Feed Forward Neural Networks

Iheb Gafsi*
INSAT Student
iheb404notfound@gmail.com

Abstract:

Neural Networks are one of the greatest inventions of all time where human made the machine a thinking creature that can do complex tasks that even humans can't do.

In this Document we will dive into the details of machine learning works and we will learn some techniques that can help us. We won't go into hard coding but instead this will cover up all the details needed to make a neural network from Scratch

Introduction:

Just like the human brain, we designed a paradigm that is based on neural networks to learn, remember, and make predictions. Albeit they work slightly different, they both lead to significant results.

Mechanisms:

These neural networks are chosen to be composed with layers of neurons and every neuron is connected to the other neurons from the previous layer. **Figure 1** shows how a neural network made of 4 layers looks like.

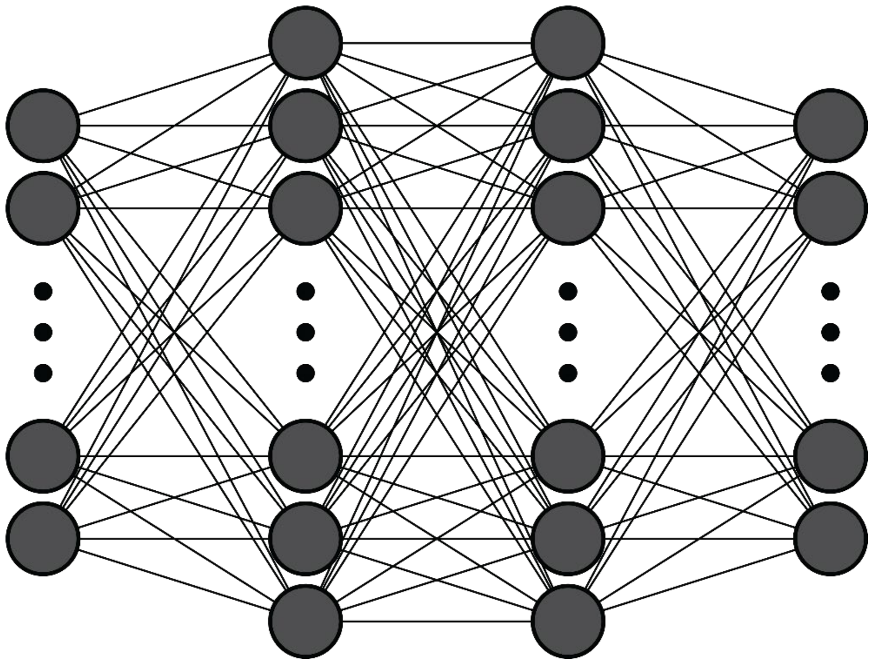


Figure 1

Note that every neural network should have an input layer and an output layer and additionally hidden layers like in Figure 1. Every neuron of the neural network holds an intensity with which it affects the output.

Mathematically:

We chose these neurons to hold floating numbers so the higher the number it holds, the higher intensity and effect it has. Similarly, all the connections are going to be just some weights and biases that are multiplied and added to the values of the neurons to result the next neuron's value. These weights and biases could be corrected/learned during the training period of the model before releasing it but wouldn't that mean that we're doing a linear regression but in a fancy way? Therefore, we will use what's called as activation functions to shape our output freely. So, the next layer z_{ij} is going to be equal to the sum of weighted and biased neurons from the previous layer.

$$z_{ij} = \sum_j w_{ij} n_{(i-1)j} + b_{ij}$$

Where w is the corresponding weight and b is the bias added. Just then we apply an activation function $n_{ij} = \Gamma(z_{ij})$. We will talk more about activation functions later in this document. Thus, our neural network is just a bunch of numbers that are added and multiplied and an activation function applied on every layer.

