# **PHYS 390 – INDEPENDENT STUDY FINAL REPORT**

**Prepared by:** Mustafa Akın Yılmaz

**Supervisor:** Özgür Esat Müstecaplıoğlu

**Subject:** Introduction to Artificial Neural Networks

**Table of Contents**

[**PHYS 390 – INDEPENDENT STUDY FINAL REPORT** 1](#_Toc520722044)

[**Introduction** 2](#_Toc520722045)

[**Biological Nervous System and Human Brain** 2](#_Toc520722046)

[**Artificial Neurons (Perceptron)** 3](#_Toc520722047)

[**Activation Functions** 4](#_Toc520722048)

[**Step Function** 4](#_Toc520722049)

[**Linear Function** 5](#_Toc520722050)

[**Sigmoid Function** 5](#_Toc520722051)

[**ReLu (Rectified Linear Unit) Function** 6](#_Toc520722052)

[**Softmax Function** 8](#_Toc520722053)

[**Artificial Neural Networks (ANN)** 9](#_Toc520722054)

[**ANN Structure** 9](#_Toc520722055)

[**Learning in ANN (ANN Training)** 10](#_Toc520722056)

[**Learning Types** 10](#_Toc520722057)

[**Error - Correction Learning** 11](#_Toc520722058)

[**How to Train an ANN** 13](#_Toc520722059)

[**ANN Example: Hand-written Digit Recognition** 21](#_Toc520722060)

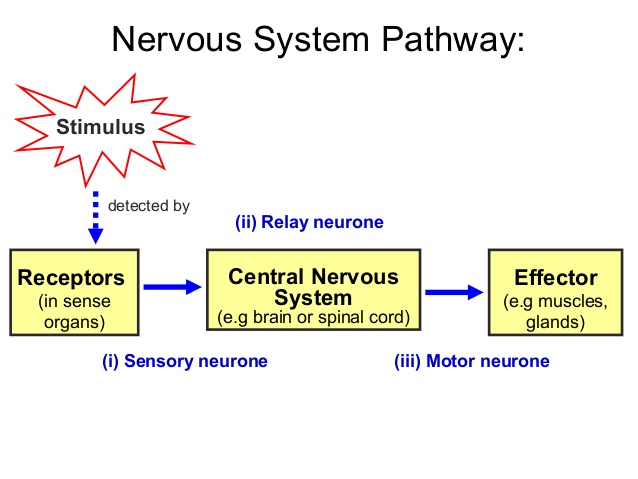
[**REFERENCES** 22](#_Toc520722061)

# **Introduction**

Artificial Neural Networks (ANN) are systems developed with the aim of deriving and discover new information automatically through learning like the characteristics of the human brain. Inspired by the human brain, ANNs have emerged as a consequence of the works on mathematical modeling of the learning process. For this reason, studies on this subject began with the modeling of neurons, the biological units that make up the brain, and their applications in computer systems. Later on, Later, in parallel with the development of computer systems, artificial intelligence has become widely used.

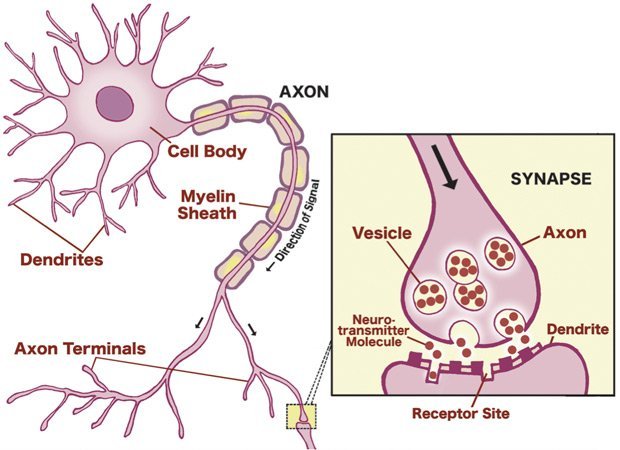
# **Biological Nervous System and Human Brain**

The biological nervous system can be thought of as a three-tiered system in which the brain, which constantly receives information, interprets it and produces an appropriate decision in its center.



The receptor nerves convert the stimuli they perceive from inside or outside the organism into electrical signals that transmit information to the brain. The information in the central nervous system is evaluated in the direction of forward and backward between the receptor and the effector nerves and appropriate responses are produced. Effector nerves turn the electrical impulses produced by the brain into appropriate responses as output of the organism.

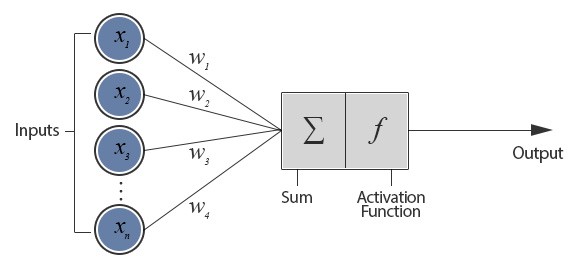
Fundamental processing unit of the central nervous system is the nerve cell and there are about 10 billion nerve cells in the human brain. As can be seen in figure 2, a biological nerve cell consists of four parts: a cell body, an axon, dendrites, and thin extensions between the axon and the dendrite of the other nerve cell.



Dendrites are responsible of transmitting information from other cells in the form of a tree structure to the cell body. The cell body collects the signals from the dendrites and transmits them to the axon part of the nerve cell. These collected signals are processed by the axon and sent to synapses. Synapses transmit newly produced signals to other nerve cells.

