Perceptron: A Journey towards building Deep Neural Networks

Aravindan T S
Advisor: Santosh Singh

May 14, 2023

Acknowledgement

I would like to begin by expressing my deepest gratitude to my advisor Dr. Santosh Singh for his invaluable guidance and feedback. I am extremely grateful to Ms. Rekha without whom this journey would not have been possible. I thank you for all the guidance you have provided me and all the hours you have spent towards helping me. I am also thankful to my family and friends for their continuous support and understanding when undertaking my research and writing my project.

Contents

1 Introduction

A biological neuron or a nerve cell is the most basic unit of the nervous system. It is an electrically excitable cell designed to transmit information to other nerve cells, muscle, or gland cells. Most neurons have a cell body (or soma), an axon, and dendrites. The cell body contains the nucleus and cytoplasm. The axon is the long thread-like part of the neuron that extends from the cell body and often gives rise to many smaller branches before ending at nerve terminals. Dendrites are membranous tree-like projections extending from the neuron cell body and which receives messages from other neurons. Synapses are the contact points where one neuron communicates with another. The dendrites are covered with synapses formed by the ends of axons from other neurons.

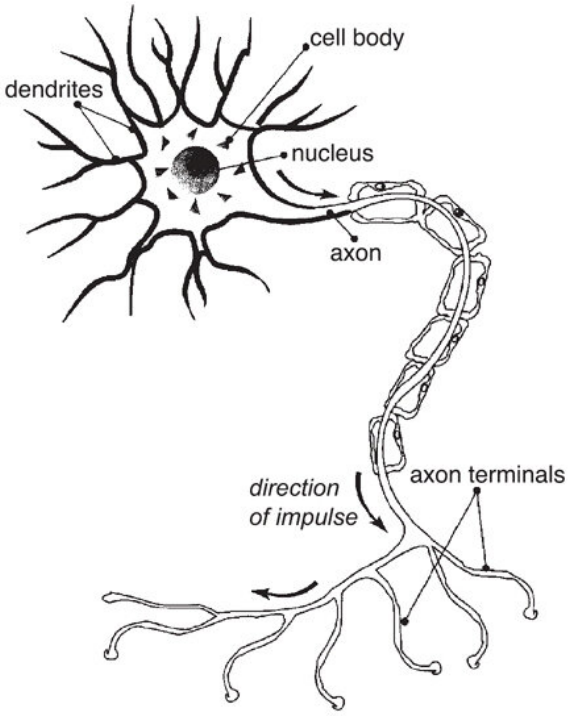


Figure 1: A biological neuron

The initial interest in artificial neural networks was sparked after the introduction of simplified neurons by Warren McCulloch and Walter Pitts in 1943. These neurons were presented as models of biological neurons and as conceptual components for circuits that could perform computational tasks. In a biological neuron, an input signal is received through the dendrites, processed in the soma and passed through the axon and the synapse to the dendrites of another neuron, muscle or gland cell. The McCulloch-Pitts model of a neuron can be divided into 2 parts. The first part receives an input, performs an aggregation and based on this aggregated value, the second part produces an output. This model of a neuron put forward by McCulloh and Pitts is known as a perceptron.

The perceptron was first implemented in 1958 by Frank Rosenblatt. Rosenblatt first described the perceptron in 1957 in his paper titled 'The Perceptron: A Perceiving and Recognizing Automaton'. In this paper, Rosenblatt mentioned that since the advent

of electronic computers and modern servo systems, there was an increased interest in a machine which would be capable of conceptualizing inputs received directly from the physical environment of light, sound, temperature, etcetera rather than requiring a human agent to digest and code the necessary information. A major requirement of such a system is that it must be able to recognize complex patterns of information which are extremely similar or are experimentally related. The system must be able to recognize the same object in different orientations, sizes, colors, or transformations and against different backgrounds. This can be carried out, to a certain extent, by digital procedures on a computer. But it would be hard to design a general analytic program for this without storing a large library of references against which the percept could be compared. If the system is to be economical, the number of functional units in the storage system or memory should be much less than the number of forms or memories to be retained. This last requirement ruled out conventional computer systems of the time. Also, if a memory with millions of patterns stored in it must be scanned sequentially to identify an object, the time required by conventional systems (of the time) would have been excessive. Hence, the proposed system must economize on storage space as well as be able to identify an object without resorting to a sequential search procedure. Rosenblatt then stated that it was feasible to construct an electronic or electromechanical system which would learn to recognize similarities or identities between patterns of optical, electrical, or tonal information in a manner which was closely analogous to the processes of a biological brain. The proposed system would depend on probabilistic principles for its operation and would gain its reliability from properties of statistical measurements obtained from large populations. This system was called a perceptron.

Later in 1969, Marvin Minsky and Seymour Papert, in their book titled 'Perceptrons: An Introduction to Computational Geometry', argued that there were several fundamental problems with Rosenblatt's perceptron model. They stated that there were certain tasks, such as the calculation of topological function of connectedness and the calculation of parity which the model could not solve. Due to its inability to calculate parity, the perceptron could not learn to evaluate the logical function of exclusive-or (XOR). These results led Minsky and Papert to the conclusion that research on perceptrons and their possible extensions were futile. This led to most neural network funding being redirected and researchers leaving the field. Only a few researchers like Teuvo Kohonen, Stephen Grossberg, James Anderson and Kunihiko Fukushima continued their efforts. This, along with other factors, caused the first AI winter which lasted until the early 1980s.

Today, Neural Networks are one of the most widely and extensively researched topics with a myriad of applications like facial recognition, weather forecasting, financial forecasting, medical diagnosis, etc. Even ChatGPT is based on the transformer architecture, which is a type of neural network.

2 Mathematics behind Neural Networks

Artificial Neural Networks function on the principle of learning from data. What is "learning" here? Learning is the process by which the neural network recognizes patterns in the dataset and uses those patterns to predict the output of new data for which the output is unknown. Let us take the example of a credit approval system for a bank. There is no formula by which the bank can pinpoint when the credit should be approved. Instead, what the bank does is it uses historical records of previous customers to figure out a good enough formula for credit approval. The record of each customer has information related to the credit such as annual salary, years in residence, outstanding loans, etc. Using this data, a successful formula for credit approval is formulated which can be used on future applicants. This is known as learning from data.

This system has multiple components. There is the input x (customer information used by the bank to make a credit decision), the unknown target function $f : \chi \to \gamma$ (the ideal formula for credit approval), where χ is the input space(set of all possible inputs x), and γ is the output space (set of all possible outputs which in this case is either yes or no). There is a dataset $\mathcal{D}$ of input-output examples $(x_1, y_1), .., (x_N, y_N)$, where $y_N = f(x_N)$ for $n = 1, ..., N$. There is also a learning algorithm that uses the dataset $\mathcal{D}$ to pick a formula $g : \chi \to \gamma$ that approximates f. The algorithm chooses g from a set of candidate formulas (the hypothesis set $\mathcal{H}$). When the credit approval decision for a new customer is to be made, the bank bases its decision on g (f is unknown). g has to faithfully replicate f in order to achieve a good decision. In order to achieve this, the algorithm chooses g which best matches f on the training examples of previous customers with the hope that it will continue to match f on new customers.

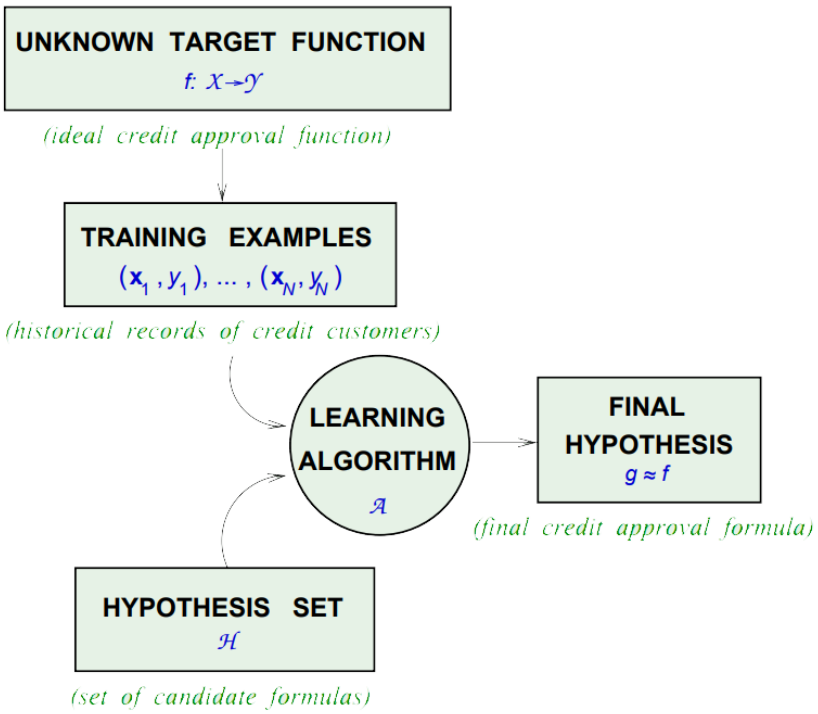


Figure 2: Basic setup of the learning problem

Consider the following model: $\chi = \mathbb{R}^d$ is the input space, where $\mathbb{R}^d$ is the $d-$dimensional Euclidian space, $\gamma = \{+1, -1\}$ is the output space (this denotes a binary yes/no decision). The hypothesis set $\mathcal{H}$ is specified through a functional form that all the hypothesis $h \in \mathcal{H}$ share. The functional form $h(x)$ chosen here gives different weights to the different coordinates of x. The weighted coordinates are then combined to get the weighted sum and this is then compared to a threshold value. For the credit example,

$$\text{Credit is approved if } \sum_{i=1}^{d} w_i x_i > \text{threshold},$$

$$\text{Credit is denied if } \sum_{i=1}^{d} w_i x_i < \text{threshold}.$$

This formula can be written as

$$h(\mathbf{x}) = \text{sign}\left(\sum_{i=1}^{d} w_i x_i + \theta\right),$$

where $x_1, \cdots, x_d$ are the components of the vector $\mathbf{x}$; $h(\mathbf{x}) = +1$ means 'approve credit' and $h(\mathbf{x}) = -1$ means 'deny credit'; $\text{sign}(s) = +1$ if $s > 0$ and $\text{sign}(s) = -1$ if $s < 0$.[1] The weights are $w_1, \cdots, w_d$, and the threshold is determined by the bias term θ.

This model of $\mathcal{H}$ is called the perceptron. Hence, like the biological neuron, the perceptron receives input in the input layer (similar to the dendrites in the biological neuron), processes it, and provides an output in the output layer (similar to the soma, axon and synapse in the biological neuron). The learning algorithm will search $\mathcal{H}$ by looking for weights and bias that perform well on the data set. Some of the weights $w_1, \cdots, w_d$ may end up being negative, corresponding to a negative effect on credit approval. The bias value θ may end up being large or small in order to adjust the threshold value. The optimal choices of weights and bias define the final hypothesis $g \in \mathcal{H}$ that the algorithm produces.

Here, the output of the perceptron is determined by the sign function based on the weighted sum of the inputs and the bias term. Hence, the sign function is the activation function for the above model. The activation function decides whether a neuron should be activated or not by calculating the weighted sum and further adding bias to it. The purpose of the activation function is to introduce non-linearity into the output of a neuron. A neural network without an activation function is essentially just a linear regression model. The activation function does the non-linear transformation to the input making it capable to learn and perform more complex tasks. Other commonly used activation functions are sigmoid, hyperbolid tangent (tanh), Rectified Linear Unit (ReLU) and variations of ReLU. So, a general form of the previous equation would be,

$$h(\mathbf{x}) = \mathcal{F}\left(\sum_{i=1}^{d} w_i x_i + \theta\right)$$

Now, let us consider a model where $d = 2$ and the activation function is the sign function. If the total input is positive, the pattern will be assigned to class +1 and if the total input is negative, the pattern will be assigned to class -1. The separation between the two classes in this case is a straight line, given by the equation:

$$w_1 x_1 + w_2 x_2 + \theta = 0$$

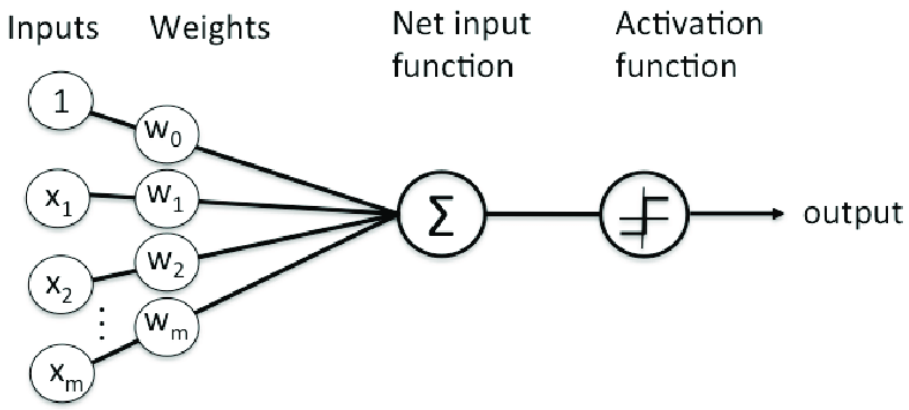


Figure 3: Rosenblatt's perceptron model

The single layer network represents a linear discriminant function.

This equation can also be written as:

$$x_2 = -\frac{w_1}{w_2}x_1 - \frac{\theta}{w_2}$$

The weights determine the slope of the line and the bias determines the offset (how far the line is from the origin). It is important to note that the weights can be plotted in the input space: the weight vector is always perpendicular to the discriminant function.

