 \; if \; n < 0 \end{aligned} \tag{3}$$

where $n = \sum_{i=1}^{m} \; w_{ij}p_i + b$, where m is the number of inputs. Note that when applying the perceptron learning algorithm the problem must be a binary classification problem, hence the number of outputs j will always be 1. Hence, in the algorithm below we will not include the j. Multiclass classification is generally not separable and hence a multilayer neural network will be used. The perceptron learning algorithm is applied in instances where the classification is linearly separable and only a single layer is required. The perceptron learning algorithm is depicted in Algorithm 1:

Algorithm 1 Perceptron Learning Algorithm

```
1: Set the weights and bias to zero or small random values.
2: while algorithm has not converged do
3:     for i ← 1, noOfTrainingInstances  do
4:         Calculate f(n)
5:         if f(n) != t then
6:             Update the weights using w_i = w_i + (t − f(n)) ∗ p_i
7:             Update the bias b = b + (t − f(n))
8:         end if
9:     end for
10: end while
```

The algorithm begins by setting the weights and bias to zero or random small values. The algorithm performs a pass through each of the instances in the training set to determine whether the current weights and bias result in the target output t in the instance.

This pass through the training set is called an *epoch*. If there is a difference in the activation of the perceptron, $f(n)$ and t the weights and and bias are updated using the learning rules in line 6 and 7 of the algorithm respectively. Alternative learning rules for the perceptron algorithm can include a learning rate $\alpha[1]$:

$w_i = w_i + \alpha * (t - f(n)) * p_i$

$b = b + \alpha(t - f(n))$

Consider the binary classification problem in Table 3. Suppose that we initialize the weights and bias to be zero:

Epoch 1

First training instance: p = [1 -1 -1], t =-1

$$\text{n= [1 -1 -1] x } \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} + 0 = 0$$

f(n) = 1

Change in weights and bias:

$w_1 = 0 + (-1 - 1) * 1 = -2$

$w_2 = 0 + (-1 - 1) * -1 = 2$

$w_3 = 0 + (-1 - 1) * -1 = 2$

$b = 0 + (-1 - 1) = -2$

Second training instance: p = [1 1 -1], t =1

$$\text{n= [1 1 -1] x } \begin{pmatrix} -2 \\ 2 \\ 2 \end{pmatrix} + (-2) = -4$$

f(n) = -1

Change in weights and bias:

$w_1 = -2 + (1 - (-1)) * 1 = 0$

$w_2 = 2 + (1 - (-1)) * 1 = 4$

$w_3 = 2 + (1 - (-1)) * -1 = 0$

$b = -2 + (1 - (-1)) = 0$

Epoch 2

First training instance: p = [1 -1 -1], t =-1

$$\text{n= [1 -1 -1] x } \begin{pmatrix} 0 \\ 4 \\ 0 \end{pmatrix} + 0 = -4$$

f(n) = -1

t is equal to $f(n)$ hence there is no change to weights and bias.
Second training instance: p = [1 1 -1], t =1

$$\text{n= [1 1 -1] x } \begin{pmatrix} 0 \\ 4 \\ 0 \end{pmatrix} + 0 = 4$$

f(n) = 1

t is equal to $f(n)$ hence there is no change to weights and bias. As there has been no change in weights and bias for the entire epoch, the algorithm has converged and will terminate.

The single layer perceptron is limited and a mulitlayer perceptron may be needed to solve real world problems. The learning algorithm that is generally used to train a mulitlayer perceptron neural network is the *backpropagation* learning algorithm. This is presented in the next section.

8 Backpropagation

The backpropagation learning algorithm was the first algorithm introduced to provide learning for multilayer neural networks. The algorithm is also referred to as the *generalized delta rule* [1]. As indicated by the name of the algorithm errors are propagated back after feedforward learning and weights and biases are updated accordingly. Thus the algorithm is comprised of three main processes:

- Training in a feedforward manner from the input layer to the output layer.

- The error calculated at the output layer is then propagated back through the network.
- The weights and biases are updated based on the propagated error.