# **Artificial Neurons (Perceptron)**

An artificial neuron is a mathematical function conceived as a model of biological neurons. Artificial neurons are elementary units in an artificial neural network. An artificial neuron receives one or several inputs (acts like dendrites), sums them, and produces an output (acts like axon terminal). Usually, all inputs are separately weighted (these weights can be thought as synapses) and the sum is passed through a linear or non-linear activation function.



## **Activation Functions**

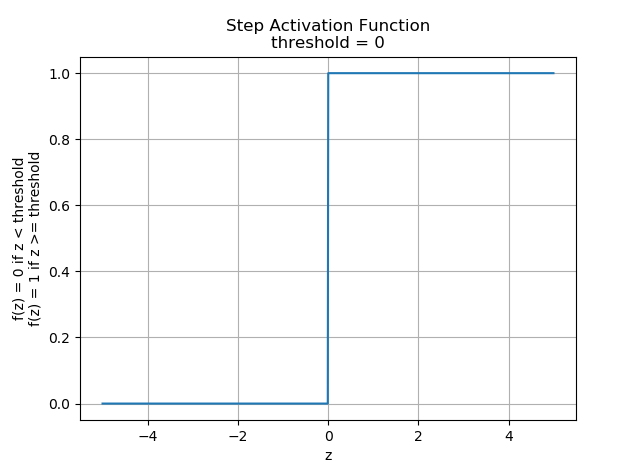
Biological nerve cells have evolved to fire when a stimuli passes a certain threshold. As explained in artificial neurons part, an artificial neuron calculates weighted sum of inputs. This sum can be anything ranging from -∞ to +∞. The neuron really doesn’t know the bounds of the value. So how do we decide whether the neuron should fire or not? Inspired by biological neurons, We decided to add “activation functions” for this purpose. To check the sum value produced by a neuron and decide whether outside connections should consider this neuron as “fired” or not. Or rather let’s say  “activated” or not. Linearly and especially non-linearly activated neurons give an artificial neural network the ability to capture complex patterns in the input that may not be meaningful to a human being but which may be meaningful to computer.

There exists linear and several non-linear activation functions. Each function has some advantages and disadvantages for training a neural network that will be explained in detail later. Below, most widely used activation functions are explained.

Note: Let’s suppose “z” be weighted sum of all inputs (maybe plus a bias) for an artificial neuron.

### **Step Function**

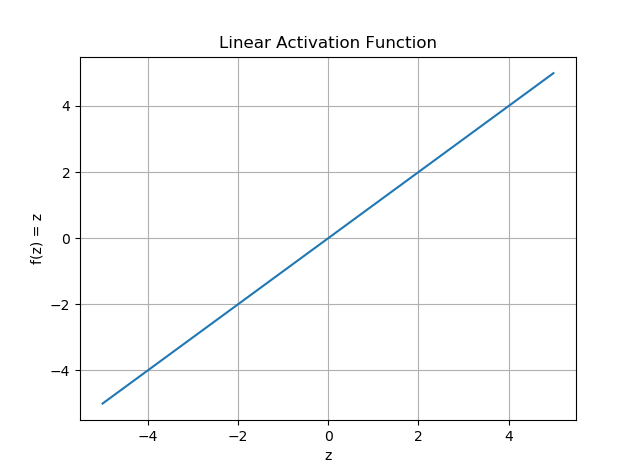
The first function that comes to our mind is step activation function. If the value of z is above some certain threshold, declare the neuron as “activated”. If less than the threshold, then say it is not.



The step function outputs 1 (activated) when value >= threshold, and 0 (not activated) when value < threshold. This makes a non-linear activation function for a neuron. However, it has some drawbacks. Suppose we will create a binary classifier something returns “yes” or “no” (activated or not activated). The step activation function could exactly do that. For instance, it can say whether a tumor is malignant or benign. The first drawback comes here. This activation cannot say about probabilities of two classes (malignant and benign in this case). It just states whether malignant or benign. Since the class probabilities are very important in lots of cases like the tumor case and bank fraud detections, step function is not widely used in real world applications.

### **Linear Function**

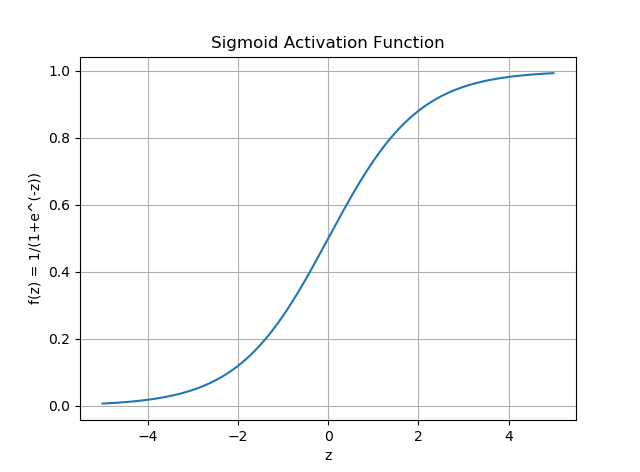
This activation function is in the form of **f(z) = cz** where c is a constant and z is the input. That means the output of a neuron having linear activation will be a linear combination of inputs (weighted sums).