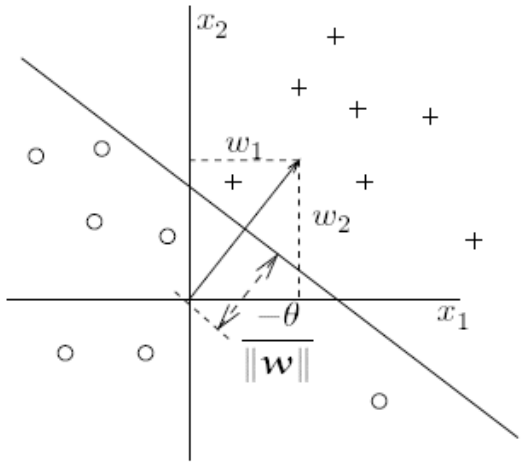


Figure 4: Geometric representation of the discriminant function and the weights

Now we come to the question of determining the ideal weights and bias for the model. This is determined by the *perceptron learning algorithm* (PLA). It is an iterative procedure that adjusts the weights and bias.

$$w_i(t+1) = w_i(t) + \Delta w_i(t),$$
$$\theta(t+1) = \theta(t) + \Delta\theta(t).$$

where t is the index of the current iteration.

2.1 Perceptron learning algorithm

Suppose we have a set of learning samples consisting of an input vector $\mathbf{x}$ and an output $d(\mathbf{x})$. The perceptron learning algorithm is as follows:

1. Start with random weights for the connections;

2. Select an input vector $\mathbf{x}$ from the set of training samples;

3. If $y \neq d(\mathbf{x})$ (the perceptron produces incorrect response), modify all connections according to: $\Delta w_i = d(\mathbf{x})x_i$;

4. Repeat from step 2.

Besides the weights, the bias θ must also be updated. θ is considered as a connection w_0 between the output neuron and a dummy unit with value $x_0 = 1$. With the perceptron learning rule given above, this threshold is updated according to:

$$\Delta\theta = \begin{cases} 0 & \text{if the perceptron responds correctly;} \\ d(x) & \text{otherwise.} \end{cases}$$

2.2 Convergence theorem

For the perceptron learning rule there exists a convergence theorem, which states the following:

Theorem : *If there exists a set of connection weights w^* which is able to perform the transformation $y = d(\mathbf{x})$, the perceptron learning rule will converge to some solution (which may or may not be the same as w^*) in a finite number of steps for any initial choice of the weights.*

Proof: Since the length of the vector w^* does not play a role (because of the sign operation), we take $\|w^*\| = 1$. Because w^* is a correct solution, the value $|w^* \cdot \mathbf{x}|$, where $\cdot$ denotes the inner product, will be greater than 0 or: there exists a $\delta > 0$ such that $|w^* \cdot \mathbf{x}| > \delta$ for all inputs $\mathbf{x}$. Now define $\cos\alpha \equiv w \cdot w^*/\|w\|$. When according to the perceptron learning rule, connection weights are modified at a given input $\mathbf{x}$, we know that $\Delta w = d(\mathbf{x})\mathbf{x}$, and the weight after modification is $w' = w + \Delta w$. From this it follows that:

$$\begin{aligned} w' \cdot w^* &= w \cdot w^* + d(\mathbf{x}) \cdot w^* \cdot \mathbf{x} \\ &= w \cdot w^* + \operatorname{sgn}(w^* \cdot \mathbf{x})\, w^* \cdot \mathbf{x} \\ &> w \cdot w^* + \delta \end{aligned}$$

$$\begin{aligned} \|w'\|^2 &= \|w + d(\mathbf{x})\mathbf{x}\|^2 \\ &= w^2 + 2d(\mathbf{x})w \cdot \mathbf{x} + \mathbf{x}^2 \\ &< w^2 + \mathbf{x}^2 \quad \big(\text{because } d(\mathbf{x}) = -\operatorname{sgn}[w \cdot \mathbf{x}]\big) \\ &= w^2 + M \end{aligned}$$

After t modifications we have:

$$\begin{aligned} w(t) \cdot w^* &> w \cdot w^* + t\delta \\ \|w(t)\|^2 &< w^2 + tM \end{aligned}$$

such that

$$\cos\alpha(t) = \frac{w^* \cdot w(t)}{\|w(t)\|} > \frac{w^* \cdot w + t\delta}{\sqrt{w^2 + tM}}$$

From this follows that $\lim_{t\to\infty} \cos\alpha(t) = \lim_{t\to\infty} \frac{\delta}{\sqrt{M}}\sqrt{t} = \infty$. $\cos\alpha \leq 1$ (definition). This is a contradiction. Hence, there must be an upper limit $t_{\max}$ for t. The system modifies its connections only a limited number of times.

Hence, we can conclude that after $t_{\max}$ modifications of the weights the perceptron is correctly performing the mapping. $t_{\max}$ will be reached when $\cos\alpha = 1$. If we start with connections $w = 0$,

$$t_{\max} = \frac{M}{\delta^2}$$

Note: $w^* \cdot \mathbf{x}$ can be equal to 0 for a w^* which classifies all data correctly. However, another w^* can be found for which the quantity will not be 0.

2.3 Activation functions

A neural network's prediction accuracy is defined by the type of activation function used. The most commonly used activation functions are non-linear activation functions. A neural network works just like a linear regression model where the predicted output is the same as the provided input if an activation function is not defined. Same is the case if a linear activation function is used where the output is similar as the input fed along with some error. A linear activation function's boundary is linear and if such an activation function is used, the network can only adapt to the linear changes of the input. In the real world, a lot of data possess non-linear characteristics. Hence non-linear activation functions are preferred over linear activation functions in a Neural Network. The following are some of the most commonly used activation functions in artificial neural networks:

Binary step function:

The binary step function is defined as,

$$f(x) = 1 \text{ if } x \geq 0$$

$$f(x) = 0 \text{ if } x < 0$$

The derivative of $f(x)$ with respect to x is 0. Binary step functions are generally used for binary classifications.

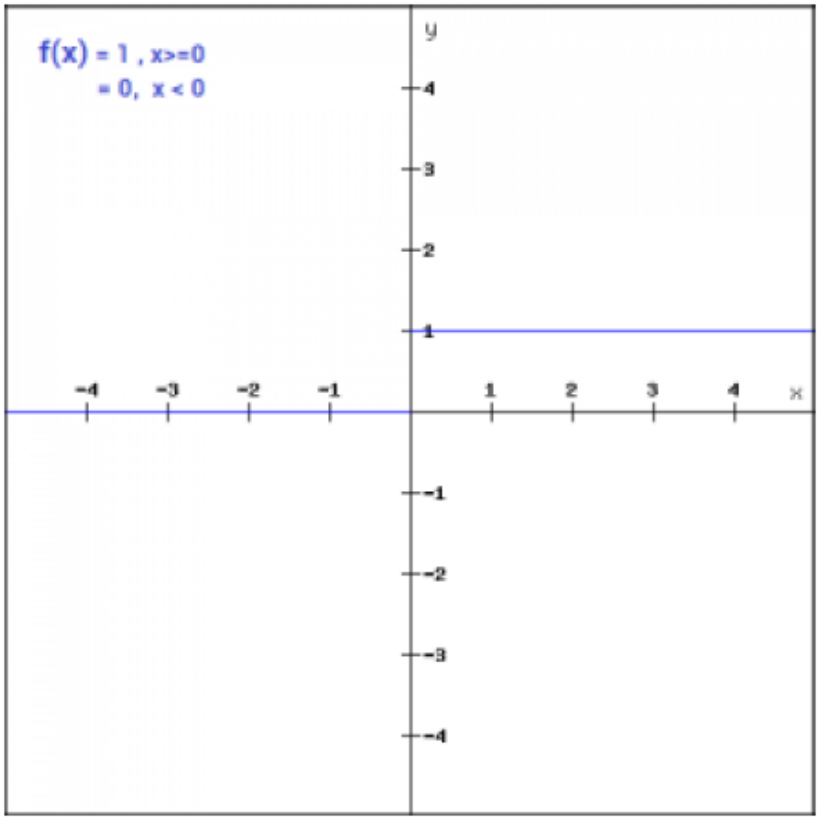


Figure 5: Binary step function

Linear function:

The linear activation function is defined as,

$$f(x) = ax, \text{ where } a \text{ is a constant}$$

The derivative of $f(x)$ with respect to x is a. Since its derivative is not 0 (unlike the binary step function), this activation function can be used in artificial neural networks with more than one layer. This shall be explored further in chapter 3.

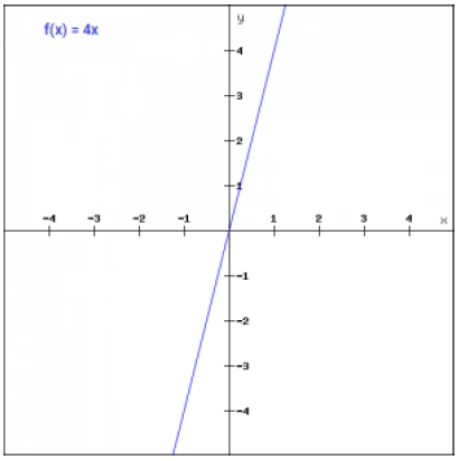


Figure 6: Linear activation function

Sigmoid function:

The sigmoid activation function is defined as,

$$f(x) = \frac{1}{1+e^{-x}}$$

The derivative of $f(x)$ with respect to x is $f(x)(1 - f(x))$. The sigmoid function is the most widely used activation function. It transforms values to the range of 0 to 1.

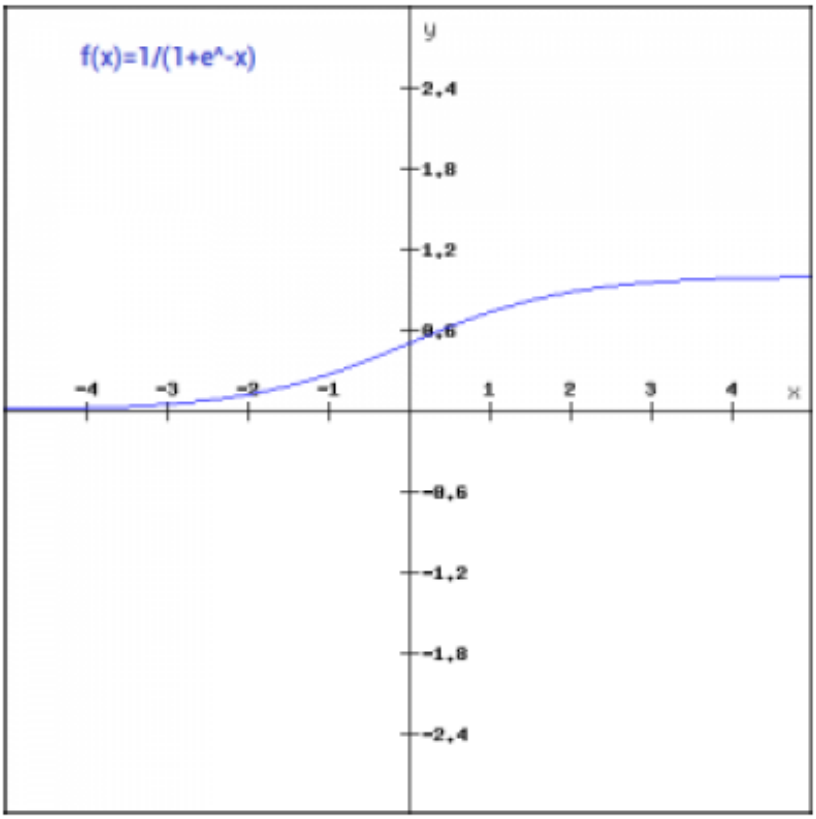


Figure 7: Sigmoid function

Tanh function:

The tanh activation function is defined as,

$$f(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$$

The derivative of $f(x)$ with respect to x is $1 - f^2(x)$. The tanh activation function transforms values to the range of -1 to 1. The tanh function is similar to the sigmoid function but it is symmetric around the origin. This results in different signs of outputs from previous layers which will be fed as input to the next layer.

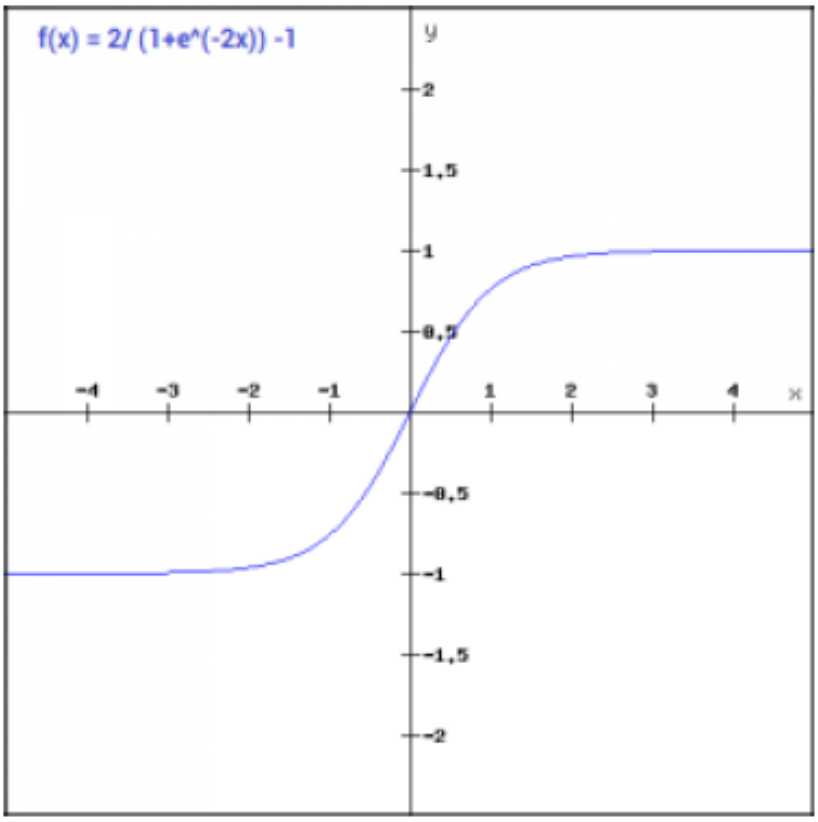


Figure 8: tanh function

ReLU function:

The ReLU activation function is defined as,

$$f(x) = \max(0, x)$$

The derivative of $f(x)$ with respect to x is $f'(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x > 0 \end{cases}$. The derivative is undefined at $x = 0$. ReLU stands for rectified linear unit. ReLU is more efficient than other functions because as all the neurons are not activated at the same time. Only a certain number of neurons are activated at a time.