In this section we will examine one version of the backpropagation learning algorithm. The backpropagation algorithm is a gradient descent algorithm. There are different versions of the algorithm based on aspects such as procedures used to update the weights. The architecture that the backpropagation algorithm will be applied to consists of single input layer, one or more hidden layers and a single output layer. The number of nodes in the input and output layers are equivalent to the number of inputs and outputs respectively. The number of nodes in the hidden layers are problem dependent and a design decision. Various activation functions $f(n)$ have been used by different implementations of the backpropagation algorithm. It is important that the activation function used is differentiable as the algorithm uses derivatives in calculating the error that is propagated back. Commonly used activation functions include the binary sigmoid function [1]:

$$f(n) = \frac{1}{1 + \exp(-n)} \tag{4}$$

with the derivative:

$$f'(n) = f(n)(1 - f(n)) \tag{5}$$

and the bipolar sigmoid function [1]:

$$f(n) = \frac{2}{1 + \exp(-n)} - 1 \tag{6}$$

with the derivative:

$$f'(n) = \frac{1}{2}(1 + f(n))(1 - f(n)) \tag{7}$$

An example of a multilayer architecture which the backpropagation algorithm can be applied to is illustrated in Figure 11. The neural network contains one input layer with three inputs, one hidden layer with two nodes and one output layer with two outputs. The input and hidden layers have a bias with an input of 1. The biases are indicated with an index of 0, e.g. v_{01} which is the bias to the hidden node h_1 from the input layer.

In this section we will look at the backpropagation algorithm with one hidden layer presented in [1]. This algorithm can easily be extended to include additional hidden layers. The algorithm will be divided into three processes, namely, feedforward learning, backpropagation of the error and update of the weights and biases. The overall algorithm is depicted in 2 and each of these processes are illustrated in Algorithm 3, 4 and 5. The algorithm continues until a stopping condition is met. Examples of stopping conditions commonly used include when there is no change in the error or an increase in the of the

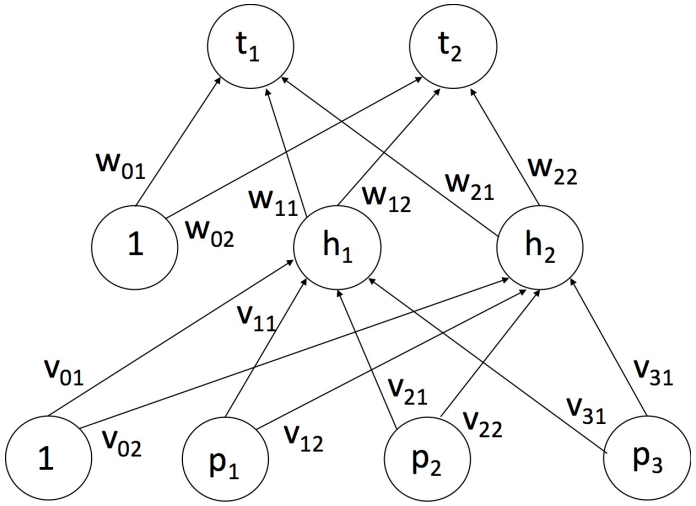

Figure 11: Example multilayer architecture

weights and bias [1]. The weights are initialized to small random values. The activation functions used are quite sensitive to the weights and bias and hence these values need to be chosen carefully. A common range used if -0.5 to 0.5.

The weighted sum of the inputs and bias is represented by n_1 for the hidden layer and n_2 for the output layer. The function n_1 is calculated for each node j in the hidden layer and is defined as:

$$n_{1j} = v_{0j} + \sum_{l=1}^{n} v_{lj} * p_l \tag{8}$$

where j is the number of nodes in the hidden layer, n is the number of inputs.
Similarly, n_2 is calculated for each node m in the output layer and is defined as:

$$n_{2m} = w_{0m} + \sum_{i=1}^{j} w_{im} * f(n_{1i}) \tag{9}$$

where m is the number of nodes in the output layer, j is the number of nodes in the hidden layer.

Equations used for the backpropagation of the error include:

Error information term for each node in the output layer:

$$\delta_k = (t_k - f(n_{2k}))f'(n_{2k}) \tag{10}$$

where k=1..m

The error information term is the difference between the target output specified in the training instance t and the activation of the output node $f(n_2)$ and the derivative of the activation.