Since linear activation has an output ranging from -∞ to +∞, it is not useful for classification but useful for regression problems such that price estimations or image restorations. The problem with linear activation is, even if we connect many artificial neurons all having linear activation, the output is still going to be a linear combination of inputs. So no matter how many neurons we add, we can replace these neurons with a single one. All in all, a network of neurons with only linear activation cannot capture complex and non-linear patterns in the data.

### **Sigmoid Function**

Sigmoid activation is in the form of **f(z) =**  where z is the input features. Its plot is shown in figure below. As can be seen in sigmoid functions plot, it is like a “smooth” step activation function. Now lets take a glance at details of that function. First, sigmoid has a non-linear nature, so combinations of this function are also non-linear. This non-linearity basically and maybe most importantly gives sigmoid activation ability of capturing complex, non-linear patterns in input. Sigmoid activation is also a non-binary activation function that can take any value between 0 and 1. If this feature is taken into account, it is not wrong to say sigmoid is an “analog” like actvation function unlike step activation function. If we look at the plot carefully, it is obvious that between z values -2 to 2, f(z) values are very steep. Which means, any small changes in the values of z in that region will cause values of f(z) to change significantly. That means this function has a tendency to bring the f(z) values to either end of the curve.

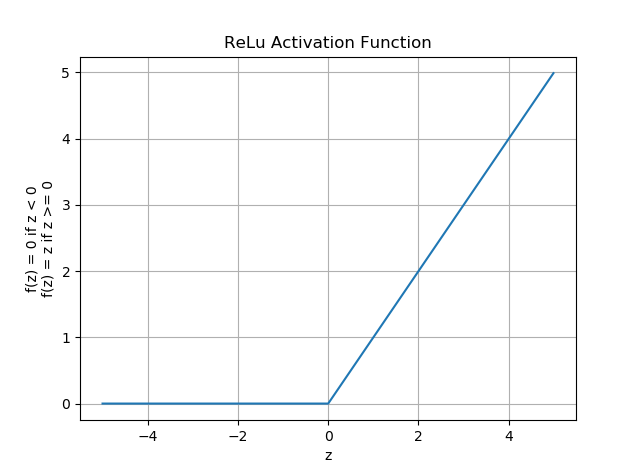
****

Looks like it’s good for a classifier considering its property. It tends to bring the activations to either side of the curve (above z = 2 and below z = -2 for example). Making clear distinctions on prediction. That analog characteristic gives sigmoid function a crucial property: It can be used to detect probabilities for “yes” or “no” questions explained in step function part. Unlike step activation, sigmoid function can tell us a tumor is maligmant with a probablity of “p”. Another advantage of this activation function is, unlike linear function, the output of the activation function is always going to be in range (0,1) compared to (-inf, inf) of linear function. So we have our activations bound in range. It is nice because it won’t blow up the activations then.

Even if it looks like sigmoid is a very adventegaous function, it has a drawback and in fact a very serious one. If you notice, towards either end of the sigmoid function, the f(z) values tend to respond very less to changes in z. The gradient at that region is going to be small. It gives rise to a problem of “vanishing gradients”. So what happens when the activations reach near the “near-horizontal” part of the curve on either sides? Gradient is small or has vanished (cannot make significant change because of the extremely small value). The artificial neural network refuses to learn further or is drastically slow. I will explain in detail what “gradient” is and how it is used in learning process of an artificial neuron in later parts. For now, just remember gradients are very crucial and small gradients may cause learning of a neuron to stop.

### **ReLu (Rectified Linear Unit) Function**

ReLu function has the form of **f(z) = max (0,z)** where z denotes input features. As can be seen in its plot below, it gives an output f(z) = z if z is positive and f(z) = 0 otherwise. At first look ReLu would look like having the same problems of linear function (not bounded range, and not capturing complex structures), as it is linear in positive axis. First of all, ReLu is non-linear in nature. Hence combinations of ReLu are also non-linear. Its slope is 0 and 1 in negative and positive axis respectively.



The first thing to mention in ReLu, it can take values between 0 and +∞. This means it can blow up the activation. It is a disadvantage of course, but ReLu activation has a very big advantage. It is not computationally costly like sigmoid function. As discussed in sigmoid activation, it causes neurons to fire in an analog way. That means almost all activations will be processed to describe the output of a network. In other words the activation is dense. This is costly. We would ideally want a few neurons in the network to not activate and thereby making the activations sparse and efficient. Relu provides this benefit. Imagine a big network having many many artificial neurons. Because of the characteristic of ReLu function (0 is negative axis), some activaions yield 0 (not fired) and others will be fired. This means fewer neurons are fired unlike sigmoid.

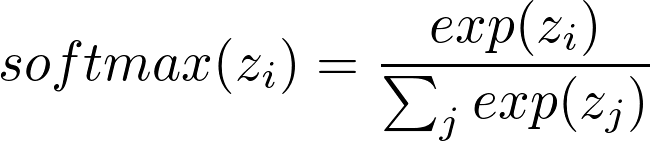
Another advantage in point of computational cost and time is in terms of gradients. Because of the horizontal line in ReLu (z < 0), the gradient can go towards 0. As I will explain in learning process of neurons later, neurons having 0 gradient go into a state where they stop responding to variations in error/ input. This problem can cause several neurons to just die and not respond making a substantial part of the network passive. There are variations in ReLu to mitigate this issue by simply making the horizontal line into non-horizontal component. For example f(z) = 0.01z for z < 0 will make it a slightly inclined line rather than horizontal line. This is “leaky ReLu”. There are other variations too. The main idea is to let the gradient be non zero and recover during training eventually.