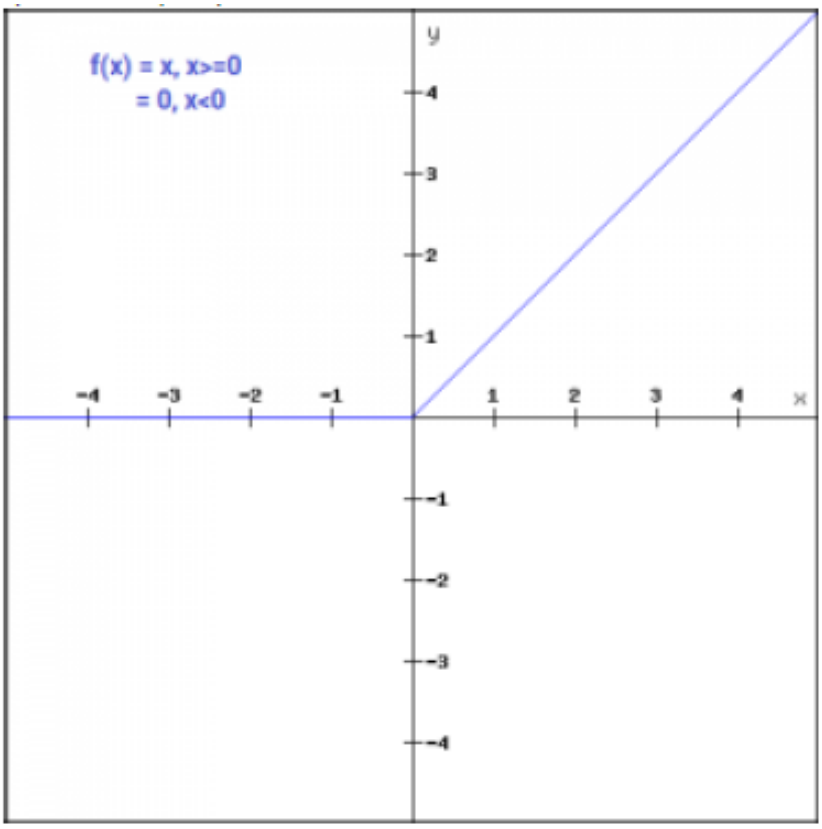


Figure 9: ReLU function

Leaky ReLU function:

The leaky ReLU activation function is defined as,

$$f(x) = \max(0.01 * x, x)$$

The derivative of $f(x)$ with respect to x is $f'(x) = \begin{cases} 0.01 & \text{if } x < 0 \\ 1 & \text{if } x > 0 \end{cases}$. The derivative is undefined at $x = 0$. Leaky ReLU is an improvised version of ReLU function where for negative values of x, instead of defining the value of the function as zero, it is defined as an extremely small linear component of x.

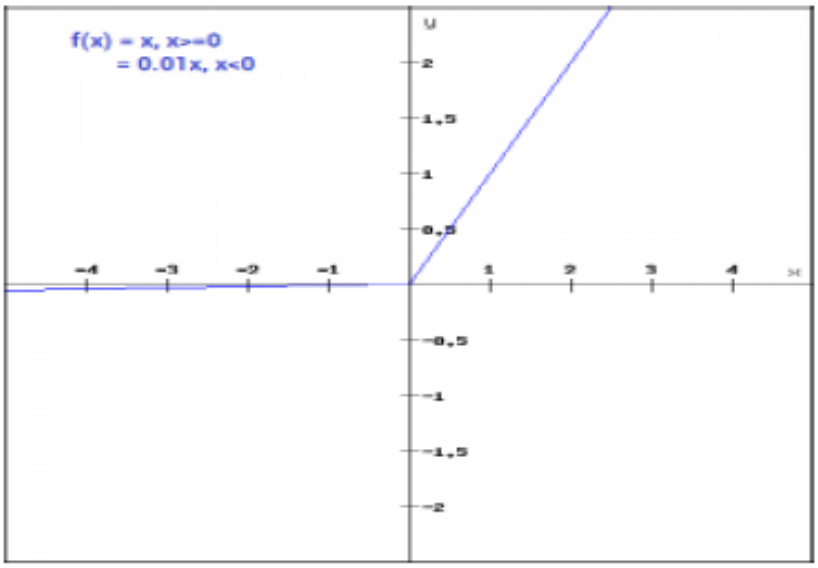


Figure 10: Leaky ReLU function

2.4 Feasibility of learning

Now, we come to the question of whether learning is actually feasible. The unknown target function f is the object of learning. We know the value of f on all the points in the training data $\mathcal{D}$. This does not mean that we have learned f since it does not guarantee that we know the value of f for data outside $\mathcal{D}$. We can only say that we have learned something if the dataset $\mathcal{D}$ tells us anything outside of $\mathcal{D}$. Since f is unknown except inside $\mathcal{D}$, any function that agrees with $\mathcal{D}$ could conceivably be f. Hence, we cannot exclude any function that agrees with $\mathcal{D}$ from being true to f. The whole purpose of learning f is to be able to predict the value of f on unseen data points. The quality of the learning will be determined by how close our prediction is to the true value.

Although it might not be possible to learn the full target function f, it is possible to infer something outside of $\mathcal{D}$ using only $\mathcal{D}$ in a probabilistic way. Consider the following example: A bin contains red and green marbles, possibly infinitely many. The probability of picking a red marble at random is μ and that of picking a green marble is $1-\mu$. We pick a random sample of N independent marbles (with replacement) from this bin and observe the fraction ν of red marbles within the sample. The *Hoeffding Inequality* quantifies the relationship between ν and μ. It states that for any sample size N,

$$\mathbb{P}[|\nu - \mu| > \epsilon] \leq 2e^{-2\epsilon^2 N} \quad \text{for any } \epsilon > 0$$

Here, $\mathbb{P}[.]$ denotes the probability of an event and ϵ is any positive value. The Hoeffding Inequality essentially means that as the sample size N grows, it becomes exponentially unlikely that ν will deviate from μ by more than ϵ. Note that only the size N of the sample affects the bound, not the size of the bin.

Now, let us see how the relates to the learning problem. For the bin model, the unknown was μ whereas for the learning problem, the unknown is the target function $f : \chi \to \gamma$.

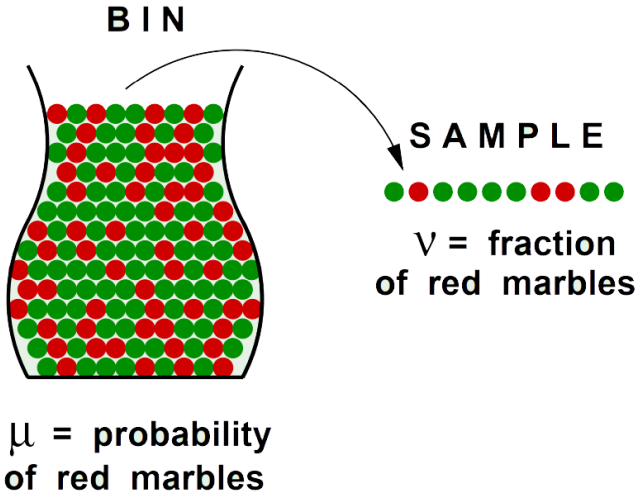


Figure 11: A random sample is picked from a bin of red and green marbles

Take any $h \in \mathcal{H}$ and compare it to f on each point $\mathrm{x} \in \chi$. If $h(\mathrm{x}) = f(\mathrm{x})$, colour the point x green. Else, colour it red. The colour that each point gets is not known to us since f is unknown. However, if we pick x at random according to some probability distribution P over the input space χ, we know that x will be red with some probability μ and green with probability $1-\mu$. The space χ now behaves like the bin. The training examples play the role of samples from the bin.

The error rate within the sample, which corresponds to ν in the bin model, will be called the *in-sample error*,

$$\begin{aligned} E_{\text{in}}(h) = & \text{ (fraction of } \mathcal{D} \text{ where } f \text{ and } h \text{ disagree)} \\ = & \frac{1}{N}\sum_{n=1}^{N}[h(\mathbf{x}_n) \neq f(\mathbf{x}_n)], \end{aligned}$$

where statement $= 1$ if the statement is true, and $= 0$ if the statement is false. In the same way, we define the *out-of-sample error*,

$$E_{\text{out}}(h) = \mathbb{P}[h(\mathrm{x}) \neq f(\mathrm{x})]$$

This corresponds to μ in the bin model. The probability is based on the distribution P over $\mathcal{X}$ which is used to sample the data points x.

Substituting the new notation E_{in} for ν and E_{out} for μ, the Hoeffding Inequality can be rewritten as

$$\mathbb{P}\left[|E_{\text{in}}(h) - E_{\text{out}}(h)| > \epsilon\right] \leq 2e^{-2\epsilon^2 N} \quad \text{for any } \epsilon > 0$$

where N is the number of training examples. The in-sample error E_{in}, just like ν, is a random variable that depends on the sample. The out-of-sample error E_{out}, just like μ, is unknown but not random.

Let us consider a finite hypothesis set $\mathcal{H}$ instead of just one hypothesis h,

$$\mathcal{H} = \{h_1, h_2, \cdots, h_M\}$$

We can construct a bin equivalent in this case by having M bins. Each bin still represents the input space $\mathcal{X}$, with the red marbles in the mth bin corresponding to the points $\mathrm{x} \in \mathcal{X}$ where $h_m(\mathrm{x}) \neq f(\mathrm{x})$. The probability of red marbles in the mth bin is $E_{\text{out}}\ (h_m)$ and the fraction of red marbles in the mth sample is $E_{\text{in}}\ (h_m)$, for $m = 1, \cdots, M$. Although the Hoeffding Inequality still applies to each bin individually, the situation becomes more complicated when we consider all the bins simultaneously. This is because the hypothesis h has to be fixed before the dataset is generated in order for the bound to be valid. If h is changed after you generate the data set, the assumptions that are needed to prove the Hoeffding Inequality no longer hold. With multiple hypotheses in $\mathcal{H}$, the learning algorithm picks the final hypothesis g based on $\mathcal{D}$, i.e. after generating the data set. The statement we would like to make is that

$$\text{"P } [|E_{\text{in}}(g) - E_{\text{out}}\ (g)| > \epsilon] \text{ is small"}$$

for the final hypothesis g.

Which hypothesis is selected to be g depends on the data. So, we cannot just plug in g for h in the Hoeffding inequality. So, we try to bound $\mathbb{P}\left[|E_{\text{in}}\ (g) - E_{\text{out}}\ (g)| > \epsilon\right]$ in a way that does not depend on which g the learning algorithm picks. Since g has to be one of the h_m 's regardless of the algorithm and the sample, it is always true that

$$\begin{aligned} \text{" } |E_{\text{in}}\ (g) - E_{\text{out}}\ (g)| > \epsilon\text{"} \Longrightarrow \text{" } & |E_{\text{in}}\ (h_1) - E_{\text{out}}\ (h_1)| > \epsilon \\ & \text{or } |E_{\text{in}}\ (h_2) - E_{\text{out}}\ (h_2)| > \epsilon \\ & \cdots \\ & \text{or } |E_{\text{in}}\ (h_M) - E_{\text{out}}\ (h_M)| > \epsilon\text{"} \end{aligned}$$

We now apply two basic rules in probability:

1) If $\mathcal{B}_1 \Longrightarrow \mathcal{B}_2$, then $\mathbb{P}\left[\mathcal{B}_1\right] \leq \mathbb{P}\left[\mathcal{B}_2\right]$

2) if $\mathcal{B}_1, \mathcal{B}_2, \cdots, \mathcal{B}_M$ are events, then

$$\mathbb{P}\left[\mathcal{B}_1 \text{ or } \mathcal{B}_2 \text{ or } \cdots \text{ or } \mathcal{B}_M\right] \leq \mathbb{P}\left[\mathcal{B}_1\right] + \mathbb{P}\left[\mathcal{B}_2\right] + \cdots + \mathbb{P}\left[\mathcal{B}_M\right]$$

The second rule is known as the union bound. Putting the two rules together, We get

$$\begin{aligned} \mathbb{P}\left[|E_{\text{in}}\ (g) - E_{\text{out}}\ (g)| > \epsilon\right] \leq \mathbb{P}\big[& \quad |E_{\text{in}}\ (h_1) - E_{\text{out}}\ (h_1)| > \epsilon \\ & \text{or } |E_{\text{in}}\ (h_2) - E_{\text{out}}\ (h_2)| > \epsilon \\ & \ldots \\ & \text{or } |E_{\text{in}}\left(h_M\right) - E_{\text{out}}\ (h_M)| > \epsilon\big] \\ & \leq \sum_{m=1}^{M} \mathbb{P}\left[|E_{\text{in}}\ (h_m) \quad E_{\text{out}}\ (h_m)| > \epsilon\right]. \end{aligned}$$