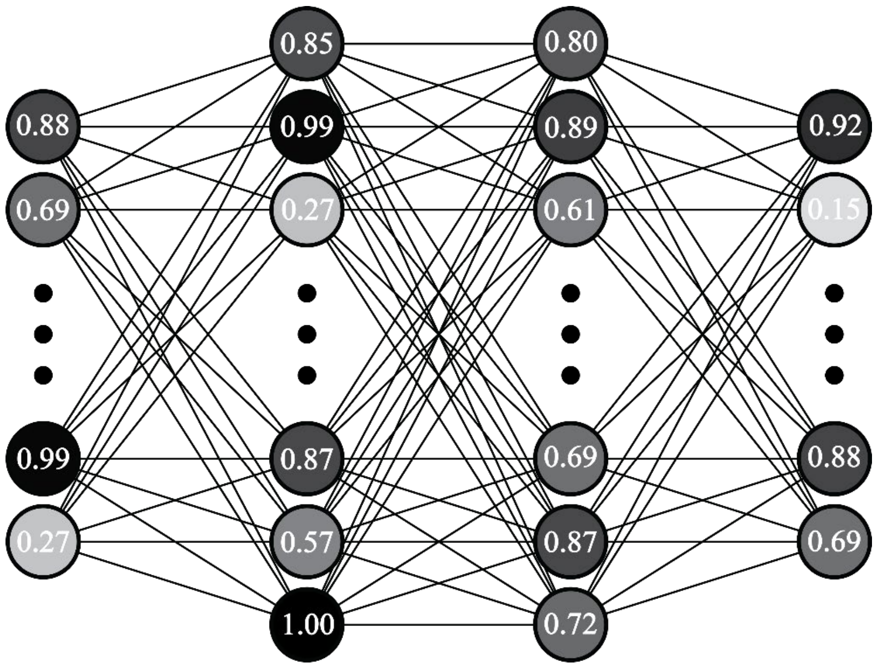


Figure 2

Note: . represents the Matrix Multiplication law

$\odot$ represents the Hadamard Product law

Activation functions:

It is highly advised to have the bijectivity of a function so that it can be an activation function. Here we're covering the most popular activation functions that we can use in our models.

- **Sigmoid:**

  Name: Sigmoid

  Properties: Sigmoid is a bijective function the output of which is delimited between 0, 1.

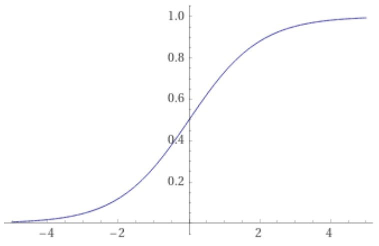
  Domain: $\sigma: \mathbb{R} \to]0,1[$

  Definition:

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

  Derivative:

$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x)\big(1-\sigma(x)\big)$$



- **ReLU:**

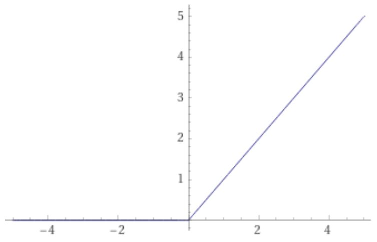
  Name: Rectified Linear Unit ReLU

  Properties: ReLU's specific property is that it's null when the input is negative and equals to the input in the other case.

  Domain: $ReLU: \mathbb{R} \to \mathbb{R}^+$

  Definition:

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



  Derivative: it's important to note that mathematically ReLU isn't differentiable in 0 but conventionally we will consider it to be 0.

$$\frac{\partial ReLU(x)}{\partial x} = \begin{cases} 1, & x > 0 \\ 0, & x \le 0 \end{cases}$$

- **Softplus:**

  Name: Softplus

  Properties: Softplus has almost the same shape as the ReLU function but it's smoother and more versatile to shaping.

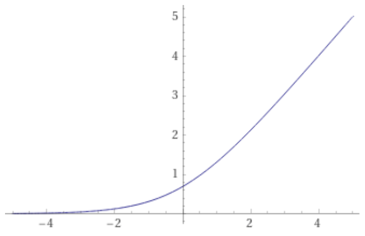
  Domain: $\zeta: \mathbb{R} \to \mathbb{R}^+$

  Definition:

$$\zeta(x) = \ln(1+e^{x})$$

  Derivative:

$$\frac{\partial \zeta(x)}{\partial x} = \frac{e^{x}}{1+e^{x}}$$



- **Softmax:**
  Name: Softmax
  Properties: Softmax is a different type of activation functions as it doesn't depend only on x, it also depends on the others, what it essentially does is calculating the probabilities of every and each x in respect to others that sum up to one. It is most used in the classification problems.
  Domain: $\varsigma: \mathbb{R}^N \to \mathbb{R}^N$
  Definition:

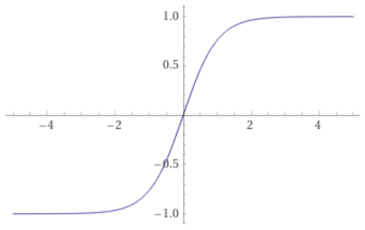
$$\varsigma(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{N} e^{x_j}}$$

  Derivative:

$$\frac{\partial \varsigma(x_i)}{\partial x} = \varsigma(x_i)\big(1-\varsigma(x_i)\big) - \sum_{j\neq i}^{N} \varsigma(x_i)\varsigma(x_j)$$

- **Tanh:**
  Name: Hyperbolic Tangent
  Properties: Tanh is a very common bijective hyperbolic function that is delimited between -1 and 1. It is very smooth and versatile.
  Domain: $tanh: \mathbb{R} \to]-1,1[$
  Definition:

$$\tanh x = \frac{\sinh x}{\cosh x} = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

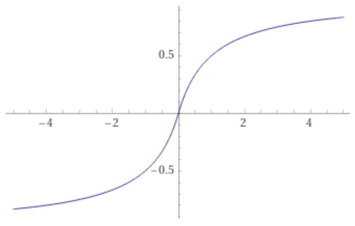


  Derivative:

$$\frac{\partial \tanh x}{\partial x} = \frac{1}{\cosh^2 x} = \operatorname{sech}^2 x$$