On the whole, ReLu is less computationally expensive than sigmoid because it involves simpler mathematical operations. That is a good point to consider when we are designing deep artificial neural networks.

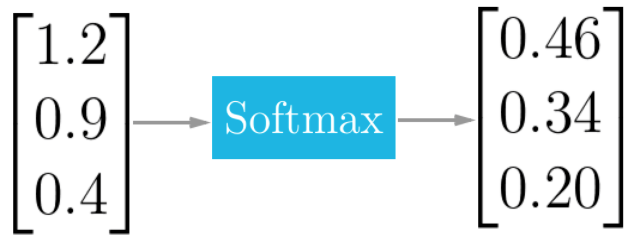
### **Softmax Function**

We have seen that sigmoid activation can be used as a classifier for two classes (malignant or benign). In that case, sigmoid activation’s output just gives us the probability of one class (say malignant, probabilty = p). Since sum of the probabilities of all classes equal to 1 and if there exists just 2 classes, then the other class’ probabilty is certain (benign, probability = 1 - p). To sum up, using only one neuron having sigmoid activation function can tell us two probabilities of two classes.

Now let say we wish to classify an object in 3 different classes (say car?, ship?, plane?). In classification with more than 2 classes scenario, sigmoid activation becomes useless because of the explanation above. Here, “softmax” activation takes over the duty. The word “soft” states that it gives us the probabilities of each class, do not give step function like results. To be clear, it does not say Pcar = 1, Pship = 0, Pplane = 0, but it says Pcar = 0.7, Pship = 0.2, Pplane = 0.1.



Above expression gives the probability of object being in class “i”. Here “z” is the input vector to the softmax activation.



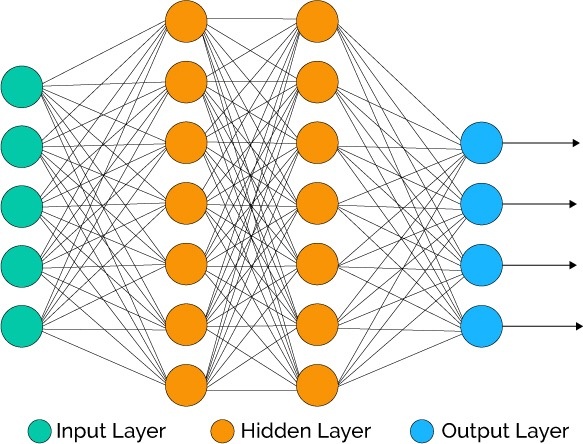
Just for an example consider the vector above. It is for 3 classes case like our car, ship, plane example. Let say weighted sum of a neuron returns the vector that is the input to softmax layer. After calculating the output after softmax layer using given formula, the output is nothing but the probabilities of 3 classes. If we look at the probabilities, we can see that 0.46 + 0.34 + 0.20 = 1 as expected.

# **Artificial Neural Networks (ANN)**

Artificial neural networks are mathematical systems consisting of one or many artificial neurons linked together in a weighted fashion. These neurons can be placed to share the same layer and receive the same input set (single layer network), or they can be placed in different layers and receive the output of the neurons in the previous layer as input (multi layer network). The artificial neurons correspond roughly to real neurons and are interconnected in a network, which forms artificial neural networks.

## **ANN Structure**

Artificial neural networks are basically composed of 3 layers. These are, respectively, input layer, hidden layer(s) and output layer. Input layer is the layer that comes from the outside world into the artificial neural network. In this layer, there are as many cells(neurons) as there are inputs, and the inputs are transmitted to the hidden layer without any processing. Hidden layer processes the information received from the input layer and transmits it to the next layer. The number of hidden layers and the number of cells in the hidden layer may vary from network to network (even number of hidden layers may be 0). Cell numbers in hidden layers are independent of input and output numbers. Output layer handles the information coming from the hidden layer and sends it to the outer world in accordance with the incoming input to the input layer. The number of cells in the output layer can be more than one . Each output cell has one output. Each cell is tied to all the cells in the previous layer. The structure of a possible ANN is shown below.



## **Learning in ANN (ANN Training)**

Until now, we have seen artificial neurons and how they work. Also we have led in the subject of artificial neural networks that may consist of many many artificial neurons. In previous parts, we glanced at biological nervous system. Undoubtedly, the most important feature of the brain, which is at the center of the central nervous system, is the ability to learn. Despite the accelerated developments in science, it is not fully clarified how the brain exactly does that. Scientists believe that, during the learning process, the brain’s neural structure changes by the way of increasing or decreasing the strength of it's synaptic connections depending on their activity. This is why more relevant information can be recalled. Because more relevant information will have stronger synaptic connections.

We mentioned that artificial neural networks are a product of attempts to simulate real neural networks. So, learning process is indispensable for not only biological neural networks but also artificial neural networks. Even though it is simplified, artificial neural networks can model this learning process by adjusting weighted connections (training a network) between neurons in the network. This method can simulate strengthening and weakening of the synaptic connections found in the brain.

Learning ability of artificial neural networks are very useful for some problems. Now let’s consider handwritten digit recognition problem. Since all humans have a unique hand writing and there are many ways to write a digit (for example 9), it is extremely difficult to identify features by hand and extremely difficult to write rules for this problem. Artificial neural networks can solve these kind of problems much better.