Applying the Hoeffding Inequality to the M terms one at a time, we can bound each term in the sum by $2e^{-2\epsilon^2 N}$. Hence,

$$\mathbb{P}\left[|E_{\text{in}}\ (g) - E_{\text{out}}\ (g)| > \epsilon\right] \leq 2Me^{-2\epsilon^2 N}$$

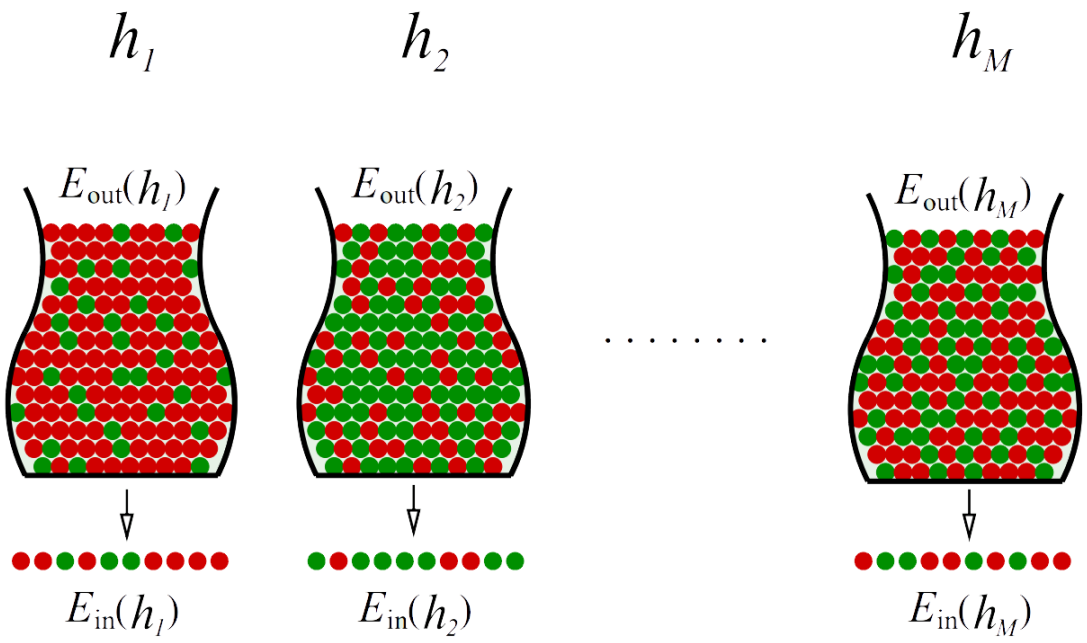


Figure 12: Multiple bins depicting the learning problem with M hypotheses

We are trying to simultaneously approximate all $E_{\text{out}}\ (h_m)$'s by the corresponding $E_{\text{in}}\ (h_m)$'s. This lets the algorithm choose any hypothesis based on E_{in} and expect that the corresponding E_{out} to uniformly follow suit.

We can see that the dataset $\mathcal{D}$ does tell us something outside $\mathcal{D}$ in a probabilistic sense. Hence, we can conclude that learning is indeed feasible.

The model produces a hypothesis g to approximate the unknown target function f. If learning is successful, then g should approximate f well, which means $E_{\text{out}}\ (g) \approx 0$. However, this is not what we get from the probabilistic analysis. What we get instead is $E_{\text{out}}\ (g) \approx E_{\text{in}}\ (g)$. We still have to make $E_{\text{in}}\ (g) \approx 0$ in order to conclude that $E_{\text{out}}\ (g) \approx 0$.

We cannot guarantee that we will find a hypothesis that achieves $E_{\text{in}}(g) \approx 0$, but at least we will know if we find it. If $E_{\text{in}}(g) \approx 0$, by the Hoeffding Inequality, $E_{\text{out}}(g) \approx 0$.

Hence, the feasibility of learning is split into 2 parts:

1. Can you make sure that $E_{\text{out}}\ (g)$ is close enough to $E_{\text{in}}\ (g)$?

2. Can you make $E_{\text{in}}\ (g)$ small enough?

The Hoeffding Inequality helps to answer the first question. The second question is answered only after the learning algorithm on the actual data and see how small we can get $E_{\text{in}}\ (g)$ to be.

The complexity of $\mathcal{H}$: If the number of hypotheses M increases, the probability of $E_{\text{in}}\ (g)$ being a poor estimator of $E_{\text{out}}\ (g)$ is higher according to the Hoeffding Inequality. M can be thought of as a measure of the 'complexity' of the hypothesis set $\mathcal{H}$ that we use. If we want an affirmative answer to the first question, we need to keep the

complexity of $\mathcal{H}$ in check. However, if we want an affirmative answer to the second question, we stand a better chance if $\mathcal{H}$ is more complex, since g has to come from $\mathcal{H}$. So, a more complex $\mathcal{H}$ gives us more flexibility in finding some g that fits the data well, leading to small $E_{\text{in}}(g)$. This tradeoff in the complexity of $\mathcal{H}$ is approximation-generalization tradeoff.

3 Neural Network Architecture

We have already seen the architecture of artificial neural networks with a single layer in chapter 2 (Rosenblatt's perceptron model). Even though the single layer perceptron model technically has 2 layers (input and output layer), it is a referred to as a single layer neural network since we generally do not count the input layer while talking about the number layers in an ANN model. An ANN model can have 3 types of layers: an input layer, an output layer and hidden layers. An ANN with 2 or more layers would have one or more hidden layers. The input layer receives the input signal to be processed. The required task such as prediction and classification is performed by the output layer. An arbitrary number of hidden layers that are placed in between the input and output layer are the true computational engine of the ANN.

The single layer perceptron model is a feedforward neural network. A feedforward Artificial Neural Network is a type of neural network in which the information flows in only one direction, from the input layer, through one or more hidden layers, to the output layer, without any feedback loops. It is called "feedforward" because the data only moves forward through the network and never loops back on itself.

Now, let us talk about the multilayer perceptron model (MLP). The multilayer perceptron model is an ANN with one or more hidden layers. Similar to a feedforward network, in a MLP the data flows in the forward direction from input to output layer. The neurons in the MLP are trained with the backpropagation learning algorithm. Backpropagation is a process involved in training a neural network. It involves taking the error rate of a forward propagation and feeding this loss backward through the neural network layers to fine-tune the weights. In an MLP, every node in one layer is connected to every other node in the next layer. We make the network deeper by increasing the number of hidden layers. Each layer in an MLP can have multiple nodes. Each node in the hidden layer takes the weighted sum of its inputs, and passes it through a non-linear activation function. This is the output of the node, which then becomes the input of another node in the next layer. The signal flows from left to right, and the final output is calculated by performing this procedure for all the nodes. Training this deep neural network means learning the weights associated with all the edges.

3.1 Feedforward propagation

Consider the MLP in figure 13. Let i be the index of the layer. Then, the number of nodes in the ith layer is denoted by $n^{[i]}$. The input layer is denoted by the index $i = 0$, the first hidden layer is denoted by the index $i = 1$ and so on. For this MLP, $n^{[0]} = 3$ (input layer), $n^{[1]} = 4$ (first hidden layer), $n^{[2]} = 4$ (second hidden layer) and $n^{[3]} = 1$ (output layer). Each node acts in a similar way to the single layer perceptron. It calculates the weighted sum of its inputs, applies an activation function to it and passes it as input to all the nodes in the next layer (only the output layer does not pass the computed value to any node).

The number of nodes in the input layer of an MLP denotes the number number of features of the input data. For example, the input data of the MLP in figure 13 has 3

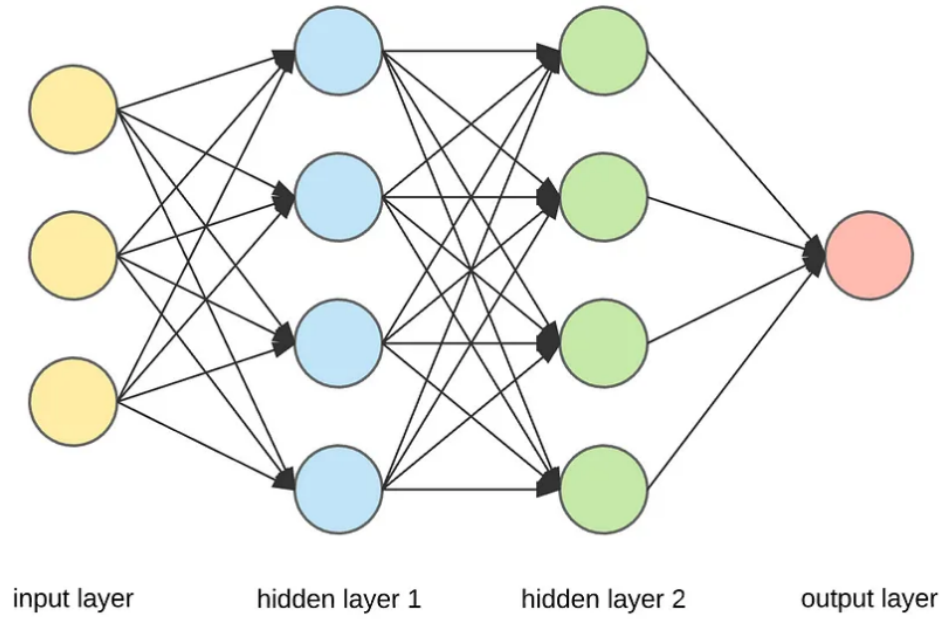


Figure 13: A multilayer perceptron model

features. If j is the index of the feature, the jth feature of a data point X is represented as x_j. It can be represented in vector form as,

$$X = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix},$$

where n is the number of features and X is a data point from the input data. The size of the matrix is $n^{[0]} \times 1$. For the given MLP (figure 13),

$$X = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix},$$

Each node in the output and hidden layers calculates the weighted sum of its inputs and applies an activation function to it. This is the output of that particular node. Hence, if a hidden or output layer has n nodes, its output can be represented in vector form as,

$$A^{[i]} = \begin{bmatrix} A_1^{[i]} \\ A_2^{[i]} \\ \vdots \\ A_n^{[i]} \end{bmatrix},$$

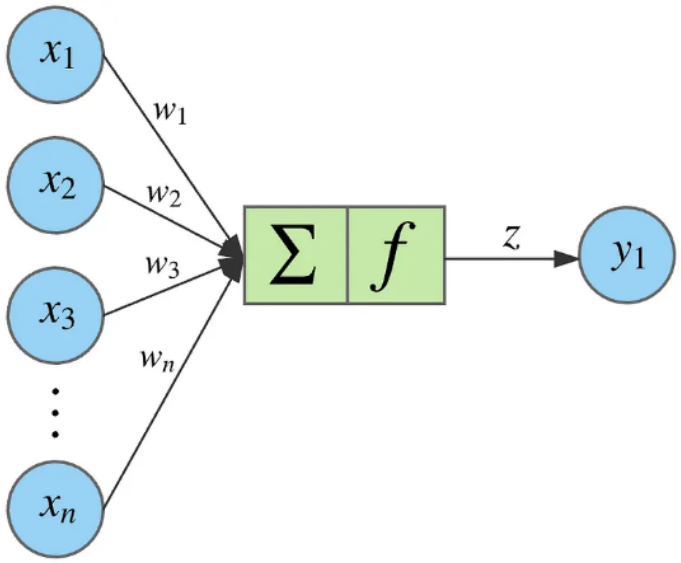


Figure 14: Zoomed in image of any hidden or output node

where n is the number of nodes in the layer, i is the index of the layer and $A^{[i]}$ is the output of the layer. Since the output of a node is an activation function applied to the weighted sum, it can be represented as,

$$A^{[i]} = g^{[i]}\mathcal{Z}^{[i]}$$

where $g^{[i]}$ is the activation function of the ith layer and,

$$\mathcal{Z}^{[i]} = \begin{bmatrix} \mathcal{Z}_1^{[i]} \\ \mathcal{Z}_2^{[i]} \\ \vdots \\ \mathcal{Z}_n^{[i]} \end{bmatrix},$$

where $\mathcal{Z}^{[i]}$ is the total weighted sum of the ith layer and $\mathcal{Z}_j^{[i]}$ is the weighted sum of the jth node of the ith layer. $\mathcal{Z}^{[i]}$ is calculated as,

$$\mathcal{Z}^{[i]} = W^{[i]}X + b^{[i]}$$

where $W^{[i]}$ is a matrix with the weights corresponding to the ith layer and $b^{[i]}$ is a column matrix with bias values corresponding to the ith layer. $A^{[i]}$, $\mathcal{Z}^{[i]}$ and $b^{[i]}$ are vectors of size $n^{[i]} \times 1$ and $W^{[i]}$ is a vector of size $n^{[i]} \times \mathrm{n}^{[i-1]}$. So, if a layer i has j nodes and layer $i-1$ has k nodes, the k dimensional input is transformed into a j dimensional output in the ith layer.