- **Softsign:**
  Name: Softsign
  Properties: The same properties as tanh.
  Domain: $ss: \mathbb{R} \to]-1,1[$
  Definition:

$$ss(x) = \frac{x}{1+|x|}$$



  Derivative:

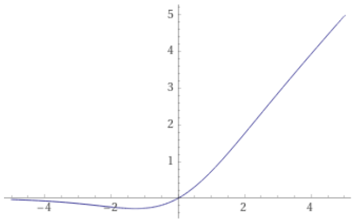
$$\frac{\partial ss(x)}{\partial x} = \frac{1}{2|x| + x^2 + 1}$$

- **Swish:**
  <u>Name:</u> Swish
  <u>Properties:</u> This function looks just like
  <u>Domain:</u> $sw: \mathbb{R} \to]-1,1[$
  <u>Definition:</u>

$$sw(x) = x\sigma(x)$$

  <u>Derivative:</u>

$$\frac{\partial sw(x)}{\partial x} = sw(x) + \sigma(x)\big(1 - sw(x)\big)$$

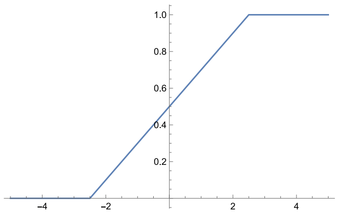


- **Hard Sigmoid:**
  <u>Name:</u> Hard Sigmoid
  <u>Properties:</u> Just a replicate of the sigmoid function but hardened up
  <u>Domain:</u> $\varpi: \mathbb{R} \to [0,1]$
  <u>Definition:</u>

$$\varpi(x) = \begin{cases} 0, & x < -2.5 \\ 1, & x > 2.5 \\ 0.2x + 0.5 \end{cases}$$

  <u>Derivative:</u>

$$\frac{\partial \varpi(x)}{\partial x} = \begin{cases} 0, & x \leq -2.5 \\ 0, & x \geq 2.5 \\ 0.2 \end{cases}$$

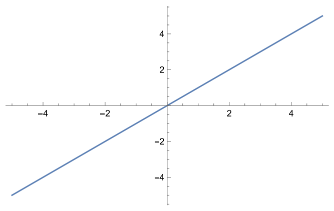


- **Linear:**
  <u>Name:</u> Linear
  Properties: This is the identity function, we can use it for linear regression problems, it's rigid so it's no match for complex neural networks.
  <u>Domain:</u> $\lambda: \mathbb{R} \to \mathbb{R}$
  <u>Definition:</u>

$$\lambda(x) = x$$

  <u>Derivative:</u>

$$\frac{\partial \lambda(x)}{\partial x} = 1$$

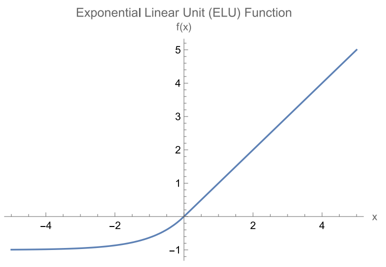


- **ELU:**
  <u>Name:</u> Exponential Linear Unit.
  <u>Properties:</u> This function is similar to ReLU but it avoids the dying ReLU problem as it's bijective and smooth.
  <u>Domain:</u> $ELU_{alpha}: \mathbb{R} \to]-\alpha, +\infty[$
  <u>Definition:</u>

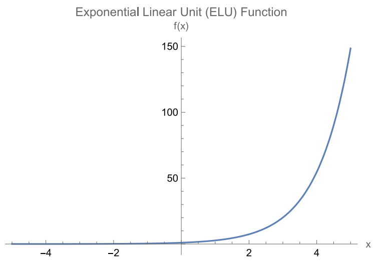
$$ELU_{\alpha}(x) = \begin{cases} \alpha(e^x - 1), & x < 0 \\ x, & x \geq 0 \end{cases}$$

  <u>Derivative:</u>

$$\frac{\partial ELU_{\alpha}(x)}{\partial x} = \begin{cases} \alpha e^x, & x < 0 \\ 1, & x \geq 0 \end{cases}$$



- **Exponential:**
  Name: Exponential
  Properties: It is a quite smooth bijective function that suits the training, the only inconvenient is that we can face overflow when dealing with massive numbers.
  Domain: $exp: \mathbb{R} \rightarrow \mathbb{R}$
  Definition:

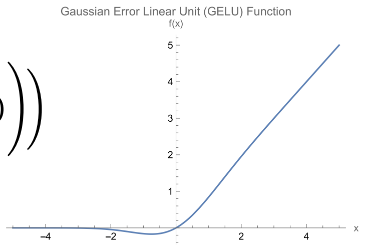


$$\exp(x) = e^{x}$$

  Derivative:

$$\frac{\partial \exp(x)}{\partial x} = \exp(x)$$

- **GELU:**
  Name: Gaussian Error Linear Unit
  Properties: It's another smooth bijective approximation to the ReLU activation function.
  Domain: $GELU: \mathbb{R} \rightarrow]-0.11, +\infty[$
  Definition:



$$GELU(x) = \frac{x}{2}\left(1 + \tanh\left(\sqrt{\frac{\pi}{2}}(x + 0.044715x^{3})\right)\right)$$