### **Learning Types**

#### **Supervised Learning**

The artificial neural network is trained before it is used. During training, both the input values and the desired output values for those input values are given to the system. By comparing the desired output with the output produced by the network, an error calculation is made and the weights are updated.

#### **Unsupervised Learning**

There are no supervisors who help the system to learn. Only the inputs are given to the system, the relationships between the parameters in the examples are expected to be learned by the system.

#### **Reinforcement Learning**

Input data is given to the network and the result is asked to be evaluated by the supervisor. The network weights are updated by rewarding and punishing. The aim of reinforcement learning is to maximize the reward the system receives through trial-and-error.

**Note:** We will cover supervised learning in this study.

### **Error - Correction Learning**

Error-correction learning, used in supervised learning, is the technique that compares the ANN’s output with the desired output (target) to calculate the error signal, and uses that error to adjust weights between the neuron connections. The purpose of this training process is to minimize this error signal, and to bring ANN’s output and target output closer. The most direct method to train a network by error signals is called backpropagation algorithm. Before I dive into the details of learning process and backpropagation algorithm, I will introduce 2 commonly used error types in supervised learning. I will be using **L (loss)** to donate error for a single output-target pair, **E (error)** to donate the total error in the system.

#### **Mean Squared Error (MSE)**

Let **y** be the single calculated output of a network and **t** be the single desired output (target) corresponding to **y**. Then,

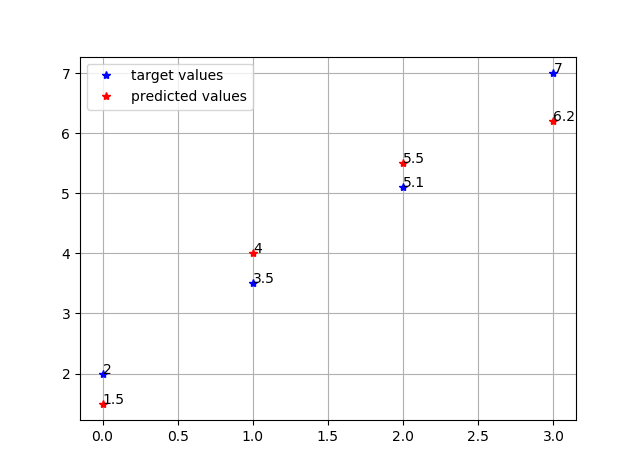
**L =**  for a single output-target pair.

Now let’s suppose there are **M** different output-target pairs in the system. Then,

**E =**  is the average of each squared errors in output-target pairs (MSE).

MSE tells us how well predictions of a system fit to the real values. Looking at the formula above, we can easily say that, when MSE approaches to zero it means that output and target values get closer to each other. When MSE is zero, then all the target-output pairs perfectly match which means they are equal. MSE is usually used in regression problems like house price prediction since in regression problems output can take any number without bounds. Of course, MSE can also be used in classification problems (yes/no). However a better and widely used choice for error in these problems is cross-entropy error which I will discuss in the following part.

Let me give an example on calculating MSE.



target = [2, 3.5, 5.1, 7]

predicted = [1.5, 4, 5.5, 6.2]

E = x [] = 0.3249

#### **Cross - Entropy Error**

Cross-entropy error is used in measuring performance of classification problems whose output is a probabilty value that can take a value between 0 and 1. Cross-entropy error increases as the predicted probability diverges from the real label. So if the actual label is 1 (yes) and predicted probability of being class 1 is 0.05, then cross-entropy error will be a high value. Like MSE, a perfect fit would result in 0 cross-entropy error.

Cross - entropy error is calculated as the following.

**L =** is the single output-target pair loss,

**E =** is the cross-entropy error.

Here: C 🡪 the total number of classes in the problem,

M 🡪 the total number of samples,

log 🡪 natural logarithm,

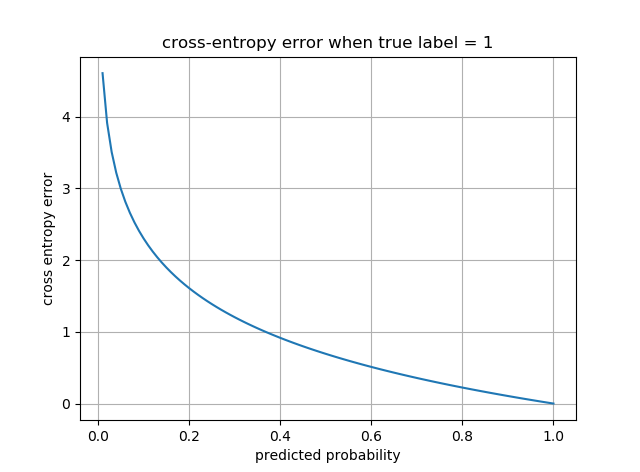
y 🡪 binary indicator (0 or 1) if class label c is the correct classification for observation o,

p 🡪 predicted probability observation o is of class c.

In binary classification where C = 2, loss can be simplified as:

**L = - [y log(p) + (1 - y) log(1 - p)]**

**E =**  ,since there are two clasess and if probability of being one class is p than the probability of being the other class is 1-p.