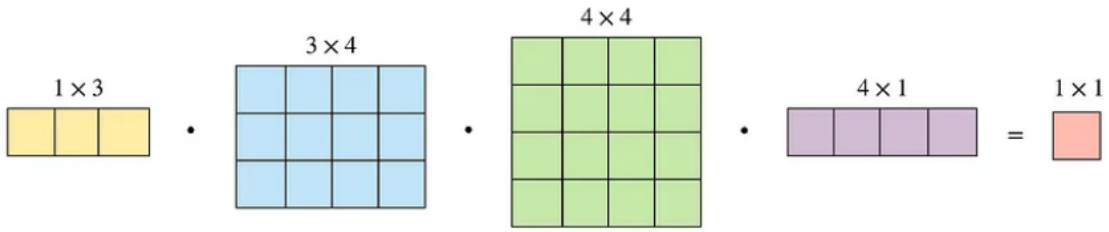
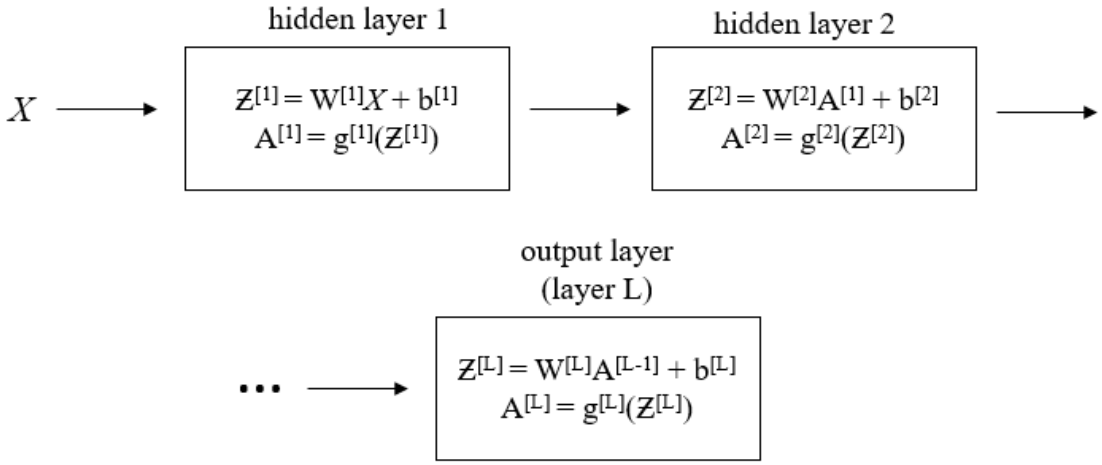


Figure 15: Input, output and weight matrices of the MLP from figure 13

Consider an MLP with L layers, The feedforward algorithm for this MLP is as follows:



$A^{[L]}$ is the final predicted value of the MLP. We have looked at the feedforward part of the MLP algorithm. Now, let us take a look at backpropagation.

3.2 Backpropagation

Backpropagation is the process by which the error rate of the feedforward propagation is calculated and fed back through the neural network to update the weights. The error of the feedforward propagation calculated is the called the loss. A loss function is used to calculate the loss. The most commonly used loss function for binary classification in ANNs is the binary cross-entropy/log loss function which is defined for binary classification as:

$$\mathcal{L}(y, \hat{y}) = -y \log \hat{y} - (1 - y) \log(1 - \hat{y})$$

where $\mathcal{L}(y, \hat{y})$ is the binary log loss function, y is the actual output and $\hat{y}$ is the predicted output. If $y = 1$, then the loss becomes $-\log \hat{y}$, which is minimized when the predicted probability $\hat{y}$ is close to 1. If $y = 0$, then the loss becomes $-\log(1 - \hat{y})$, which is minimized when the predicted probability $\hat{y}$ is close to 0. Therefore, the binary log loss function encourages the network to predict high probabilities for the positive class when the actual output is 1, and low probabilities for the positive class when the actual output is 0.

Now, let us see how this loss is propagated back through the layers to update the weights and bias. The formula to update the weights and bias is as follows:

$$w^{[i]}(t+1) = w^{[i]}(t) + A^{[i]}(t)dw^{[i]}(t)$$

$$b^{[i]}(t+1) = b^{[i]}(t) + A^{[i]}(t)db^{[i]}(t)$$

where i is the index of the layer, t is the number of the iteration, A is the activated weighted sum, w is the weights, b is the bias and dw and db are the derivatives of the loss function with respect to the weights and bias respectively. The unknown quantities here as of now are dw and db. Hence we have to calculate these for each layer.

Consider an MLP with N layers which has the sigmoid function as the activation function in the output layer and the ReLU function as the activation function of all the hidden layers. The sigmoid function and its derivative are defined as follows:

$$g(x) = \frac{1}{1+e^{-x}}$$

$$g'(x) = g(x)(1-g(x))$$

The ReLU function and its derivative are defined as follows:

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases}$$

$$f'(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases}$$

The loss is calculated after the activated weighted sum is calculated for the Nth layer. The activated weighted sum of the final layer is the predicted output of the MLP. Hence, the loss is calculated as,

$$\mathcal{L} = -y \log A^{[N]} - (1-y)\log(1-A^{[N]})$$

Now we have to calculate $\frac{d\mathcal{L}}{dw^{[N]}}$ and $\frac{d\mathcal{L}}{db^{[N]}}$.

$$\frac{d\mathcal{L}}{dw^{[N]}} = \frac{d\mathcal{L}}{d\mathcal{Z}^{[N]}} \cdot \frac{d\mathcal{Z}^{[N]}}{dw^{[N]}}$$

$$\frac{d\mathcal{L}}{db^{[N]}} = \frac{d\mathcal{L}}{d\mathcal{Z}^{[N]}} \cdot \frac{d\mathcal{Z}^{[N]}}{db^{[N]}}$$

$\implies$

$$\frac{d\mathcal{L}}{dw^{[N]}} = \frac{d\mathcal{L}}{dA^{[N]}} \cdot \frac{dA^{[N]}}{d\mathcal{Z}^{[N]}} \cdot \frac{d\mathcal{Z}^{[N]}}{dw^{[N]}}$$

$$\frac{d\mathcal{L}}{db^{[N]}} = \frac{d\mathcal{L}}{dA^{[N]}} \cdot \frac{dA^{[N]}}{d\mathcal{Z}^{[N]}} \cdot \frac{d\mathcal{Z}^{[N]}}{db^{[N]}}$$

$\implies$

$$\frac{d\mathcal{L}}{dw^{[N]}} = (\frac{-y}{A^{[N]}} + \frac{1-y}{1-A^{[N]}})(A^{[N]}(1-A^{[N]}))A^{[N-1]} = (A^{[N]}-y)A^{[N-1]} = d\mathcal{Z}^{[N]}A^{[N-1]}$$

$$\frac{d\mathcal{L}}{db^{[N]}} = (\frac{-y}{A^{[N]}} + \frac{1-y}{1-A^{[N]}})(A^{[N]}(1-A^{[N]})) = (A^{[N]} - y) = d\mathcal{Z}^{[N]}$$

Now that we have calculated $\frac{d\mathcal{L}}{dw^{[N]}}$ and $\frac{d\mathcal{L}}{db^{[N]}}$, we can update the weights and bias accordingly for the Nth layer. The next step is to calculate $\frac{d\mathcal{L}}{dw^{[N-1]}}$ and $\frac{d\mathcal{L}}{db^{[N-1]}}$.

$$\frac{d\mathcal{L}}{dw^{[N-1]}} = \frac{d\mathcal{L}}{d\mathcal{Z}^{[N]}} \cdot \frac{d\mathcal{Z}^{[N]}}{dw^{[N-1]}}$$

$$\frac{d\mathcal{L}}{db^{[N-1]}} = \frac{d\mathcal{L}}{d\mathcal{Z}^{[N]}} \cdot \frac{d\mathcal{Z}^{[N]}}{db^{[N-1]}}$$

$\implies$

$$\frac{d\mathcal{L}}{dw^{[N-1]}} = \frac{d\mathcal{L}}{d\mathcal{Z}^{[N]}} \cdot \frac{d\mathcal{Z}^{[N]}}{dA^{[N-1]}} \cdot \frac{dA^{[N-1]}}{d\mathcal{Z}^{[N-1]}} \cdot \frac{d\mathcal{Z}^{[N-1]}}{dw^{[N-1]}}$$

$$\frac{d\mathcal{L}}{db^{[N-1]}} = \frac{d\mathcal{L}}{d\mathcal{Z}^{[N]}} \cdot \frac{d\mathcal{Z}^{[N]}}{dA^{[N-1]}} \cdot \frac{dA^{[N-1]}}{d\mathcal{Z}^{[N-1]}} \cdot \frac{d\mathcal{Z}^{[N-1]}}{db^{[N-1]}}$$

$\implies$

$$\frac{d\mathcal{L}}{dw^{[N-1]}} = d\mathcal{Z}^{[N-1]} A^{[N-2]}$$

$$\frac{d\mathcal{L}}{db^{[N-1]}} = d\mathcal{Z}^{[N-1]}$$

Here, since the activation function is the ReLU function,

$$d\mathcal{Z}^{[N-1]} = \begin{cases} \frac{d\mathcal{L}}{d\mathcal{Z}^{[N]}} \cdot \frac{d\mathcal{Z}^{[N]}}{dA^{[N-1]}} & \text{if } \mathcal{Z}^{[N-1]} > 0 \\ 0 & \text{if } \mathcal{Z}^{[N-1]} \leq 0 \end{cases} = \begin{cases} (A^{[N]} - y)w^{[N]} & \text{if } \mathcal{Z}^{[N-1]} > 0 \\ 0 & \text{if } \mathcal{Z}^{[N-1]} \leq 0 \end{cases}$$

This calculation similarly continues for all the hidden layers. For the first hidden layer, the $A[N-1]$ from the above equation for $\frac{d\mathcal{L}}{dw^{[N-1]}}$ corresponding to it would be the input data X. Hence,

$$\frac{d\mathcal{L}}{dw^{[1]}} = d\mathcal{Z}^{[1]} X$$

$$\frac{d\mathcal{L}}{db^{[1]}} = d\mathcal{Z}^{[1]}$$

After all the weights and bias are updated for each layer, the feedforward propagation is again initiated and the loss function is calculated. Then the error is again backpropagated. This process is repeated until the required weights and bias are found that can perfectly classify the data or that can reduce the error below a certain threshold or any other condition provided by the user.

4 Implementation

Now, let us take a look at the implementation of the single and multilayer perceptron algorithms.

4.1 Single layer perceptron model

For the single layer perceptron model, I have divided the dataset into 5 parts and created 5 single layer perceptron models with each having a different activation function. The pseudocode for the single layer perceptron is as follows:

```
Z ⟵ dataset
X ⟵ input data
Y ⟵ output data
X_train ⟵ 80% of X (dataset is shuffled)
X_test ⟵ remaining 20% of X
Y_train ⟵ 80% of Y
Y_test ⟵ remaining 20% of Y
X_train ⟵ transpose of X_train
X_test ⟵ transpose of X_test
Y_train ⟵ transpose of Y_train
Y_test ⟵ transpose of Y_test
X1_train ⟵ array with X_train divided into 5 parts
X1_test ⟵ array with X_test divided into 5 parts
Y1_train ⟵ array with Y_train divided into 5 parts
Y1_test ⟵ array with Y_test divided into 5 parts


Function perceptron_model_sign()
    Pass in: array x representing an input data point
    Pass in: array w representing weights
    weighted_sum ⟵ w·x
    y ⟵ sign of weighted_sum
    Pass out: y
Endfunction

Function perceptron_model_sigmoid()
    Pass in: array x representing an input data point
    Pass in: array w representing weights
    weighted_sum ⟵ w·x
    y ⟵ sigmoid of weighted_sum
    If y≥0.5
        Pass out: 1
    Else
        Pass out: -1
    Endif
Endfunction

Function perceptron_model_tanh()
```

```
        Pass in: array x representing an input data point
        Pass in: array w representing weights
        weighted_sum ⟵ w·x
        Pass out: sign of tanh of weighted_sum
Endfunction

Function perceptron_model_relu()
        Pass in: array x representing an input data point
        Pass in: array w representing weights
        weighted_sum ⟵ w·x
        y ⟵ max of 0 or weighted_sum
        If y = 0
                Pass out: -1
        Else
                Pass out: -1
        Endif
Endfunction

Function perceptron_model_leakyrelu()
        Pass in: array x representing an input data point
        Pass in: array w representing weights
        weighted_sum ⟵ w·x
        If weighted_sum ≥ 0
                y ⟵ weighted_sum
        Else
                y ⟵ 0.01 × weighted_sum
        Endif
        Pass out: sign of y
Endfunction

Function model_training()
        Pass in: array X representing input data
        Pass in: array Y representing output data
        Pass in: count representing the index of the part of the data (data was previously
        divided into 5 parts)
        w ⟵ array of size 25 with random decimal values (24 (number of features in
        the data) + 1(bias))
        l ⟵ size of Y
        y_p ⟵ array of size l with 1 as every entry
        For iter = 0 to 10000
                For i = 0 to l
                        If count = 0
                                y_p[i] ⟵ Call: perceptron_model_sign(Pass: ith column of X,
                                Pass: w)
                        Endif
                        Else if count = 1
                                y_p[i] ⟵ Call: perceptron_model_sigmoid(Pass: ith column of
                                X, Pass: w)
```

Endelseif
Else if count = 2
y_p[i] ⟵ Call: perceptron_model_tanh(Pass: ith column of X, Pass: w)
Endelseif
Else if count = 3
y_p[i] ⟵ Call: perceptron_model_relu(Pass: ith column of X, Pass: w)
Endelseif
Else if count = 4
y_p[i] ⟵ Call: perceptron_model_leakyrelu(Pass: ith column of X, Pass: w)
Endelseif
Else if y_p[i] ≠ Y[i]
w ⟵ w - (y_p[i] - Y[i]) × ith column of X
Endelseif
Endfor
Endfor
Plot: Confusion matrix with actual output Y and predicted output y_p
Pass out: w
Endfunction

Function model_testing()
Pass in: array X representing input data
Pass in: array Y representing output data
Pass in: array W representing the weights
Pass in: count representing the index of the part of the data (data was previously divided into 5 parts)
l ⟵ size of Y
y_p ⟵ array of size l with 1 as every entry
For i = 0 to l
If count = 0
y_p[i] ⟵ Call: perceptron_model_sign(Pass: ith column of X, Pass: w)
Endif
Else if count = 1
y_p[i] ⟵ Call: perceptron_model_sigmoid(Pass: ith column of X, Pass: w)
Endelseif
Else if count = 2
y_p[i] ⟵ Call: perceptron_model_tanh(Pass: ith column of X, Pass: w)
Endelseif
Else if count = 3
y_p[i] ⟵ Call: perceptron_model_relu(Pass: ith column of X, Pass: w)
Endelseif