Weight correction term for each node in the output layer:

$$\Delta w_{ik} = \alpha \delta_k f(n_{1i}) \tag{11}$$

where k = 1..m and i = 1..j

The weight correction term is the product of the learning rate α, the error information term of the output node and the activation of the hidden node.

Bias correction term for each node in the output layer:

$$\Delta w_{0k} = \alpha \delta_k \tag{12}$$

where k = 1..m

Similarly, the bias correction error is the product of the learning rate α and error information term of the output node.

Sum of the delta inputs from the output layer for each node in the hidden layer:

$$\delta n_i = \sum_{k=1}^{m} \delta_k w_{ik} \tag{13}$$

where i = 1..j

Error information term for each node in the hidden layer:

$$\delta_i = \delta n_i f'(n_{1i}) \tag{14}$$

where i = 1..j

Weight correction term for each node in the hidden layer:

$$\Delta v_{li} = \alpha \delta_i p_l \tag{15}$$

where l = 1..n and i = 1..j

The weight correction term is the product of the learning rate α, the error information term of the hidden node and input value

Bias correction term for each node in the hidden layer:

$$\Delta v_{0i} = \alpha \delta_i \tag{16}$$

where i = 1..j

Similarly, the bias correction error is the product of the learning rate α and error information term of the output node.

The equations for updating the weights and bias are:

Update weights and bias for the output layer

$$w_{ik}(new) = w_{ik}(old) + \Delta w_{ik} \tag{17}$$

where i = 0..j and k = 1..m

Update weights and bias for the hidden layer

$$v_{li}(new) = v_{li}(old) + \Delta v_{li} \tag{18}$$

where l = 0..n and i = 1..j

Algorithm 2 Backpropagation learning algorithm

1: Set the weights and bias to zero or small random values.
2: **while** The stopping condition is not met **do**
3: Perform feedforward learning
4: Backpropagation of error
5: Update weights
6: **end while**

Algorithm 3 Perform feedforward learning

1: Calculate n_1 for each node in the hidden layer
2: Calculate the activation $f(n1)$ for each node in the hidden layer
3: Calculate n_2 for each node in the output layer
4: Calculate the activation $f(n2)$ for each node in the output layer

Algorithm 4 Backpropagation of error

1: Calculate the error information term for each node in the output layer
2: Calculate the weight correction term for each node in the output layer
3: Calculate the bias correction term for each node in the output layer
4: Calculate the sum of delta inputs for each node in the hidden layer
5: Calculate the error information term for each node in the hidden layer
6: Calculate the weight error term for each node in the hidden layer
7: Calculate the bias error term for each node in the hidden layer

Once the neural network is trained, during the testing phase it is applied to unseen instances. The weighted sum of the inputs in each test case and the bias is calculated. The activation function used in the learning algorithm is applied to this weighted sum to determine the output.

9 Hopfield Neural Network

In the previous sections we examined neural networks for pattern classification. In this section we look at the use of neural networks for pattern association. Pattern association involves the neural network memorizing a pattern and still recognizing the pattern when it is input at a later stage even if it is not exactly the same as the pattern the neural network has learned. An example of this is specimen signatures which are stored in bank systems as your signature is never exactly the same as a previous time that you have signed. These neural networks are referred to as *associative memory* neural networks [1] and are single layer neural networks. The training data is comprised of input-output vector pairs. If the purpose of the neural network is to remember the pattern, namely, the input vector than the input and output vectors are the same and we refer to this as *autoassociative memory*. If the purpose of the neural is to memorize the association between two patterns, i.e. the input and output vectors, then the input and output vectors are different and we refer to

Algorithm 5 Backpropagation: Updating weights

1: Update the weights and the bias for the output layer
2: Update the weights and the bias for the hidden layer

this as *heteroassociative memory*. Associative memory neural networks can be feedforward or recurrent. Different associative memory neural networks include:

- Heteroassociative memory - feedforward neural network with input and output vectors different.
- Autoassociative memory - feedforward neural network with input and output vectors different.
- Bidirectional associative memory - recurrent heteroassociative neural network with input and output vectors different, given the input vector the neural network can produce the output vector and given the output vector the neural network can produce the input vector.
- Hopfield neural network - recurrent autoassociative neural network with input and output vectors the same.