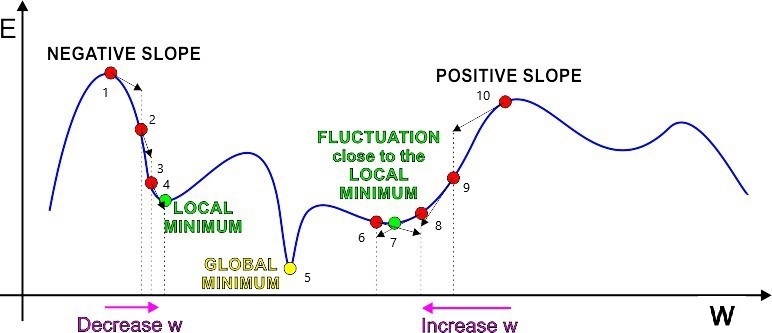
### **How to Train an ANN**

So far, we have seen 2 most commonly used error functions for supervised learning. We have also seen that our ultimate aim of training a network is to minimize total error in the network. It is now time to ask how these errors are used in training, what their roles are on updating connection weights in an ANN. To answer these questions, first I should introduce “gradient descent” concept.

#### **Gradient Descent**

The meaning of “gradient descent” concept lies in its name. Gradient means “slope” and descent means “going down”. Gradient descent follows the slope of curve of a function and descends to combination of parameters in the function that gives the minimum value.

Let’s suppose curve of an error function having a single parameters w be **E(w)** is like this:



Our objective is to find the parameter w that minimizes E(w) (point 5) in the example, and to find it we first chose an initial point. The optimum parameter is called “global minimum”. There are three cases that w can take initially.

**Case 1:** Initial value of w is on the falling edge (negative slope), i.e point 2.

Ideally, we want to reach the global minimum (point 5), but gradient descent will try to reach point 4 which is a global minimum point. The reason for that is gradient descent never climbs, once it reaches point 4 it will stay there. At falling edge points, (slope of the error function with respect to parameter w) is negative. So we should increase value of w with some step size to reach a minimum point. This step size is called “learning rate” and it is commonly donated as “α”.

**Case 2:** Initial value of w is on the rising edge (positive slope), i.e point 9.

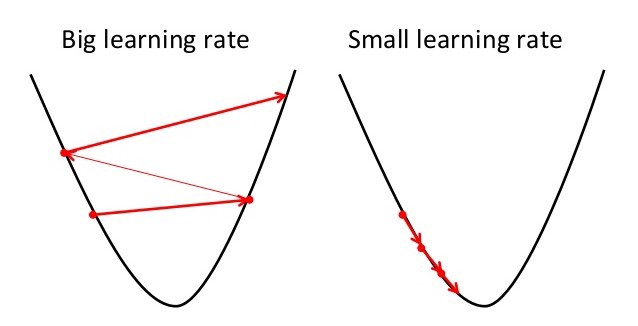
In that case is positive and we should decrease value of w. Gradient descent will try to reach point 7. Once w reaches the value at point 7, gradient descent will stop working since the slope of the error function is 0 at that point.

**Case 3:** Initial value of w is on minimum points (slope = 0), i.e point 5.

In that case, nothing happens since slope is zero. Initial value of w remains unchanged.

If you noticed that, gradient descent does not guarantee you to reach global minimum. The reason is, gradient descent cannot know exact shape and of the error function. In above example, we have seen an error function that depends on only one parameter. As number of parameters increase, error function will be more and more complex. It will be multiple dimensional, and as humans we will not be able to plot it.

If we consider above example again and assuming we know the shape of error function, we may avoid reaching local minimum points. We can do that by adjusting learning rate. As I explained in case 1, learning rate is the step size that tells us how much we increase or decrease the value of parameters at each gradient descent step.



If the value of α is too big, than we may end up bouncing let say from point 3 to somewhere between point 4 and point 5. Hence, we may never reach point 4 (local minimum) or even reach any minimum including global minimum. If the value of α is too small, we will reach a minimum point very very slowly. Thus gradient descent algorithm will take more time to converge a minimum. In the end, an appropriate learning rate is determined by trial and error as luck is an effective factor.

So far, I have tried to explain gradient descent verbally and aimed to give you a clear understanding. Now, let me describe it in a form of equation:

*Repeat until convergence for all parameters (slope of error function with respect to all parameters):*

*w := w – α() α is learning rate and positive.*

When the slope is positive, we should decrease w. Since α() is positive, by substracting it from w will result in a decrease in w. When the slope is negative, we should increase w. Since α() is negative, by substracting it from w will result in an increase in w.

This weight update rule is one of the most popular optimization algorithm used in training of an ANN.

#### **Weight Updates in ANN**

I have shown that, gradient descent algorithm helps us to find neural network parameters that minimize error in the network. For an ANN to learn, we should provide a set of samples called “training set” to the network. This set contains samples of inputs to the network and desired corresponding outputs. When training set is specfied, ANN architecture is built and network weights are randomly initialized, training can be started. Learning of ANN consists of two phases.

##### **Forward Propogation**

This phase begins by showing input features to the network. Without any change, inputs are transferred to next layer of neurons. All neurons process coming information (weighted sum + activation), and finally the output layer calculates the output. Below I provide an example.