Else if count = 4
y_p[i] ⟵ Call: perceptron_model_leakyrelu(Pass: ith column of X, Pass: w)
Endelseif
Endfor
Plot: Confusion matrix with actual output Y and predicted output y_p
Endfunction

For i = 0 to 4
optimal_weight ⟵ Call: model_training(Pass: X1_train[i], Pass: Y1_train[i], Pass: i)
Print optimal_weight
Call: model_testing(Pass: X1_test[i], Pass: Y1_test[i], Pass: optimal_weight, Pass: i)

4.2 Neural Network with one hidden layer

For the nerual network with a single hidden layer, I have used the tanh activation function for the hidden layer and the sigmoid activation function for the output layer. The pseudocode for this is given below.

import dataset
X ⟵ input data
Y ⟵ output data
X_train ⟵ 80% of X (dataset is shuffled)
X_test ⟵ remaining 20% of X
Y_train ⟵ 80% of Y
Y_test ⟵ remaining 20% of Y
X ⟵ transpose of X
X_train ⟵ transpose of X_train
X_test ⟵ transpose of X_test

Function layer_size()
Pass in: array X representing input data
Pass in: array y representing output data
n0 ⟵ number of features of input data
n1 ⟵ number of nodes in the output layer (1)
Pass out: n0
Pass out: n1
Endfunction

Function initialization_parameters()
Pass in: array n0 representing the number of nodes in the input layer
Pass in: array n1 representing the number of nodes in the hidden layer
Pass in: array n2 representing the number of nodes in the output layer
W1 ⟵ random matrix of size n1 × n0 * 0.01
b1 ⟵ random matrix of size n1 × 1

W2 ⟵ random matrix of size n2 × n1 * 0.01
b2 ⟵ random matrix of size n2 × 1
parameters ⟵ array with entries W1, b1, W2 and b2
Pass out: parameters
Endfunction

Function forward_propagation()
Pass in: array X representing input data
Pass in: array parameters which contains the weights and bias
W1 ⟵ W1 from parameters
b1 ⟵ b1 from parameters
W2 ⟵ W2 from parameters
b2 ⟵ b2 from parameters
Z1 ⟵ W1·X + b1
A1 ⟵ tanh(Z1)
Z2 ⟵ W2·A1 + b2
A2 ⟵ sigmoid of Z2
result ⟵ array with entries Z1, A1, Z2 and A2
Pass out: A2
Pass out: result
Endfunction

Function total_cost()
Pass in: A2 which is the output of layer 2
Pass in: y which is the actual output
m ⟵ size of y
logprobs ⟵ y logA2 + (1-y) log(1-A2)
cost ⟵ - (sum of elements in logprobs)/m
Pass out: cost
Endfunction

Function backward_propagation()
Pass in: array parameters with entries W1 (weights of hidden layer), b1 (bias of hidden layer), W2 (weights of output layer), b2 (bias of output layer)
Pass in: array result with entries Z1 (weighted sum of hidden layer), A1 (Z1 with the activation function of the hidden layer applied to it), Z2 (weighted sum of output layer), A2 (Z2 with the activation function of the output layer applied to it)
Pass in: array X representing input data
Pass in: array y representing actual output
m ⟵ size of y
W1 ⟵ W1 from parameters
b1 ⟵ b1 from parameters
W2 ⟵ W2 from parameters
b2 ⟵ b2 from parameters
Z1 ⟵ Z1 from result
A1 ⟵ A1 from result
Z2 ⟵ Z2 from result

A2 ⟵ A2 from result
dZ2 ⟵ A2 - y
dW2 ⟵ (1/m)(dZ2 · $A1^T$)
db2 ⟵ (1/m)(sum of elements of dZ2)
dZ1 ⟵ ($W2^T$ · dZ2)(1 - $A1^2$)
dW1 ⟵ (1/m)(dZ1 · X^T)
db1 ⟵ (1/m)(sum of elements of dZ1)
gradient ⟵ array with entries dW1, db1, dW2 and db2
Pass out: gradient
Endfunction

Function update_parameters()
Pass in: array parameters with entries W1 (weights of hidden layer), b1 (bias of hidden layer), W2 (weights of output layer), b2 (bias of output layer)
Pass in: array gradient with entries dW1 (derivative of W1 with respect to the loss function), db1 (derivative of b1 with respect to the loss function), dW2 (derivative of W2 with respect to the loss function), db2 (derivative of b2 with respect to the loss function)
Pass in: learning_rate which contains the learning rate
W1 ⟵ W1 from parameters
b1 ⟵ b1 from parameters
W2 ⟵ W2 from parameters
b2 ⟵ b2 from parameters
dW1 ⟵ dW1 from gradient
db1 ⟵ db1 from gradient
dW2 ⟵ dW2 from gradient
db2 ⟵ db2 from gradient
W1 ⟵ W1 - (learning_rate * dW1)
b1 ⟵ b1 - (learning_rate * db1)
W2 ⟵ W2 - (learning_rate * dW2)
b2 ⟵ b2 - (learning_rate * db2)
parameters ⟵ array with entries W1,b1,W2 and b2
Pass out: parameters
Endfunction

Function nn_1_model()
Pass in: array X representing input data
Pass in: array y representing output data
Pass in: n1 representing the number of nodes in the hidden layer
Pass in: n_iteration representing the number of iterations
Pass in: learning_rate representing the learning rate
Pass in: print_cost representing whether to print the cost or not
n0, n2 ⟵ Call: layer_size(Pass: X,y)
costs ⟵ array with no entries to store the cost per iteration
parameters ⟵ Call: initialization_parameters(Pass: n0, n1, n2)
For i = 0 to n_iteration
A2, result ⟵ Call: forward_propagation(Pass: X, parameters)
cost ⟵ Call: total_cost(Pass: A2, y)

gradient ⟵ Call: backward_propagation(Pass: parameters, result, X, y)
parameters ⟵ Call: update_parameters(Pass: parameters, gradient, learn ing_rate)
If print_cost is true and i%100 = 0
add entry cost to array costs
Endif
Endfor
Plot: costs with respect to n_iteration with the fixed learning_rate
Pass out: parameters
Endfunction

parameters ⟵ Call: nn_1_model(Pass: X_train, y_train, n1 = 30, n_iteration = 100000, learning_rate = 0.01, print_cost = true)

Function predict()
Pass in: array parameters with entries W1 (weights of hidden layer), b1 (bias of hidden layer), W2 (weights of output layer), b2 (bias of output layer)
Pass in: array X representing input data
A2, result ⟵ Call: forward_propagation(Pass: X, parameters)
predictions ⟵ A2 > 0.5
Pass out: predictions * 1
Endfunction

Y_P_train ⟵ Call: predict(Pass: parameters, X_train)
Plot: confusion matrix with actual training output Y_train and predicted training output Y_P_train

Y_P_test ⟵ Call: predict(Pass: parameters, X_test)
Plot: confusion matrix with actual testing output Y_test and predicted testing output Y_P_test

4.3 Deep Neural Network

For the deep neural network model, I have used the inbuilt functions from the keras package to develop the model. I have used the tanh activation function in the hidden layers and the sigmoid activation function in the output layer. I have only created two hidden layers for this model. The creation of more layers follows easily from code given below.

import dataset
X ⟵ input data
y ⟵ output data
X_train ⟵ 80% of X (dataset is shuffled)
X_test ⟵ remaining 20% of X
y_train ⟵ 80% of Y
y_test ⟵ remaining 20% of Y

model ⟵ Call: models.Sequential(
Pass: Call : layers.Dense() Pass: 32 (number of nodes), activation = 'tanh'
Pass: Call : layers.Dense() Pass: 16 (number of nodes), activation = 'tanh'
Pass: Call : layers.Dense() Pass: 1 (number of nodes), activation = 'sigmoid')

optimizer ⟵ Call: optimizers.Adam()
Pass: learning_rate = 0.01

Call: model.compile
Pass: optimizer=optimizer, loss="binary_crossentropy" (binary classification)

history ⟵ Call: model.fit(
Pass: X_train, y_train, epochs = 100000 (number of iterations))

y_train_pred ⟵ Call: model.predict(Pass: X_train)
y_test_pred ⟵ Call: model.predict(Pass: X_test)

If y_train_pred ¿ 0.5
    y_train_pred ⟵ 1
Endif
Else
    y_train_pred ⟵ 0
Endelse

Plot: confusion matrix with y_train and y_train_pred Plot: confusion matrix with y_test and y_test_pred Plot: cost function

5 Results and Conclusion

The following is the dataset used for the program:

bmi	Age	asa_status	baseline_c	baseline_c	baseline_c	baseline_d	baseline_d	baseline_d	baseline_c	baseline_p	baseline_p	ahrq_ccs	ccsCompli	ccsMort30	complicati	dow	gender	hour	month	moonphas	mort30	mortali
19.31	59.2	1	1	0	0	0	0	0	0	0	0	19	0.18337	0.007424	-0.57	3	0	7.63	6	1	0	-0.
18.73	59.1	0	0	0	0	0	0	0	0	0	0	1	0.312029	0.016673	0.21	0	0	12.93	0	1	0	-0.
21.85	59	0	0	0	0	0	0	0	0	0	0	6	0.150706	0.001962	0	2	0	7.68	5	3	0	0.
18.49	59	1	0	1	0	0	1	1	0	0	0	7	0.056166	0	-0.65	2	1	7.58	4	3	0	-0.
19.7	59	1	0	0	0	0	0	0	0	0	0	11	0.197305	0.002764	0	0	0	7.88	11	0	0	
20.24	59	0	1	0	0	0	0	0	0	0	1	14	0.068478	0	0	1	0	7.63	0	3	0	0.
21.18	59	0	1	0	0	0	0	0	0	0	0	14	0.068478	0	0	0	0	9.62	10	3	0	
18.99	58.9	0	0	0	0	0	0	0	0	0	0	1	0.312029	0.016673	-0.38	4	0	8.6	6	1	0	
22.2	58.9	1	0	0	0	0	0	0	0	0	0	1	0.312029	0.016673	0	4	0	13	10	0	0	
20.83	58.9	1	1	6	0	0	0	1	0	0	0	18	0.466129	0.012903	1.87	4	1	10.05	5	1	0	2.

Figure 16: Dataset

This is a dataset with features such as gender, rate, mortality rate, health complications, etc. It has 24 columns (features) and 14636 rows (data points.) The ANN models created predict if the person would have health complications based on all the other features. The number of iterations was set to 10000 for all the models.

5.1 Single layer perceptron model

The confusion matrices and accuracies for the different perceptron models with different activation functions are as follows:

5.1.1 Perceptron model with the sign activation function

The sign activation function is just the binary step function with -1 instead of 0.

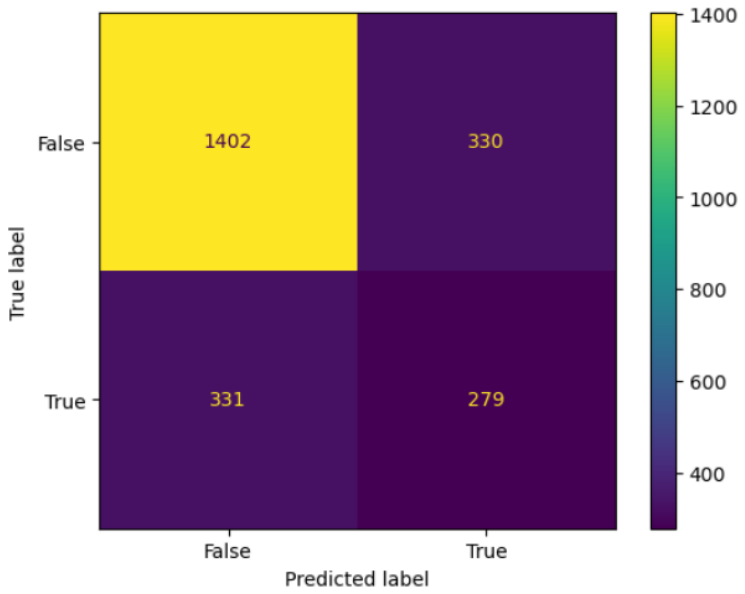


Figure 17: Confusion matrix of the training data with the sign activation function

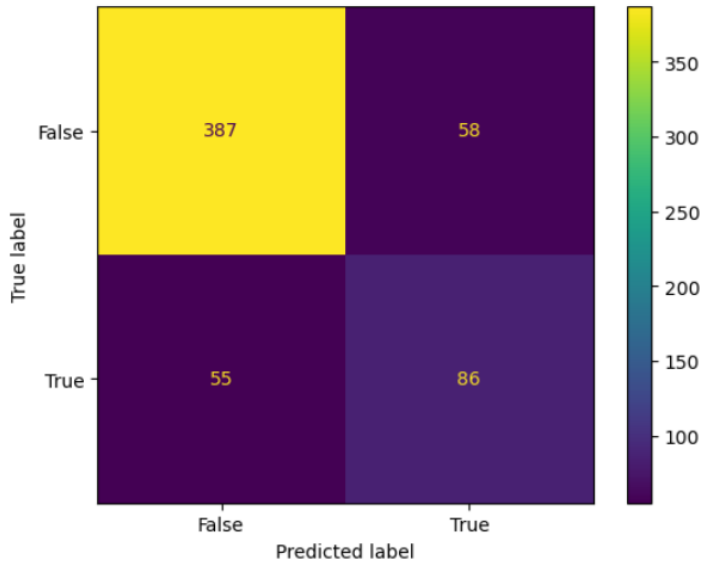


Figure 18: Confusion matrix of the testing data with the sign activation function

The accuracy of the model is as follows:

- 71.77% of the training data was classified accurately.
- 80.72% of the testing data was classified accurately.