In this section we are going to study the Hopfield neural network. As in the case of the other learning algorithms we have studied, there different versions of the Hopfield neural network. We are going to study the Hopfield neural network presented in Fausett [1]. Figure 12 illustrates a Hopfield neural network with input vector of size four with four inputs p_1 to p_4. In autoassociative memory the input and output vectors are the same but to distinguish the input and output in the neural network the output is denoted by the vector $\mathbf{y}$ with four outputs y_1 to y_4.

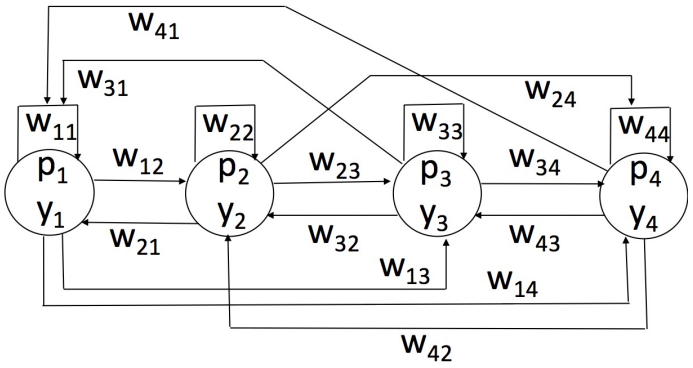

Figure 12: Example of a Hopfield Neural Network

The Hopfield neural network is trained by determining the weight matrix. Once it is trained it is applied to a pattern that contains a missing or incorrect component and the neural network is applied to correct the vector.

If n is the length of the input vector, the weight matrix for the neural network is n x n. The corresponding weight matrix for the neural network in Figure 12 is:

$$\text{W}=\begin{pmatrix} \text{w}_{11} & \text{w}_{12} & \text{w}_{13} & \text{w}_{14} \\ \text{w}_{21} & \text{w}_{22} & \text{w}_{23} & \text{w}_{24} \\ \text{w}_{31} & \text{w}_{32} & \text{w}_{33} & \text{w}_{34} \\ \text{w}_{41} & \text{w}_{42} & \text{w}_{43} & \text{w}_{44} \end{pmatrix}$$

The patterns in the training data can be binary or bipolar. To train the Hopfield neural network the weight matrix is calculated using matrix multiplication (outer products) and addition. The weights on the diagonal of the matrix is 0 as indicated in equation 19:

$$w_{ii} = 0 \quad i = 1, .., n \tag{19}$$

If the training data consists of binary values each weight for $i \neq j$ is calculated using:

$$w_{ij} = \sum_{e} [2p_i - 1][2p_j - 1] \quad i = 1, ..., n \quad j = 1, ..., n \tag{20}$$

where e is an element of the training set

This is equivalent to multiplying the input and output(same as the input vector) vector by 2, subtracting 1, and taking the outer product of the vectors. If the training data consists of bipolar values each weight for $i \neq j$ is calculated using:

$$w_{ij} = \sum_{e} p_i p_j \quad i = 1, ..., n \quad j = 1, ..., n \tag{21}$$

where e is an element of the training set

This is equivalent to taking the output product of the input and output(same as the input vector) vectors. Once the neural network is trained it is applied to identify the stored patterns with missing or incorrect components in the vector representing the pattern. The application algorithm is depicted in Algorithm 6.