  Derivative:

$$\frac{\partial GELU(x)}{\partial x} = 0.398942x(1 + 0.13414x^{2})\,\mathrm{sech}^{2}\left(\sqrt{\frac{\pi}{2}}(x + 0.044715x^{3})\right) + \frac{1}{2}\left(1 + \tanh\left(\sqrt{\frac{\pi}{2}}(x + 0.044715x^{3})\right)\right)$$

Forward Propagation:

Now that you have a basic understanding of how neural networks work in general, we will dive into more details of the propagation of the input to the output which is called Forward Propagation.
Conventionally we will represent every layer of neurons with a vector that has the dimensionality of the units of that particular layer therefore the weights are going to be matrices and the biases are going to be vectors with the same size as the corresponding layer. The table down below emphasizes the lengths of every matrix and vector, the vector x stands for the input, y for the output

Matrix/Vector	**Dimension**
i	a_0
w_1	$a_1 \times a_0$
$z_1,\ l_1,\ b_1$	$a_1 \times 1$
w_n	$a_n \times a_{n-1}$
$z_n,\ l_n,\ b_n$	$a_n \times 1$
o	$a_N \times 1$

Initially, all the weights are going to be initialized randomly (preferably from the standard distribution) and the biases are going to be initialized to zero. Then the model will learn and adjust these parameters. For instance, we can use the neural network of Figure 2 as an example. We will use the Sigmoid activation function for every layer. So, to get to the output we will follow the next steps:

- Input to first hidden layer:
$$z_1 = w_1 i + b_1$$
$$z_1 = \begin{pmatrix} 0.3 & \cdots & 1.0 \\ \vdots & \ddots & \vdots \\ -0.2 & \cdots & 2.1 \end{pmatrix} \begin{pmatrix} 0.8 \\ \vdots \\ 0.2 \end{pmatrix} + \begin{pmatrix} 0.1 \\ \vdots \\ -2 \end{pmatrix} = \begin{pmatrix} 3 \\ \vdots \\ 0.1 \end{pmatrix}$$
$$l_1 = \sigma(z_1)$$
$$l_1 = \sigma\left(\begin{pmatrix} 3 \\ \vdots \\ 0.1 \end{pmatrix}\right) = \begin{pmatrix} 0.9 \\ \vdots \\ 0.5 \end{pmatrix}$$
- First hidden layer to second hidden layer:
$$z_2 = w_2 l_1 + b_2$$
$$z_2 = \begin{pmatrix} 1.2 & \cdots & -3.7 \\ \vdots & \ddots & \vdots \\ 0.9 & \cdots & -1.2 \end{pmatrix} \begin{pmatrix} 0.9 \\ \vdots \\ 0.5 \end{pmatrix} + \begin{pmatrix} 3.5 \\ \vdots \\ 1 \end{pmatrix} = \begin{pmatrix} 0.5 \\ \vdots \\ -2 \end{pmatrix}$$
$$l_2 = \sigma(z_2)$$

$$l_1 = \sigma\left(\begin{pmatrix}0.5\\ \vdots\\ -2\end{pmatrix}\right) = \begin{pmatrix}0.6\\ \vdots\\ 0.1\end{pmatrix}$$

- Second hidden layer to output:
  $z_3 = w_3 l_2 + b_3$
  $$z_3 = \begin{pmatrix}0.3 & \cdots & -1.7\\ \vdots & \ddots & \vdots\\ -2 & \cdots & -1.2\end{pmatrix}\begin{pmatrix}0.6\\ \vdots\\ 0.1\end{pmatrix} + \begin{pmatrix}-0.2\\ \vdots\\ 0.6\end{pmatrix} = \begin{pmatrix}-1\\ \vdots\\ 1.2\end{pmatrix}$$
  $o = \sigma(z_3)$
  $$o = \sigma\left(\begin{pmatrix}-1\\ \vdots\\ 1.2\end{pmatrix}\right) = \begin{pmatrix}0.2\\ \vdots\\ 0.7\end{pmatrix}$$

So, in general:

$$\boldsymbol{z_n} = \boldsymbol{w_n l_{n-1}} + \boldsymbol{b_n}$$
$$\boldsymbol{l_n} = \boldsymbol{\Gamma_n(z_n)}$$

Backward Propagation:

So far you learned how to feed forward a neural network, but all the weights and biases that we were using were totally random and do not lead to a precise result. Thus, by having the features (the inputs) and their labels (their correct output) we can optimize our model to make better predictions and higher accuracy. Therefore, we will use the backward propagation where we adjust the parameters of the connections every time, we make a prediction. The optimizer functions just calculate the loss with a loss function and try to minimize it. All the optimizers use what's known as Gradient Descent. We will cover up different optimizers and loss functions later in this document.

