Kharagpur Data Analytics Group

Associate Selection Task

Neural Networks

An Overview

Ankit Meda
22MF3IM11

Introduction

A function is a system that takes inputs and gives outputs. If you know the function, you can say with certainty that you can calculate the corresponding Y value for a particular X. But let's say we don't know the function, but we know some of the inputs and outputs the function takes and produces.

If we could **reverse engineer** the function from these values, we could use that function to calculate the Y value for an X that may not be in our data set. We could calculate the values even if there were a bit of noise or randomness in our original data. We could still capture the pattern of the data and construct a function that produces Y values that are not perfect but close enough to be helpful.

That is, we need a function approximation and, more generally, a function approximator. This is what a neural network does. Neural networks can approximate anything that can be expressed as a function, a system of inputs and outputs. That is, they are **universal function approximators**.

Assumptions

Before we start learning about the basics of Neural Networks, their working and how they make predictions, we first need to look at some assumptions we make when we make this model.

1. There exists a relation between input and output.
2. Sufficient data is present to make good predictions.
3. The Neural Network assumes that the data given shows some kind of smoothness or continuity.

While these assumptions provide a starting point, it's important to note that they are not universally applicable to all cases.

Basic Principles

In this section, we will discuss different parts which create a Neural Network and how we decide on each parameter.

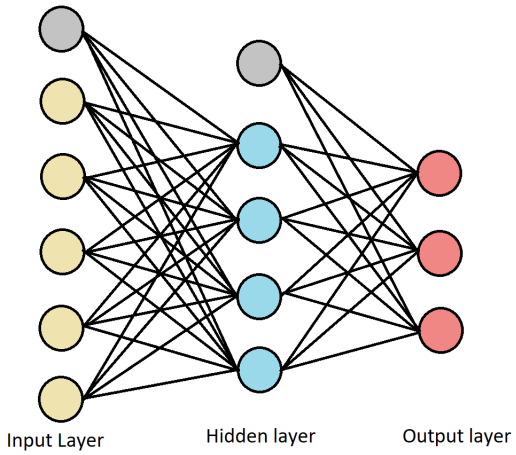

Figure 1: A simplified view of a feedforward Artificial Neural Network

The diagram shown above is a fully connected neural network. There are other types of Neural networks.

1. **Convolutional Neural Networks:** Specialised to process grid-like data, like images. They utilise convolutional layers of multiple filters or kernels that slide across the input data.

2. **Recurrent Neural Networks:** These are designed for processing sequential or time-dependent data. Unlike CNNs or fully connected NNs, these have feedback connections. This implies that information can flow forward and backwards through time, allowing information to flow recurrently. The recurrent connection allows the network to have "memory" or context, as the output of a neuron can also influence future outputs.

Neurons

From Figure 1, we observe a simple Neural Network (NN). We see some nodes and some connections. The nodes are called Neurons. The links are called weights.

The columns of neurons are called layers. There are three types of layers: i.e. Input layer, hidden layer(s) and output layer.

The input layer represents the inputs from the dataset by storing them inside. It is denoted as the 0^{th} layer.

The hidden layers are where the magic happens. It takes the incoming neuron values and passes its neuron values after processing. There can be, on paper, an infinite number of hidden layers in an NN.

The output layer is almost the same as the hidden layers, except its neuron values show the outputs and the values are not passed to the next layer. It is represented as the L^{th} layer, i.e. the NN has L layers (the input layer is not added to the number of layers).

The function of these neurons is very simple. Each neuron acts as a function that can take any number of inputs, then it performs its weighted sum (all the inputs are multiplied by weights and summed together) and gives it out as an output.

If the number of inputs is one, then it forms a simple linear line of the form

$$N(x) = wx + b$$

where w, the weight, controls the slope of the line and b, the bias weight/term, regulates the y-intercept.

We will denote the connection between the k^{th} neuron in the $(L-1)^{th}$ layer and the j^{th} neuron in the L^{th} layer as $w_{jk}^{[L]}$.

Let us configure a hypothetical NN of L layers with n neurons in the input layer and m neurons in the output layer. Let us assume that $(L-1)^{th}$ layer has n neurons and the first layer, i.e. the first hidden layer has m neurons. The notation [.] in the superscript will represent the layer number.

Now, for the neuron we discussed before, if we account for multiple inputs, then the equation becomes

$$N_1^{[1]}(x_1^{[0]}, x_2^{[0]}, .., x_n^{[0]}) = w_{(1,1)}^{[1]} x_1^{[0]} + w_{(1,2)}^{[1]} x_2^{[0]} + ... + w_{(1,n)}^{[1]} x_n^{[0]} + b_1^{[1]}$$

As we can observe, the n inputs are coming from the previous layer, which is the input layer. Hence they have the superscript [0]. The subscript in the neuron N represents its position in the layer.

Let us define a **row weight vector** as follows.

$$w_j^{[L]} = \begin{pmatrix} w_{j,1}^{[L]} & w_{j,2}^{[L]} & w_{j,3}^{[L]} & ... & w_{j,n}^{[L]} \end{pmatrix}_{1 \times n}$$

This weight vector represents the weights for the j^{th} neuron in the L^{th} layer with all the n neurons in the $(L-1)^{th}$ layer.

Let us also define a column input vector as follows.

$$A^{[L-1]} = \begin{bmatrix} x_1^{[L-1]} \\ x_2^{[L-1]} \\ .. \\ .. \\ x_n^{[L-1]} \end{bmatrix}_{n \times 1}$$

This input (or activation value) vector represents the activation values for the L^{th} layer derived from the $(L-1)^{th}$ layer.