The algorithm begins by assigning the output vector to the input vector. It then updates each component y_i of the output vector. The component to be updated is randomly selected. The *while* loop updating each output component y_i represents an epoch. This is done by firstly calculating the sum of the equivalent component in the input vector p_i and the weighted sum of the output vector and the ith column of the weight matrix, s_i. The activation is y_i is calculated by applying the activation function to s_i. θ_i is a parameter for each output component y_i and the best values to use is problem dependent. A common value used for θ_i is 0. Once an activation y_i are calculated the output and input vectors are updated. The process of updating the components of the output vector is continued until the algorithm has converged. The algorithm has converged when there is no change

Algorithm 6 Hopfield Application Algorithm

1: Set the output vector **y** to be equivalent to the input vector **p**:
2: **for** $i \leftarrow 1, n$ **do**
3: y_i=p_i
4: **end for**
5: **while** the algorithm has not converged **do**
6: **while** all components y_i are not updated.
7: Randomly select a component y_i **do**

$$s_i = p_i + \sum_{j=1}^{n} y_j w_{ji} \tag{22}$$

$$y_i = 1 \text{ if } s_i > \theta_i$$
8: $$y_i = y_i \text{ if } s_i = \theta_i$$
9: $$y_i = 0 \text{ if } s_i < \theta_i$$
10:
11: Update the input vector **p** according to the changes to **y**
12: **end while**
13: **end while**

to the y_i components for an entire epoch.

Example

Suppose that you are required to train a Hopfield neural network to store the pattern [1 1 1 0]. All $\theta_i = 0$. As the pattern is binary to train the neural network we firstly multiply the input vector by 2 and minus 1. We then take the outer product of the resulting vector and itself to produce a 4x4 weight matrix. The values on the diagonal are set to 0.

$$\text{W=} \begin{pmatrix} 1 \\ 1 \\ 1 \\ -1 \end{pmatrix} \begin{pmatrix} 1 & 1 & 1 & -1 \end{pmatrix} = \begin{pmatrix} 0 & 1 & 1 & -1 \\ 1 & 0 & 1 & -1 \\ 1 & 1 & 0 & -1 \\ -1 & -1 & -1 & 0 \end{pmatrix}$$

Suppose that the new pattern that the neural network is required to correct is [0 0 1 0] which has errors in the first two components of the vector. So the first step is to set **y** to **p**:

p=[0 0 1 0] y=[0 0 1 0]

The step is to perform the first epoch to update each component y_i. Randomly choose y_1:

$$s_1=0+\begin{pmatrix} 0 & 0 & 1 & 0 \end{pmatrix}\begin{pmatrix} 0 \\ 1 \\ 1 \\ -1 \end{pmatrix}=1$$

$s_1 > \theta_1$, therefore $y_1 = 1$ and y = [1 0 1 0], p=[1 0 1 0]

Randomly choose y_4 as the component to update:

$$s_4=0+\begin{pmatrix} 1 & 0 & 1 & 0 \end{pmatrix}\begin{pmatrix} -1 \\ -1 \\ -1 \\ 0 \end{pmatrix}=-2$$

$s_4 < \theta_4$, therefore $y_4 = 0$ and y = [1 0 1 0], p=[1 0 1 0]

Randomly choose y_2 as the component to update:

$$s_2=0+\begin{pmatrix} 1 & 0 & 1 & 0 \end{pmatrix}\begin{pmatrix} 1 \\ 0 \\ 1 \\ -1 \end{pmatrix}=2$$

$s_2 > \theta_2$, therefore $y_2 = 1$ and y = [1 1 1 0], p=[1 1 1 0]

Randomly choose y_3 as the component to update:

$$s_3=1+\begin{pmatrix} 1 & 1 & 1 & 0 \end{pmatrix}\begin{pmatrix} 1 \\ 1 \\ 0 \\ -1 \end{pmatrix}=3$$

$s_3 > \theta_3$, therefore $y_3 = 1$ and y = [1 1 1 0], p=[1 1 1 0]

The neural network has corrected the input vector, however the application algorithm has not converged. Hence, another epoch needs to be performed. You will find that the algorithm will converge, i.e. there will be no changes to the components of $\mathbf{y}$.

10 Deep Neural Networks

As the name suggests, deep neural networks(DNNs) are neural networks with a number of hidden layers between the input and output layers. The main difference between a deep neural networks and standard multilayer neural networks is that deep neural networks can

have hundreds of hidden layers. These neural networks can be feedforward or recurrent. Deep neural networks can perform supervised and unsupervised learning. Two measures are used to report on the performance of deep neural networks, namely, accuracy and the loss of the neural network calculated by the loss function. The two main areas where DNNs have made a contribution is image processing, where DNNs have outperformed humans, and automatic speech recognition. As in the previous neural networks we have examined, learning takes place from the input layer to the output layer. In deep learning neural networks learning is incremental and the neural network learns layer by layer.