To work with numbers, here are the initial weights, biases and input-output pairs (training set). Activation function of hidden layer and output layer is sigmoid. Lets see what neural network calculates for given inputs i1, i2.

h1 = i1\*w1 + i2\*w2 + b1 = 0.05\*0.15 + 0.10\*0.20 + 0.35 = 0.3775

= sigmoid(h1) =  = 0.59326999 🡪 after passing weighted sum to the activation.

h2 = i1\*w3 + i2\*w4 + b1 = 0.05\*0.25 + 0.10\*0.30 + 0.35 = 0.3925

= sigmoid(h2) =  = 0.59688437

Now we calculated outputs of the hidden layer. Lets do the same thing for output layer.

o1 = \*w5 + \*w6 + b2 = 0.59326999\*0.4 + 0.59688437\*0.45 + 0.60 = 1.10590596

=  = 0.75136507

o2 = \*w7 + \*w8 + b2 = 0.59326999\*0.5 + 0.59688437\*0.55 + 0.60 = 1.22492140

=  = 0.77292846

After all, the network calculates the output given set of inputs i1 and i2. Now it is time to compare the calculated result with the desired one to compute error. Then using back-propogation algorithm we update connection weights.

##### **Backward (or simply Back) Propogation**

In forward propogation part, we have learned how to calculate output that the network produces given some inputs. After calculating the output, we are now able to compare the results with desired ones. Back propogation allows neural networks to pass calculated error, and allows us to calculate gradients of each weight and bias in the network. As discussed in gradient descent part, general formula for weight update is:

*:= – α() for all weights*

*:= – α() for all biases*

We do updates for many iterations until error is converged to a minimum. Now, lets witness how back-propogation works, and lets continue with the example shown in forward propogation.



In part 1, we have calculated the following:

h1 = 0.3775 = 0.59326999

h2 = 0.3925 = 0.59688437

o1 = 1.10590596 = 0.75136507

o2 = 1.22492140 = 0.77292846

Now lets calculate MSE:

E = ∑

Note: I added because it will be cancelled out by the exponent when we differantiate it. Since the gradients are eventually multiplied by a learning rate, it does not matter that I introduce a constant here.

We know that the target output and neural network’s output.

= = = 0.27481108

= = = 0.02356003

= + = 0.29837111

*Output Layer*

Now, we find total error. Lets start with calculating w5’s gradient. We now want to know how much change in parameter w5 affects the total error. We want to calculate which is partial derivatiof of w5 with respect to . If we apply chain rule:

= \* \*

= 2\* \*() + 0 = 0.74136507

Second term is 0 since derivative of with respect to is 0. Now lets calculate .

=  =0.75136507

= \* (1 - = 0.75136507\*(1 - 0.75136507) = 0.18681560

Finally, calculate :

o1 = \*w5 + \*w6 + b2 = 1.10590596

= = 0.59326999

Final result:

= \* \* = 0.74136507\*0.18681560\*0.59326999 = 0.08216704

We found gradient of w5. We know its initial value = 0.4. So we can update it as following:

w5 = 0.4 – α\*0.08216704 where α is learning rate and we are free to choose its value. If we set it 0.5:

w5 = 0.4 – 0.5\*0.08216704 = 0.35891648

Above calculations are for only w5. We can do the same calculations for other parameters in output layer (w6,w7,w8,b2). We perform backpropogation again and again once we find the updated parameters to converge a minimum.

*Hidden Layer*

When we finish our job for output layer and find all gradients for output layer, we can begin working on hidden layer parameters. Lets focus on w1. We are trying to find:

= \* \*

First find .

= + since affects both and . By applying chain rule:

= \* and = \* 🡪 = 0.74136507\*0.18681560 = 0.13849856

We know that o1 = \*w5 + \*w6 + b2 🡪 = w5 = 0.4

= 0.13849856\*0.4 = 0.05539942

Applying the same process for we get:

= - 0.01904912.

Hence,

= + = 0.05539942 + (- 0.01904912) = 0.03635030

Now we need to find . We know = = 0.59326999

= \* (1-) = 0.59326999 \* (1-0.59326999) = 0.24130071

Finally, we need to calculate and we know h1 = i1\*w1 + i2\*w2 + b1

= i1 = 0.05

All in all:

= \* \* = 0.03635030\*0.24130071\*0.05 = 0.00043857

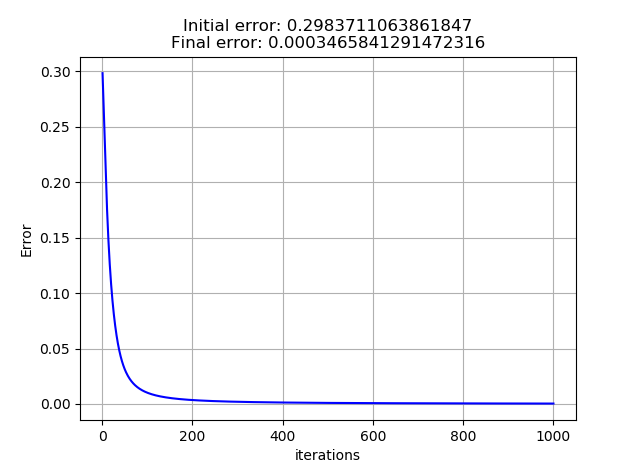
We found gradient of w1. We know its initial value is 0.15. So we can update it as following:

w1 = 0.15 – α\*0.00043857 where α is learning rate. If we set it as we set in output layer 0.5:

w1 = 0.15 – 0.5\*0.00043857 = 0.14978072

I have shown gradient calculation for only w1. We can do the same calculations for hidden layer parameters (w2,w3,w4,b1).

I have created the exact neural network architecture and setup in python. When I ran the code, I get the results below.