5.1.2 Perceptron model with the sigmoid activation function

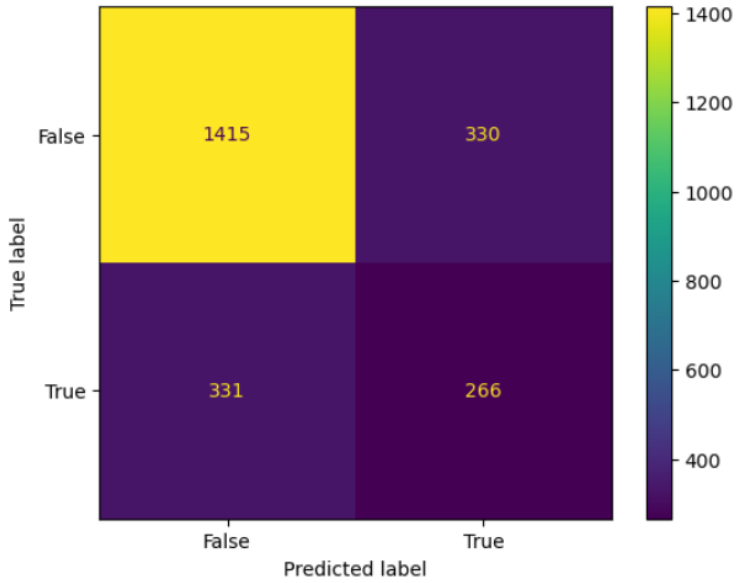


Figure 19: Confusion matrix of the training data with the sigmoid activation function

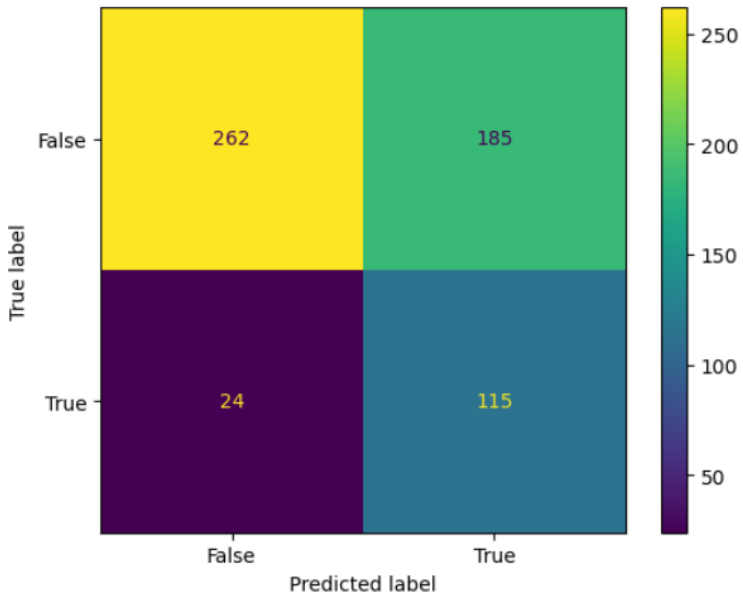


Figure 20: Confusion matrix of the testing data with the sigmoid activation function

The accuracy of the model is as follows:

- 71.77% of the training data was classified accurately.
- 64.33% of the testing data was classified accurately.

5.1.3 Perceptron model with the tanh activation function

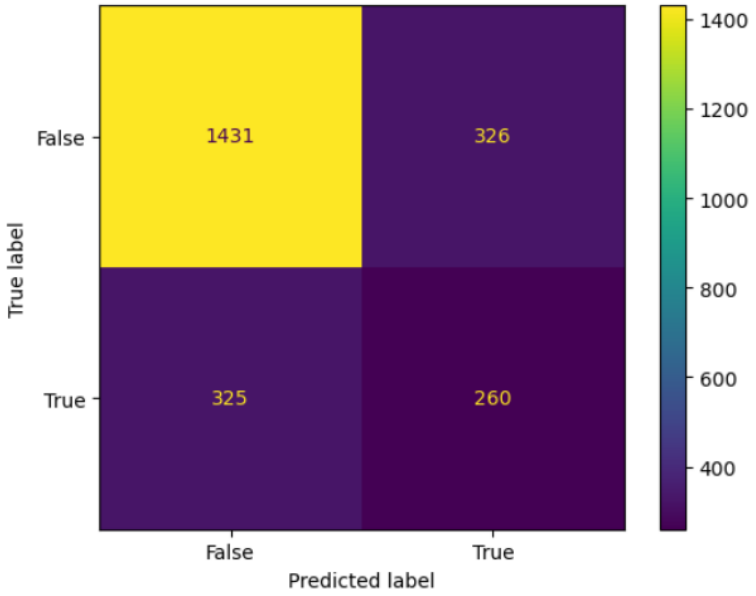


Figure 21: Confusion matrix of the training data with the tanh activation function

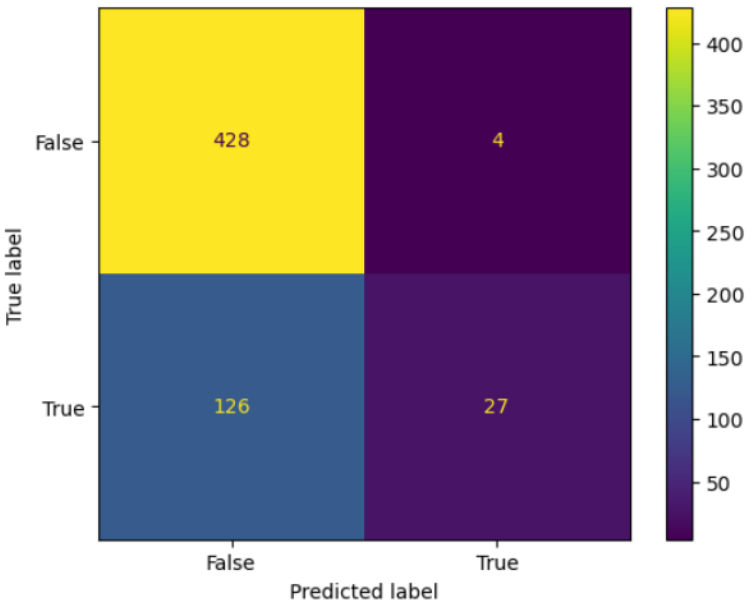


Figure 22: Confusion matrix of the testing data with the tanh activation function

The accuracy of the model is as follows:

- 72.20% of the training data was classified accurately.
- 77.77% of the testing data was classified accurately.

5.1.4 Perceptron model with the ReLU activation function

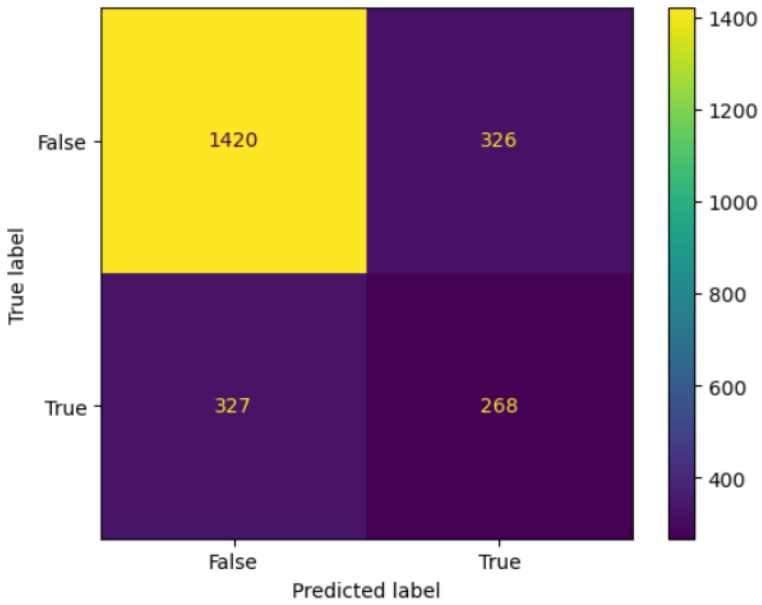


Figure 23: Confusion matrix of the training data with the ReLU activation function

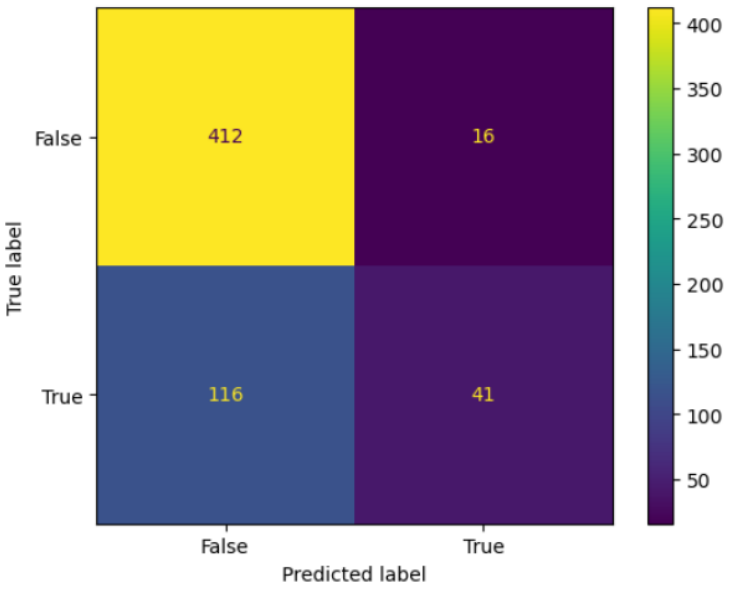


Figure 24: Confusion matrix of the testing data with the ReLU activation function

The accuracy of the model is as follows:

- 72.10% of the training data was classified accurately.
- 77.43% of the testing data was classified accurately.

5.1.5 Perceptron model with the leaky ReLU activation function

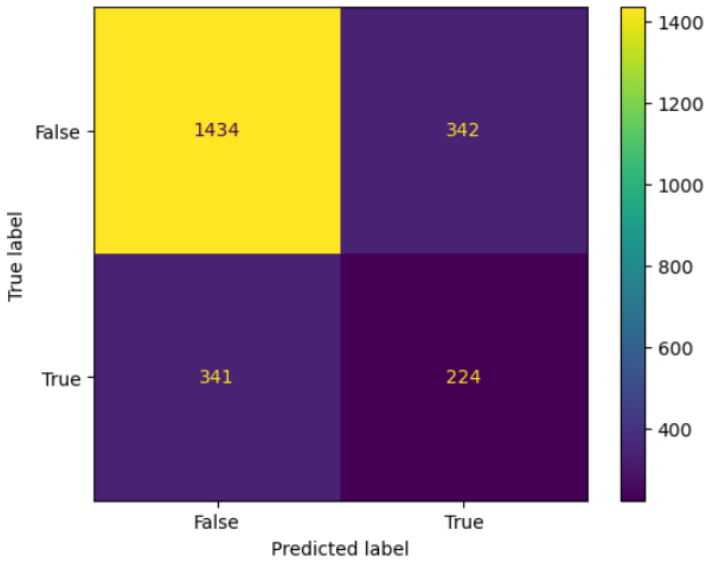


Figure 25: Confusion matrix of the training data with the leaky ReLU activation function

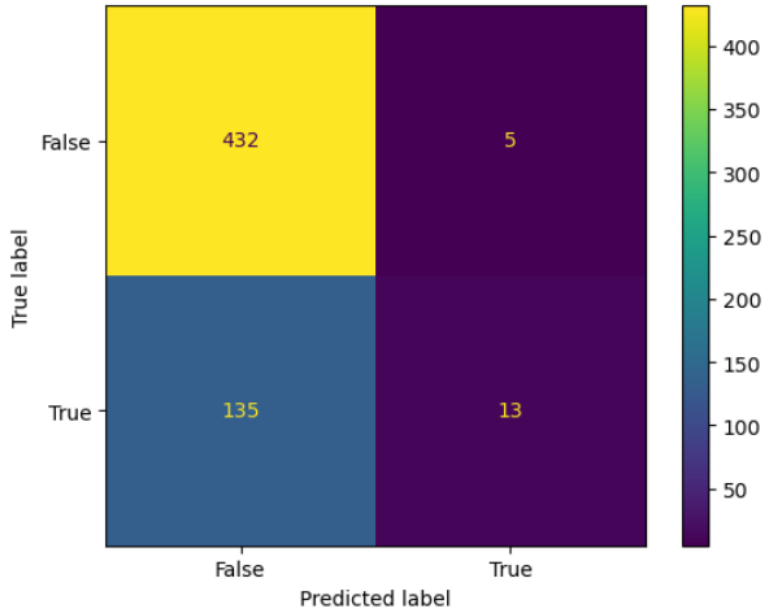


Figure 26: Confusion matrix of the testing data with the leaky ReLU activation function

The accuracy of the model is as follows:

- 70.82% of the training data was classified accurately.
- 76.06% of the testing data was classified accurately.