In the case of deep neural networks a *loss function* is used to measure the difference between the activation of the neural network and the target values specified in the training set [4]. The loss function measures the "loss" of the neural network. Once the loss is determined the weights are updated to reduce the loss. One of the factors that effect the choice of the loss function to use is whether the problem at hand is regression or classification. In the case of regression the following can be used as loss functions:

- Mean squared error
- Mean squared logarithmic error
- Mean absolute error

The mean squared error is the function generally used for regression. The following loss functions can be used for binary classification:

- Binary cross-entrophy
- Hinge loss
- Squared hinge loss

Similarly, loss functions for multiclass classification include:

- Multiclass cross-entrophy
- Sparse multiclass cross-entrophy
- Kullback Leibler divergence

Once the error is calculated using the loss function the weights are updated to reduce the loss using an *optimizer* [4]. Optimizers commonly used by deep learning neural networks include:

- Backpropagation
- Gradient descent

- Stochastic gradient descent
- Mini-batch gradient descent
- Momentum
- Nesterov accelerated gradient
- Adagrad optimizer
- AdaDelta optimizer
- Adam
- RMSProp

The Adam optimizer has proven to be the best optimizer in terms of lower training time and efficiency. Optimizers usually require a learning rate in the weight update function. The most appropriate learning rate to use is problem dependent and a parameter value for the neural network. If the backpropagation neural network is used the loss function must be differentiable.

Different activation functions can be used for the different layers in the neural network. It is important to choose an appropriate function for the output layer of the neural network. Activation functions commonly used include:

- Sigmoid
- Softmax
- Tanh - Hyperbolic Tangent
- ReLU - Rectified Linear Unit
- Leaky ReLU
- Swish

The are various types of deep neural networks. Commonly used deep neural networks include:

- Convolutional Neural Networks
- Autoencoders
- Restricted Boltzman Machines
- Deep Belief Networks

- Long Short Term Memory (LSTM)

Deep neural networks are susceptible to the problem of *overfitting*. Overfitting is essentially the problem of a neural network obtaining very high accuracies on the training set but cannot generalize well on the test set on which it achieves low accuracies. Therefore it is important to always report on the accuracies of a DNN on both the training and test sets and any conclusions drawn should be based on the performance on the test set. *Regularization* is used to overcome overfitting. *Early-stopping* of the learning algorithm can also be performed to reduce overfitting. Reducing the size of the neural network can also contribute to overcoming overfitting. Adding dropout has also assisted in overcoming overfitting.

In order to generalize well deep neural networks require sufficient data for learning. However, in some problem domains such data may not be available. In these instances *transfer learning* can be used. Transfer learning involves pretraining a neural network on a set of data. The pretrained neural network is then used to solve the problem for which insufficient data is available. As can be expected if the pretrained neural network is applied as is it will not generalize well. Hence, different transfer learning techniques are employed to keep certain aspects of the pretrained neural network as is and refine other aspects. Aspects of the pretrained neural network that are reused include features and/or weights. For example one technique keeps all layers as is and only the last layer of the neural network is retrained. Alternatively, the layers to leave as is and those that should be retrained can be done selectively. ImageNet is a database of images that is commonly used for pretraining deep neural networks for computer vision. Pretrained neural network models are needed for transfer learning. These are usually made available via Python libraries. Examples of pre-trained deep neural networks that can be used for computer vision are:

- VGG 16
- VGG 19
- Inception V3
- XCeption
- ResNet 50

Examples of pre-trained neural networks for natural language processing include:

- Word2Vec
- GloVe
- FastText

10.1 Convolutional Neural Networks

Convolutional neural networks(CNNs or CovNets) became popular when they were used by Alex Krizhevsky to win the ImageNet Challenge. The winning CNN is now known as *AlexNet*. Convolutional neural networks have been most effective for image classification. Hence, they will be introduced in the context of image classification. Image classification involves taking an image as input and outputting the class the image belongs to, e.g. dog, horse, etc. The output of the classifier in this case is a single class or a set of probabilities indicating the extent to which an image belongs to each class. In the case of the output being probabilities the sum of the probabilities is 1.