Note that the backward propagation only happens in the training phase of the model.

Loss Functions:

- **Mean Squared Error MSE:**

  To calculate the loss $\mathcal{L}$ we calculate the mean of the squared residuals
  $$\mathcal{L} = \frac{1}{n}\sum_{i=1}^{n}\left(Y_i - \widehat{Y}_i\right)^2$$
  Most importantly, in the matrix form, we will essentially adjust parameters based on the squared residuals, note that the loss is the mean of the components of the resulting vector.
  $$(Y - \hat{Y})\odot(Y - \hat{Y}) = \left[\begin{pmatrix}0.2\\ \vdots\\ 0.7\end{pmatrix} - \begin{pmatrix}0\\ \vdots\\ 1\end{pmatrix}\right]\odot\left[\begin{pmatrix}0.2\\ \vdots\\ 0.7\end{pmatrix} - \begin{pmatrix}0\\ \vdots\\ 1\end{pmatrix}\right]$$
  $$= \begin{pmatrix}0.2\\ \vdots\\ -0.3\end{pmatrix}\odot\begin{pmatrix}0.2\\ \vdots\\ -0.3\end{pmatrix} = \begin{pmatrix}0.04\\ \vdots\\ 0.09\end{pmatrix}$$

- **Cross Entropy:**

  The Cross Entropy loss function calculates the dissimilarity between two probability distributions, we commonly use this activation function when dealing with classification problems where we have probabilities or logits. Note that logits are the raw unnormalized outputs of the model, which means that we didn't apply any activation function on them.
  The Cross Entropy loss function looks like this:

  $$\mathcal{L} = -\sum_{i=1}^{n} \widehat{Y}_i \ln Y_i$$

  But when working with yes or no results, only the corresponding observed output is going to be equal to 1 and the others are going to be equal to 0. Therefore, we can simplify our loss function to be equal to:

  $$\mathcal{L} = -\ln Y$$

  What makes the Cross Entropy more effective in the classification problems than others is that unlike others, the more the predicted output Y gets closer to 0 when it's supposed to be 1, the more the loss explodes. In the other hand the closer it is to 1, the smaller the loss gets. Here's a short example:

  $$-\ln\begin{pmatrix}0.2\\ \vdots\\ 0.7\end{pmatrix} = \begin{pmatrix}1.6\\ \vdots\\ 0.3\end{pmatrix}$$

- **Categorical Cross Entropy:**

  The Categorical Cross Entropy loss function is a particular form of the Cross Entropy. It is often used when the classes are mutually exclusive. Where the predicted outputs are probabilities or outputs. The loss is calculated the same way.
- **Binary Cross Entropy:**

  It's also a specific type of the of the Cross Entropy, it is often used when dealing with two classes where the outputs are probabilities or logits. The loss is equal to:

  $$\mathcal{L} = -(\hat{Y}\ln Y + \left(1-\hat{Y}\right)\ln(1-Y))$$
- **Sparse Categorical Cross Entropy:**

  Sparse Categorical Cross Entropy loss function is a type of Categorical Cross entropy but it works with numbers as outputs instead of one-hot-encoded vectors. Therefore, the loss is found this way:

  $$\mathcal{L} = -\ln Y[\hat{Y}]$$
- **Mean Absolute Error MAE:**

  Works the same way as MSE but to get rid of the signs this time we use the absolute value instead of squaring the error. It should lead to the same result as the MSE. The loss is calculated like:

  $$\mathcal{L} = \frac{1}{n}\sum_{i=1}^{n}\left|Y_i - \widehat{Y}_i\right|$$

- **Mean Squared Logarithmic Error MSLE:**

  This function looks like the Mean Error loss functions but it's more used when the output takes an exponential like path, Thus, it is most used in finance and other scientific applications. The loss is calculated like:

$$\mathcal{L} = \frac{1}{n}\sum_{i=1}^{n}\left(\ln Y_i - \ln \widehat{Y_i}\right)^2$$

- **Hinge:**

  The Hinge loss function is commonly used in the classification problems, it is most used in the Support Vector Machines SVMs and it helps maximizing the margin between classes to encourage the correct classifications. The loss is calculated like:

$$\mathcal{L} = \max\left(0,1 - Y.\hat{Y}\right)$$

  Where $Y.\hat{Y}$ represents the vector dot product between the observed and the predicted outputs.