5.2 Single hidden layer ANN model

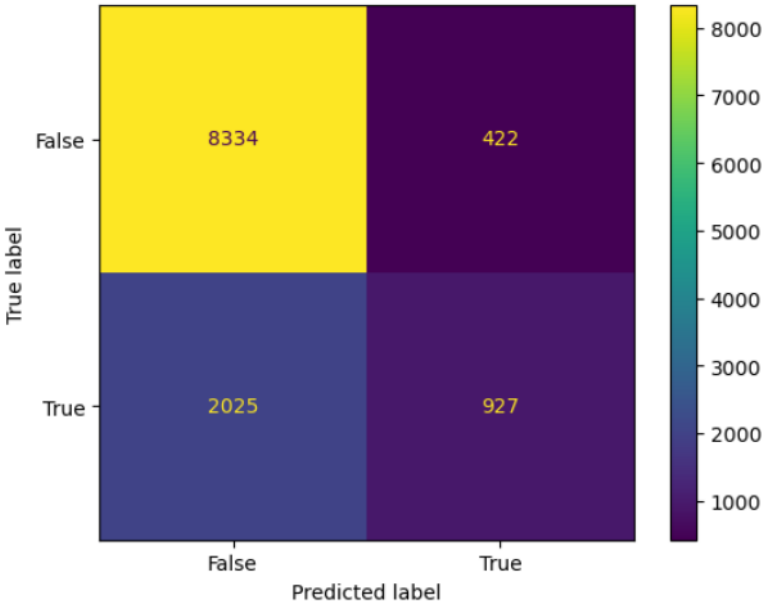


Figure 27: Confusion matrix of the training data of the ANN with one hidden layer

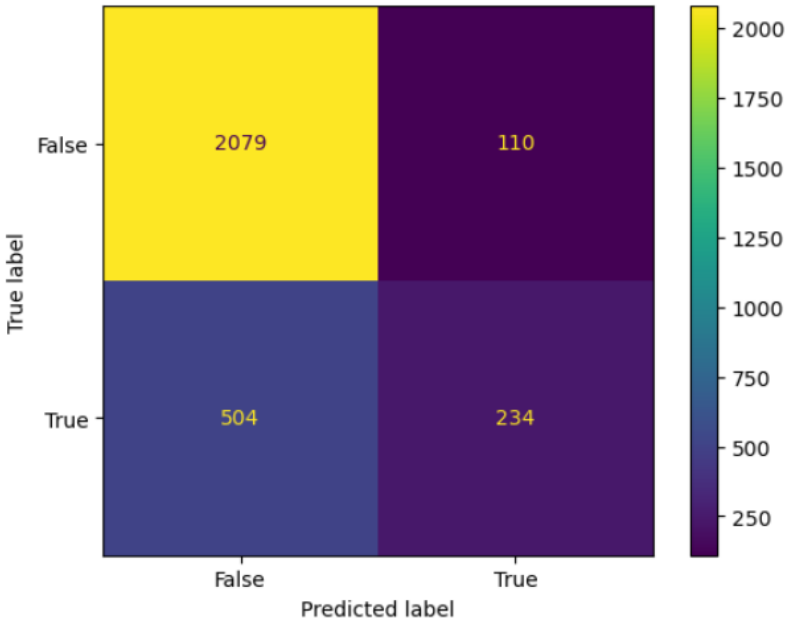


Figure 28: Confusion matrix of the testing data of the ANN with one hidden layer

The accuracy of the model is as follows:

- 79.09% of the training data was classified accurately.
- 79.02% of the testing data was classified accurately.

The plot of the cost with respect to the number of iterations is as follows:

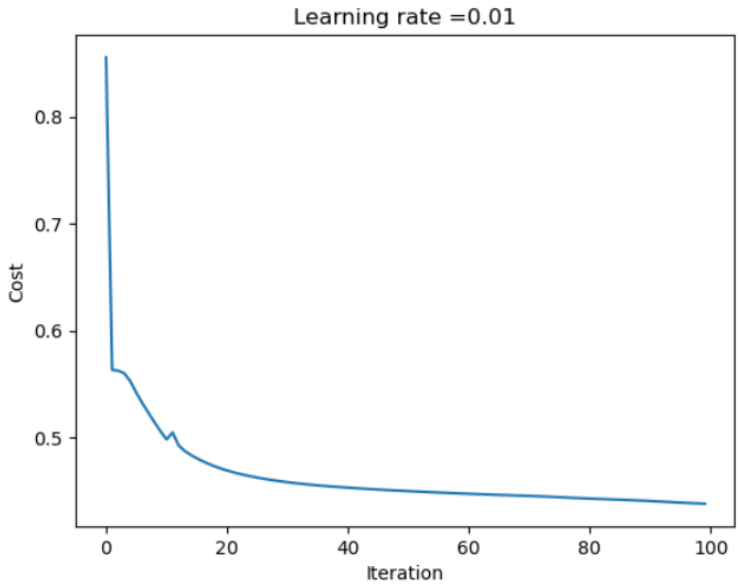


Figure 29: Graph of cost with respect to the number of iterations

5.3 Deep neural network model

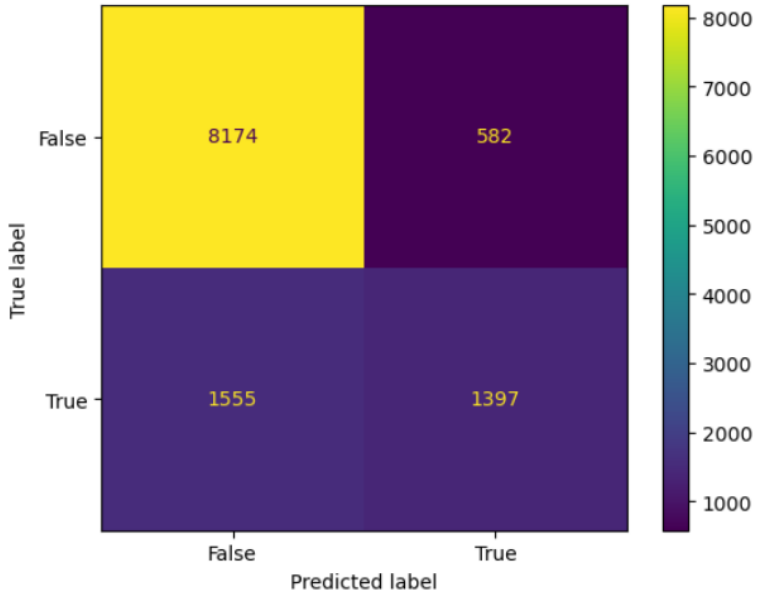


Figure 30: Confusion matrix of the training data of the deep neural network

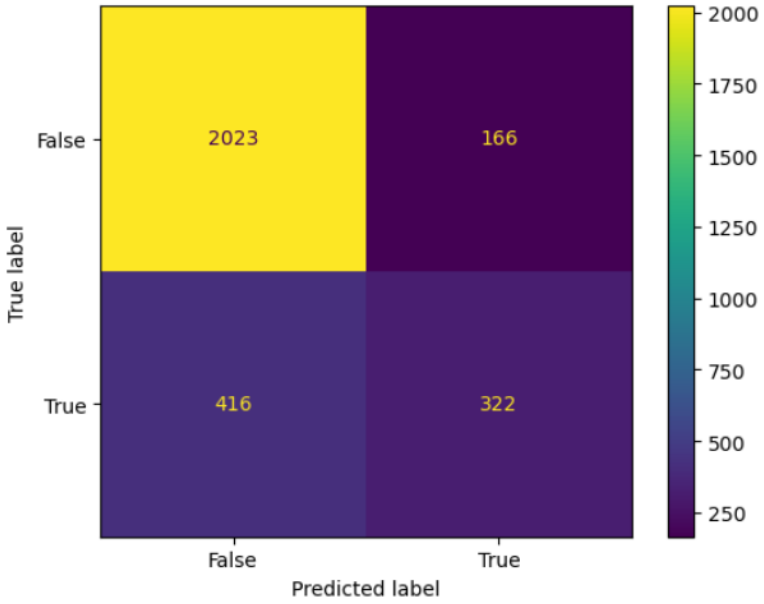


Figure 31: Confusion matrix of the testing data of the deep neural network

The accuracy of the model is as follows:

- 81.74% of the training data was classified accurately.
- 80.11% of the testing data was classified accurately.

The plot of the cost with respect to the number of iterations is as follows:

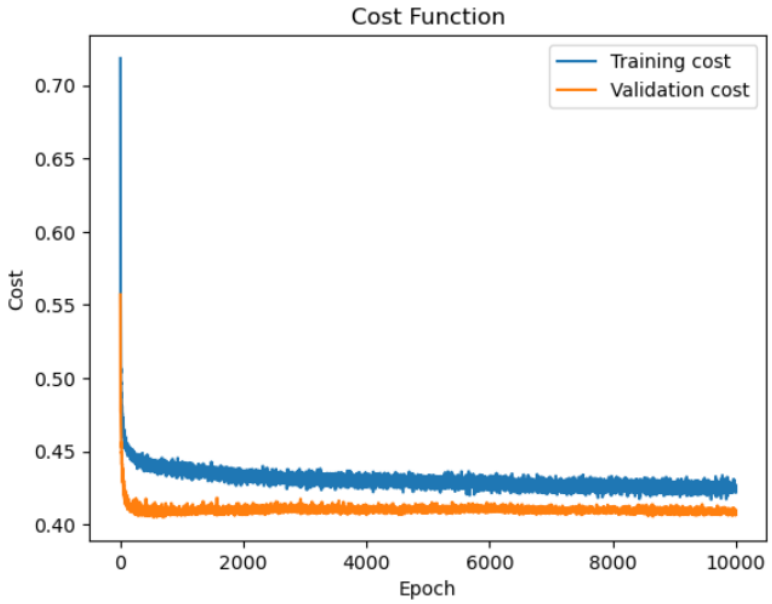


Figure 32: Graph of cost with respect to the number of iterations

5.4 Conclusion

As we can see from the results, the deep ANN model had a better accuracy in predicting the output than the ANN with a hidden layer which in turn had a better accuracy in predicting the output than the models with a single layer. Hence, neural networks with hidden layers are better at binary classification. All the perceptron models with the different activation functions produced similar results.

The use of neural networks has been a topic of great interest in the field of machine learning and artificial intelligence. The research conducted on neural networks has provided valuable insights into their respective strengths and limitations.

Single-layer neural networks are simple and easy to implement, but they may not be able to capture complex relationships in the data. On the other hand, deep neural networks, with their multiple layers, can model complex data structures and achieve higher accuracy in many applications.

In conclusion, the development and application of both single-layer and deep neural networks have significantly advanced the field of artificial intelligence and have provided powerful tools for solving a wide range of complex problems. As research in this area continues to evolve, it is likely that both types of neural networks will continue to play important roles in machine learning and AI.

6 Bibliography

Abu-Mostafa, Y. S., Magdon-Ismail, M., & Lin, H. (2012). Learning from data. AML-Book.

Alpaydin, E. (2010). Introduction to machine learning. MIT Press.

Bengio, Y. (2009). Learning deep architectures for AI. Foundations and Trends in Machine Learning, 2(1), 1-127.

Bishop, C. M. (1995). Neural Networks for Pattern Recognition. Oxford University Press.

Goh, R. (2018). Applied Data Science with Python Specialization (Online Course). Coursera. Retrieved from https://www.coursera.org/specializations/data-science-python

Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep Learning. MIT Press.

Gurney, K. (2014). An Introduction to Neural Networks. CRC Press.

Haykin, S. (1999). Neural Networks: A Comprehensive Foundation. Prentice Hall.

Hertz, J., Krogh, A., & Palmer, R. G. (1991). Introduction to the Theory of Neural Computation. Addison-Wesley.

Hornik, K., Stinchcombe, M., & White, H. (1989). Multilayer feedforward networks are universal approximators. Neural networks, 2(5), 359-366.

International Journal of Engineering Applied Sciences and Technology. (2020). ACTIVATION FUNCTIONS IN NEURAL NETWORKS, 2020 Vol. 4, Issue 12, ISSN No. 2455-2143, Pages 310-316. IJEAST. Retrieved from http://www.ijeast.com

Keras (2021). Keras Documentation (Online Resource).
Retrieved from https://keras.io/documentation/

Krose, B., & van der Smagt, P. (1996). An introduction to neural networks. University of Amsterdam, Amsterdam, The Netherlands.

McCulloch, W. S., & Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. The bulletin of mathematical biophysics, 5(4), 115-133.

McKinney, W. (2017). Python for data analysis: Data wrangling with Pandas, NumPy, and IPython. O'Reilly Media, Inc.

Minsky, M., & Papert, S. (1969). Perceptrons: An introduction to computational geometry. MIT press.

MIT. (n.d.). Introduction to Deep Learning (Online Course). MIT. Retrieved from https://introtodeeplearning.com/

Murphy, K. P. (2012). Machine Learning: A Probabilistic Perspective. MIT Press.

Ng, A. (2008). CS229: Machine Learning. Stanford University.
https://see.stanford.edu/Course/CS229

Ng, A. (2017). Deep Learning Specialization. Coursera.
https://www.coursera.org/specializations/deep-learning

Nielsen, M. (2015). Neural Networks and Deep Learning. Determination Press.

OmNamahShivai. (2021). Surgical dataset - binary classification. Kaggle.
https://www.kaggle.com/datasets/omnamahshivai/surgical-dataset-binary-classification

Reddy, G. (2019). Deep Learning (Online Course). Indian Institute of Technology Madras. Retrieved from https://nptel.ac.in/courses/106/106/106106182/

Rosenblatt, F. (1958). The perceptron: a perceiving and recognizing automaton. Cornell Aeronautical Laboratory, Report No. VG-1196-G-1.

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. Nature, 323(6088), 533-536.

Schmidhuber, J. (2015). Deep learning in neural networks: An overview. Neural Networks, 61, 85-117.

Severance, C. (2016). Python for everybody: Exploring data in Python 3. CreateSpace Independent Publishing Platform.

VanderPlas, J. (2016). Python data science handbook: Essential tools for working with data. O'Reilly Media, Inc.

Woodruff, A. (n.d.). What is a neuron? Queensland Brain Institute, The University of Queensland. https://qbi.uq.edu.au/brain/brain-anatomy/what-neuron