The first step in applying a CNN to an image for classification is converting the image to a matrix of integer values representing pixels in the range 0 to 255. In the case of colour images the image is mapped to three two dimensional arrays, for red, green and blue, stacked into a matrix. In the case of grayscale images the matrix is a single two dimensional array which forms the matrix. The size of the matrix is dependent on the resolution of the image. The larger the size of the matrix the more processing is required. Hence, in some instances *preprocessing* is performed to reduce the scale of the image.

Convolutional neural networks differ from the previous neural networks that we have studied in that not all layers are *fully connected*, some layers are sparsely connected. An example of a fully connected neural network is illustrated in Figure 13 and a sparsely connected neural network in Figure 14.

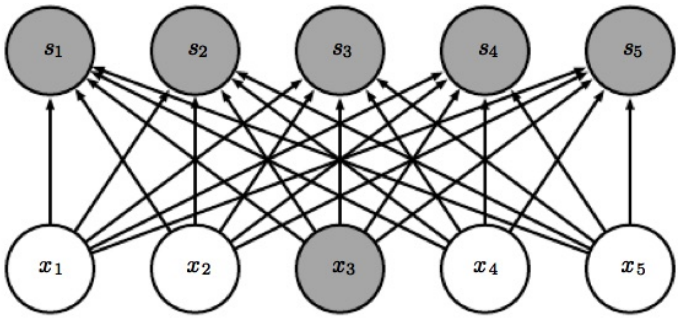

Figure 13: Fully Connected Layer [5]

As can be seen from Figure 13 in a fully connected layer every input unit is mapped to an output unit via a weight. However, in a sparsely connected neural network every input is not mapped to every output. This results in weights being shared and is referred to as *parameter sharing*.

A convolutional neural network is comprised of a number of different layers leading to the output layer. Most of the layers are sparsely connected with the at least the final layer being fully connected. A convolutional neural network is comprised of one or more of the following layers:

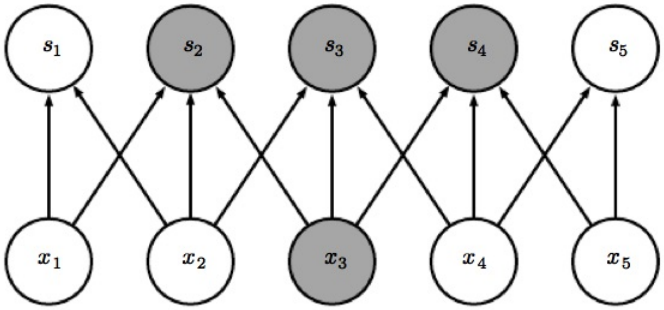

Figure 14: Sparsely Connected Layer [5]

- Convolutional layers - A convolutional layer is usually the first layer after the input layer. In addition to this convolutional layers can exist deeper in the CNN. In previous neural networks matrix multiplication is performed to multiply the weight matrix with input values. In this layer a *convolution operator* is applied. The convolution focuses the learning at this stage on a particular aspect of the image, e.g. an edge.

  Training is performed by determining the weight matrix which is referred to as *filter* or *kernel*. A kernel is a 2D weight matrix, while a filter consists of a number of kernels and represents 3D structures. Initially kernel/filer values are randomly selected or dependent on the convolution operator.

  The output of the convolutional layer is a *feature map*. The feature map is a result of applying the filter to a the image represented by the pixels. Different convolutional operators can be applied and these are used with different filters/kernels. Examples of convolution operators include: identity, edge detection, sharpen, box blur.

  Applying a convolution to an array of pixel values can be visualized as moving a window from the top left hand corner of the pixel matrix to the bottom right hand corner and applying the filter/kernel to produce the feature map. The area that the filter/kernel is applied to in the input matrix is called the *receptive field or area* and has the same dimensions as the filter/kernel. When the sliding the window and applying the filter from the top left hand corner, the number of pixels moved over is called the *stride*. This is usually 1. If it has a value of 2 then two pixels are moved over instead of 1, i.e. 1 pixels is "jumped over", which will further reduce the dimensionality of the feature map.