- **Categorical Hinge:**

  Categorical Hinge is just a hybrid of the Hinge loss functions but it's used more on the classification problems, and just like the Hinge loss, it encourages the correct predictions by maximizing the margin between them and the incorrect ones. The loss looks like:

$$\mathcal{L} = \max\left(0,1 - Y\left[\hat{Y}\right] + \max\,(Y_i \neq \hat{Y})Y\left[\hat{Y}\right]\right)$$

- **Huber:**

  The Huber loss is just a combination of MSE and MAE that is commonly used in regression tasks, the loss calculation looks like:

$$\mathcal{L} = \begin{cases} \frac{1}{2}\left(Y_i - \widehat{Y_i}\right)^2, & \left|Y_i - \widehat{Y_i}\right| \leq \delta \\ \delta\left(\left|Y_i - \widehat{Y_i}\right| - \frac{\delta}{2}\right), & \left|Y_i - \widehat{Y_i}\right| > \delta \end{cases}$$

  Where δ is the threshold parameter.

- **Poison:**

  The Poisson loss is derived from the Poisson distribution and measures discrepancy between observed and predicted values, it is often used with exponential activation functions to ensure the positivity of the outputs. It is particularly suitable for modeling count data, where the target variable represents the number of occurrences of an event. The loss is calculated like:

$$\mathcal{L} = e^{-Y}\frac{Y^{\hat{Y}}}{\hat{Y}!}$$

Optimizers:

Now that we've learnt how to calculate the loss and perform the Forward Propagation, we should now learn how to optimize our model and adjust the weights and biases, Therefore we have a couple of optimizers with specific properties.

- **Gradient Descent:**

  The Gradient Descent is one of the most powerful algorithms of all time and it is also the building block of all the other optimizers.
  The Gradient Descent Algorithm takes the loss of the output and calculates all the gradients for the parameters and then subtracts them from the initial values to minimize the error and make better predictions, so let a_n be a parameter, γ be the learning rate which is just a small constant to adjust the speed of the change of the parameters, and F the function of the output in respect to a_n, so what the gradient descent does is:

$$a_{n+1} = a_n - \gamma \nabla F(a_n)$$

  So, we can use the example of earlier to calculate the loss and modify the parameters, we will use the MSE loss in this particular example:

  The learning rate is going to be fixed to: $\gamma = 0.01$
  The loss vector here is:

$$\mathcal{L} = \left(\hat{Y} - Y\right)^2 = \begin{pmatrix} 0.2 \\ \vdots \\ -0.3 \end{pmatrix} \odot \begin{pmatrix} 0.2 \\ \vdots \\ -0.3 \end{pmatrix} = \begin{pmatrix} 0.04 \\ \vdots \\ 0.09 \end{pmatrix}$$

  Now we should calculate the gradients with respect to every variable:
  Y:

$$\frac{\partial \mathcal{L}}{\partial Y} = \frac{\partial \left(\hat{Y} - Y\right)^2}{\partial Y} = 2\left(\hat{Y} - Y\right)\frac{\partial\left(\hat{Y} - Y\right)}{\partial Y} = -2\left(\hat{Y} - Y\right)$$

  Z_3:

$$\frac{\partial \mathcal{L}}{\partial Z_3} = \frac{\partial \mathcal{L}}{\partial Y} \odot \frac{\partial Y}{\partial Z_3} = \frac{\partial \mathcal{L}}{\partial Y} \odot \frac{\partial \sigma(Z_3)}{\partial Z_3} = \frac{\partial \mathcal{L}}{\partial Y} \odot \sigma'(Z_3)$$

  b_3:

$$\frac{\partial \mathcal{L}}{\partial b_3} = \frac{\partial \mathcal{L}}{\partial Z_3} \odot \frac{\partial Z_3}{\partial b_3} = \frac{\partial \mathcal{L}}{\partial Z_3} \odot \frac{\partial w_3 l_2 + b_3}{\partial b_3} = \frac{\partial \mathcal{L}}{\partial Z_3}$$

  w_3:

$$\frac{\partial \mathcal{L}}{\partial w_3} = \frac{\partial \mathcal{L}}{\partial Z_3} \odot \frac{\partial Z_3}{\partial w_3} = \frac{\partial \mathcal{L}}{\partial Z_3} \odot \frac{\partial w_3 l_2 + b_3}{\partial w_3} = \frac{\partial \mathcal{L}}{\partial Z_3} l_2^T$$

  l_2 :

$$\frac{\partial \mathcal{L}}{\partial l_2} = \frac{\partial \mathcal{L}}{\partial Z_3} \odot \frac{\partial Z_3}{\partial l_2} = \frac{\partial \mathcal{L}}{\partial Z_3} \odot \frac{\partial w_3 l_2 + b_3}{\partial l_2} = w_3^T \frac{\partial \mathcal{L}}{\partial Z_3} \odot \frac{\partial l_2}{\partial l_2} = w_3^T \frac{\partial \mathcal{L}}{\partial Z_3}$$

  Z_2 :

$$\frac{\partial \mathcal{L}}{\partial Z_2} = \frac{\partial \mathcal{L}}{\partial l_2} \odot \frac{\partial l_2}{\partial Z_2} = \frac{\partial \mathcal{L}}{\partial l_2} \odot \sigma'(Z_2)$$

  And you just have to continue till reach the first layer. Then we can adjust our parameters:

$$w_{3n+1} = w_{3n} - \gamma \frac{\partial \mathcal{L}}{\partial w_{3n}}$$
$$b_{3n+1} = b_{3n} - \gamma \frac{\partial \mathcal{L}}{\partial b_{3n}}$$
$$\vdots$$
$$a_{n+1} = a_n - \gamma \frac{\partial \mathcal{L}}{\partial a_n}$$

Etc… Now let's recap and make it General, the table down below shows exactly how to calculate for any parameter, without diving into the math for every one of the parameters. The table down below illustrates all of these calculations.