Also, after 1000 iterations neural network’s outputs are:

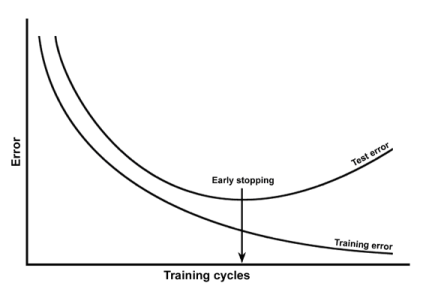
0.02874655 🡪 real output is 0.01

0.97151393 🡪 real output is 0.99

There is one thing left I should mention for training. Is it always good to have zero error on training set after training? Now lets think about the figure below.



As we can see, the rightmost graph perfectly fits to the training set i.e. it has zero error. In fact there are infinitely many ways that we can perfectly fit the training data. ANNs are powerful enough to completely memorize training data so that they can copy desired outputs as their calculated output. So we need a metric for a good training. Here comes “test set”. We know ANNs learn from comparing training outputs and their calculated outputs. It means ANNs see what the training set is. We can introduce another set called test set and we do not show it to ANN during training. By looking at not only training error and training accuracy but also test error and test accuracy, we can figure out when we should stop training. Test error and test accuracy tell us how good our ANN on a set which it never sees before. Hence, it is a measure of how an ANN generalizes a problem.



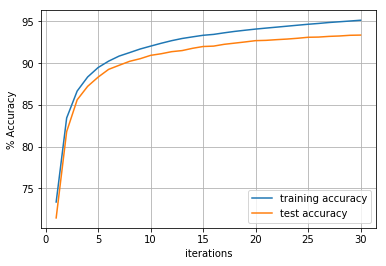
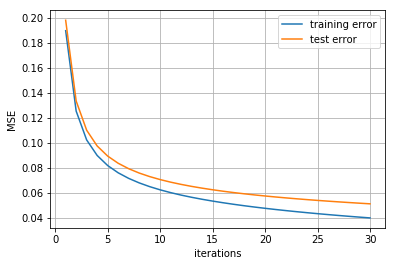
# **ANN Example: Hand-written Digit Recognition**

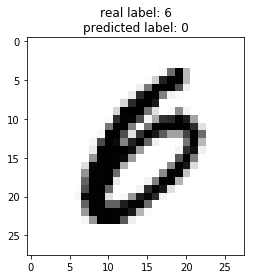
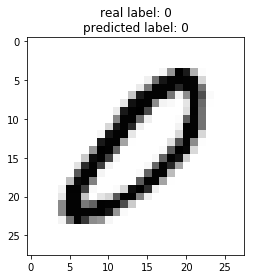
I want to show an example of ANN. I wrote my code in python from scratch and did not use any frameworks for ANNs.

* I used MNIST dataset.
* There are total number f 42000 images and their true label in the dataset.
* All images are in 28 x 28 matris format. Here is an example:



* I randomly split the dataset into 33600 training set and 8400 test set.
* I flattened all images so training set is in the form of 33600 x 784 matris.
* I used 2 hidden layers each of having 500 neurons. (Multi-layer Perceptron)
* Both hidden layers have ReLu activation function.
* I have used an output layer with softmax activation having 10 classes for each digit.
* I used MSE for gradient descent calculations. The results are shown below:





Test Set Samples (correct and wrong predictions)

# **REFERENCES**

1. Haykin, Simon S. Neural Networks and Learning Machines. Pearson, 2016.
2. David Kriesel, 2007, A Brief Introduction to Neural Networks, available at [http://www.dkriesel.com](http://www.dkriesel.com/)
3. Kröse Ben, and Patrick van der. Smagt. An Introduction to Neural Networks. University of Amsterdam, 1996.
4. Rojas Raúl. Neural Networks: A Systematic Introduction. Springer, 2000.
5. Samarasinghe, Sandhya. Neural Networks for Applied Sciences and Engineering: from Fundamentals to Complex Pattern Recognition. Auerbach, 2007.
6. Singh, Vivek. An Intuitive Explanation to Gradient Descent – Vivek Singh – Medium. Augmenting Humanity, 13 Sept. 2017, [medium.com/@viveksingh.heritage/an-introduction-to-gradient-descent-54775b55ba4f](mailto:medium.com/@viveksingh.heritage/an-introduction-to-gradient-descent-54775b55ba4f).
7. Shung, Koo Ping. Gradient Descent: Simply Explained? – Towards Data Science. Towards Data Science, 2 Apr. 2018, towardsdatascience.com/gradient-descent-simply-explained-1d2baa65c757.
8. Mazur, Matt. A Step by Step Backpropagation Example. 21 Nov. 2017, mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/.
9. Moawad, Assaad. Neural Networks and Backpropagation Explained in a Simple Way. Augmenting Humanity, 1 Feb. 2018, medium.com/datathings/neural-networks-and-backpropagation-explained-in-a-simple-way-f540a3611f5e.
10. Bhande, Anup. What Is Underfitting and Overfitting in Machine Learning and How to Deal with It.Augmenting Humanity, 11 Mar. 2018, medium.com/greyatom/what-is-underfitting-and-overfitting-in-machine-learning-and-how-to-deal-with-it-6803a989c76.
11. Artificial Neural Networks and Neural Networks Applications - XenonStack Blog. XenonStack, 30 May 2018, www.xenonstack.com/blog/data-science/artificial-neural-networks-applications-algorithms/.