  In some instances *zero padding* may be performed on the input matrix. This involves adding a border of zeroes. Whether to perform zero padding or not is a hyperparameter for the CNN.

  This is a sparsely connected layer and hence also reduces the dimensionality of the input matrix.

- Nonlinear (ReLU) layers - The purpose of this layer is to add nonlinearity. The activation function used is the ReLU function, namely, $f(x) = max(0, x)$.
- Pooling layers - The purpose of this layer is to reduce the number of parameters. It achieves this by reducing the size of the feature map. This also reduces overfitting.
- Fully connected layers - At least the last layer, i.e. the output layer, is fully connected. In some instances the last two layers are fully connected. The softmax activation function is usually used in the units of fully connected layers. In the case of transfer learning, this is the layer that is usually retrained

You can create the architecture of your convolutional neural from scratch or alternatively you can using an existing architecture from a library. Some of the available architectures are also pre-trained on images from ImageNet and hence the entire neural network does not need to be retrained by only certain layers, usually the output layer at least. This process is called *transfer learning*. Existing CNN architectures include:

- LeNet - First CNN
- AlexNet - Winner of the ImageNet 2012 challenge
- GoogLeNet - Winner of the ImageNet 2014 challenge
- VGGNet - Second place winner in the ImageNet 2014 challenge
- ResNet - Winner of the ImageNet 2015 challenge
- DenseNet - Fairly recent CNN

10.2 Autoencoders

Autoencoders are deep learning neural networks that perform *unsupervised learning* [6, 5]. Hence, the training set does not contain labelled data. The input of an autoencoder is the the same as the output[6, 5]. The autoencoder learns its input and outputs the input in a different representation. Common applications of autoencoders include dimensionality reduction, image compression and information retrieval. An autoencoder is a feedforward neural network consisting of three layers:

- Input layer - Input to be compressed, e.g. image.
- Hidden layer- Extracts features. There may be more than one hidden layer.
- Output layer - Outputs a compressed version of the input.

Training the autoencoder is performed in the same way as previous neural networks, namely, finding the weights and bias. The weighted sum of the inputs and bias form input to the activation function. Activation functions commonly used are sigmoid function and rectified linear unit (ReLU). The backpropagation and stochastic gradient descent optimizers are used to optimize the weights and bias. The loss function typically used by autoencoders is *mean squared error* or *binary cross entrophy*. As there is no labelled data the loss function measures how accurately the input is reconstructed as the output depending on the specific objective of the problem being solved.

The autoencoder consists of two components. The first is the *encoder* which firstly compresses the input and then produces the code. The second component the *decoder* reconstructs the input using the code. Both the encoder and decoder are fully connected.

11 Online Resources

The following books are free and can be accessed online to supplement these notes and the slides provided:

M. Hagan, H. Demuth, M. Beale, and O. De Jesus, Neural Network Design.
https://hagan.okstate.edu/NNDesign.pdf, 1996.

D. Kriesel, A Brief Introduction to Neural Networks.
http://www.dkriesel.com/en/science/neural_networks, 2005.

I. Goodfellow, Y. Bengio, and A. Courville, Deep Learning. MIT Press, 2016.
http://www.deeplearningbook.org/

References

[1] L. Fausett, *Fundamentals of Neural Networks-Architectures, Algorithms and Applications*. Pearson, 1993.

[2] M. Hagan, H. Demuth, M. Beale, and O. De Jesus, *Neural Network Design*. https://hagan.okstate.edu/NNDesign.pdf, 1996.

[3] D. Kriesel, *A Brief Introduction to Neural Networks*. http://www.dkriesel.com/en/science/neuralnetworks, 2005.

[4] F. Chollet, *Deep Learning with Python*. Manning Publications Co., 2018.

[5] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016.

[6] L. Deng and D. Yu, *Deep Learning Methods and Applications*. Now, 2014.