Parameter	**Gradient**
Y	$-2(\hat{Y} - Y)$
Z_N	$\frac{\partial \mathcal{L}}{\partial Y} \odot f_N{}'(Z_N)$
Z_n	$w_{n+1}^T \frac{\partial \mathcal{L}}{\partial Z_{n+1}} \odot f_n{}'(Z_n)$
b_n	$\frac{\partial \mathcal{L}}{\partial Z_n}$
w_n	$\frac{\partial \mathcal{L}}{\partial Z_n} l_{n-1}^T$

- **Stochastic Gradient Descent SGD:**

  Now that you have already learnt what the original Gradient Descent algorithm works, Stochastic Gradient Descent (SGD) is just the same algorithm but instead of compiling the data for the entire dataset, we calculate the gradients for every data point, this prevents the long steps and gets closer to the minima of the loss.

- **Mini-Batch Gradient Descent:**

  Unlike the Original Gradient Descent and the Stochastic Gradient Descent, The Mini-Batch Gradient Descent adjusts the gradients after a specified batch of data as the Gradient Descent makes big steps and in the other hand the SGD makes some kind of noise and it doesn't make progress that easily so the Mini-Batch made a good solution in the middle.

- **AdaGrad:**

  In the previous algorithms, using a fixed learning rate made it harder for us to get to the minimum point of the loss. Therefore, the Adaptative Gradient Descent (AdaGrad) algorithm made variable individual learning rate for every parameter. This works by accumulating the gradients every iteration. The formula to correct the learning rate is:

$$\gamma_n = \frac{\eta}{\sqrt{a_n + \epsilon}}$$
$$a_n = \sum_{i=1}^{n} G_i^2$$

$$G_n = \frac{\partial \mathcal{L}}{\partial \rho_n}$$

Where a_n is the accumulator of the squared gradients G which is going to be initialized to 0, η is a constant that we choose and ϵ is a small constant usually equals to 10^{-8} to prevent dividing by 0 if a_n. And ρ is a parameter that could be a weight or a bias or just another learnable constant.

- **RMSprop:**

It turns out that Adagrad has few disadvantages that Root Mean Square Propagation (RMSprop) fixes, one of these is the accumulation control problem as the squared gradients lead to an increase of the sum and monotonically aggressive learning rate reductions. RMSprop also gets rid of the diminishing learning rate problem and the non-convex handling issue that AdaGrad faces. It is also more efficient at accelerating to get to final minima as it scales down the movements in the wrong direction.
RMSprop adapts the learning rate for each parameter in the network based on the average magnitudes of recent gradients. It maintains a running average of squared gradients for each parameter and uses this average to scale the learning rate. It calculates this moving average using a decay rate parameter β that is typically equals to 0.95 and has to be between 0 and 1.

$$\gamma_n = \frac{\eta}{\sqrt{a_n + \epsilon}}$$
$$a_{n+1} = \beta a_n + (1 - \beta) G_n^2$$

Where a_n is the accumulator of the squared gradients G which is going to be initialized to 0, η is a constant that we choose and ϵ is a small constant to prevent dividing by 0 if a_n.

- **AdaDelta:**

Adadelta algorithm is an extension of the RMSprop algorithm but instead of using one single accumulator, it uses two ones, one for the squared parameter updates and one for squared gradients, the formulas are as following:

$$\rho_{n+1} = \rho_n - \Delta\rho_n$$
$$a_{n+1} = \beta a_n + (1 - \beta) G_n^2$$
$$b_{n+1} = \beta b_n + (1 - \beta) \Delta\rho_n^2$$
$$\Delta\rho_n = \sqrt{\frac{b_n + \epsilon}{a_n + \epsilon}}$$

Where a is the accumulator for squared gradients and b is the accumulator for the squared parameter updates, they are both initialized to zero. β is the decay rate constant, ϵ is a small constant, and ρ is the targeted parameter.

- **FTRL:**

  The Follow-The-Regularized-Leader (FTRL) optimizer is designed for large-scale and sparse problems. It maintains separate accumulators for the gradient and the squared gradient, which are used to adjust the learning rate for each parameter. The FTRL equations are described with the equations down below:

$$\rho_{n+1} = \rho_n - \frac{a_\mathrm{n} - \sigma_n \rho_n}{\alpha}$$
$$\sigma_n = \frac{\sqrt{v_n} - \sqrt{v_{n-1}}}{\alpha}$$
$$a_{n+1} = a_n + G_n$$
$$v_{n+1} = v_n + G_n^2$$

  Where ρ is the parameter, G is the gradient, α the hyperparameter learning rate, and a_n, v_n are respectively the gradient accumulator and the squared gradient accumulator.

- **Gradient Descent with Momentum:**

  The key idea about the Gradient Descent with Momentum is the use of the exponential moving average just like RMSprop to get a straighter path to the global minima of the loss as the moving average makes a smoother path rather than taking zigzags to finally arrive to the end of the path which causes an important loss in the periods of time of training especially when working with huge datasets. As a result, we chose to use the decay rate parameter β that is typically token to be equal to 0.9, it helps the exponential moving average to give more importance to the new gradients and forget old ones after long time, this works because the β when taken at a high exponent it approaches 0. It's also important to note that the updated parameter isn't just updated with the gradient but with this moving average, The formulas are as following:

$$\rho_{n+1} = \rho_n - \eta v_n$$
$$v_{n+1} = \beta v_n + (1 - \beta) G_n$$

  Where ρ is the parameter that we're intending to modify, η is a constant, β is the decay rate average, v is the exponential moving average.

- **Adam:**

  The Adam Algorithm is nothing but a combination of RMSprop and Momentum to make the best-known optimization algorithm so far. It's all about combining the equations like down below:

$$\rho_{n+1} = \rho_n - \gamma_\mathrm{n} v_n$$
$$\gamma_\mathrm{n} = \frac{\eta}{\sqrt{\mathrm{a_n} + \epsilon}}$$
$$a_{n+1} = \beta_1 a_n + (1 - \beta_1) G_n^2$$
$$v_{n+1} = \beta_2 v_n + (1 - \beta_2) G_n$$

  Where β_1, β_2 are both decay rate constants that are typically equal respectively to 0.999 and 0.9, ϵ is a small constant, a is the accumulator, v is the moving average and ρ is our parameter.

- **AdamW:**

AdamW algorithm is an extension of the original Adam algorithm as it combines it with the concept of the weight regularization with an added weight decay term which helps prevent overfitting and encourages the model to learn more generalizable and robust representations. Here's how it works:

$$\rho_{n+1} = \rho_n - \gamma_n \hat{v}_n$$
$$\gamma_n = \frac{\eta}{\sqrt{\hat{a}_n + \epsilon}}$$
$$\hat{v}_n = \frac{v_n}{1 - \beta_1^n}$$
$$\hat{a}_n = \frac{a_n}{1 - \beta_2^n}$$
$$v_{n+1} = \beta_1 v_n + (1 - \beta_1) G_n$$
$$a_{n+1} = \beta_2 a_n + (1 - \beta_2) G_n^2$$

Where β_1, β_2 are both decay rate constants that are typically equal respectively to 0.999 and 0.9, ϵ is a small constant, a is the accumulator, v is the moving average, $\hat{v}$ and $\hat{a}$ are the first and second bias-corrected moving averages, and ρ is our parameter.

- **Adamax:**

Adamax is also an extension of the Adam algorithm. It is specifically designed to address some limitations of the original Adam algorithm, particularly when dealing with very large parameter updates. The key difference is that the Adam adapted learning rate is based on the first and second moments of the gradient while the one of Adamax is based on the first moment and the maximum absolute value of the gradients. Here's how it works:

$$\rho_{n+1} = \rho_n - \gamma_n m_n$$
$$m_n = \frac{a_n}{v_n + \epsilon}$$
$$\gamma_n = \frac{\eta}{1 - \beta_1^n}$$
$$a_{n+1} = \beta_1 a_n + (1 - \beta_1) G_n$$
$$v_{n+1} = \max(\beta_2 v_n, |G_n|)$$

Where β_1, β_2 are both decay rate constants that are typically equal respectively to 0.999 and 0.9, ϵ is a small constant, a is the accumulator, v is the infinity norm and ρ is our parameter.

- **Nadam:**

The Nesterov-Accelerated-Adaptive-Moment-Estimation (Nadam) Optimizer improves upon Adam by introducing Nesterov Momentum which helps accelerate convergence and improves optimization performance. Here's how it works:

$$\rho_{n+1} = \rho_n - \gamma_n \mu_n$$
$$\gamma_n = \frac{\eta}{\sqrt{\hat{a}_n + \epsilon}}$$
$$\mu_n = \beta_1 \hat{v}_n + (1 - \beta_1) G_n$$

$$\hat{v}_n = \frac{v_n}{1 - \beta_1^{\,n}}$$
$$\hat{a}_n = \frac{a_n}{1 - \beta_2^{\,n}}$$
$$v_{n+1} = \beta_1 v_n + (1 - \beta_1) G_n$$
$$a_{n+1} = \beta_2 a_n + (1 - \beta_2) G_n^2$$

Conclusion:

In this document we've learned how to make a Feed Forward Neural Network, how to get output out of it and how to train it to perform better and make better predictions.