Now by changing the neuron equation to fit the weight vector and the activation value vector, or simply, the activation vector, we get the following.

$$N_1^{[1]}(x_1^{[0]}, x_2^{[0]}, .., x_n^{[0]}) = w_1^{[1]} A^{[0]} + b_1^{[1]}$$

Or, after generalising for any neuron at position j in a hidden layer L, we get

$$N_j^{[L]}(x_1^{[L-1]}, x_2^{[L-1]}, .., x_n^{[L-1]}) = w_j^{[L]} A^{[L-1]} + b_j^{[L]}$$

We can generalise the equation to a layer by concatenating all m weight vectors, biases, and neuron values into matrices and vectors (recall that L^{th} layer has m neurons, and the one before it has n neurons).

$$W^{[L]} = \begin{bmatrix} w_1^{[L]} \\ w_2^{[L]} \\ .. \\ .. \\ w_n^{[L]} \end{bmatrix}_{m \times n}, Z^{[L]} = \begin{bmatrix} N_1^{[L]} \\ N_2^{[L]} \\ .. \\ .. \\ N_m^{[L]} \end{bmatrix}_{m \times 1}, B^{[L]} = \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ .. \\ .. \\ b_m^{[1]} \end{bmatrix}_{m \times 1}$$

Hence, we get the equation for the first layer as

$$Z^{[1]} = W^{[1]} A^{[0]} + B^{[1]}$$

Or, for the L^{th} layer

$$\boxed{Z^{[L]} = W^{[L]} A^{[L-1]} + B^{[L]}}$$

where, $Z^{[L]}$ is called the pre-activation vector of the layer L.

Weights

The connections or the weights are just numbers. They are used to break symmetry and ensure that each neuron in the network starts with a different initial state. The weights prevent the network from getting stuck in symmetric patterns during training.

Hence it is essential to initialise the weights with a strategy in mind, as they can impact the learning process and performance of the network.

Some common initialisation strategies are as follows:

1. Random initialisation: In a fully connected NN, the weights are set to random values (one of the standard approaches is using a small range with a median close to zero).

2. Xavier/Glorot initialisation: This technique allows weight initialization based on the number of inputs and outputs of each neuron.

We also need to know the number of neurons in each layer to initialise the weights. For example, in the figure above, the number of neurons from input to output are 5, 4 and 3.

Hence the first weight matrix from the input to the first hidden layer will be of type $W^{[1]}_{4\times5}$, and the final weight matrix from the hidden layer to the output layer will be of the type $W^{[2]}_{3\times4}$. This way is recommended as it results in easy computation.

Bias Neuron

A bias neuron (represented by grey-coloured neurons in Figure 1) is a special neuron that has a constant value of 1 and only has outgoing weights. The network can model a more complex relationship between input and output with bias neurons.

It aims to introduce an offset term into the network, allowing for more flexibility. Without bias neurons, the decision boundary of the network would always pass through the origin.

The bias weight linked with bias neuron is learned during the training like the other neurons, similar to the other neurons in the network. However, the learning algorithm treats the bias weights differently from the others. The bias weights are adjusted during the training to minimize the difference between predicted and desired outputs. The initial bias weights are set to **zero**, i.e. $B^{[L]} = \{0\}$ because we want to start with an **unbiased neural network**.

The **label** which comes with the data also plays an important part. If the problem statement is a binary classification, the labels are typically represented using binary format. For example, the labels can be encoded as 0 or 1, where 0 represents a class, and 1 represents another class.

If the task is to classify the inputs into more than two classes, then the label representation expands beyond the binary. The labels can be represented using **one-hot encoding**, where the labels are represented in the format like [1,0,0], [0,1,0] and [0,0,1] for three classes.

Forward Propagation

Forward propagation is the process of calculating the predicted outputs of NN based on the current weights and biases. It involves taking an unbiased model and running it by feeding the inputs through the network and obtaining the corresponding outputs.

During forward propagation, each neuron in the network receives inputs from the previous layer, performs a weighted sum of the inputs, adds a bias term, and applies an **activation function** to produce the output. This process is repeated layer by layer, starting from the input layer to the output layer.

Let us recall the general linear transformation equation for the L^{th} layer derived before

$$Z^{[L]} = W^{[L]} A^{[L-1]} + B^{[L]}$$

Here,

- $Z^{[L]}$ is the pre-activation values of the L^{th} layer
- $W^{[L]}$ is the weight matrix for the connections between the L^{th} layer and the $(L-1)^{th}$ layer
- $A^{[L-1]}$ is the activation values for the L^{th} layer derived from the $(L-1)^{th}$ layer
- $B^{[L]}$ is the bias vector for the L^{th} layer.

Let us consider the NN represented by Figure 1 for simplicity and easy understanding. It has five neurons and one bias neuron in the input layer, four neurons and one bias neuron in the hidden layer and three neurons in the output layer.

Then the linear transformation equations for layers 1 and 2 are as follows.

$$Z^{[1]}_{4\times 1} = W^{[1]}_{4\times 5} A^{[0]}_{5\times 1} + B^{[1]}_{4\times 1}$$
$$Z^{[2]}_{3\times 1} = W^{[2]}_{3\times 4} A^{[1]}_{4\times 1} + B^{[2]}_{3\times 1}$$

Upon observation, we notice that when we distribute the weights, combine them with the bias term, and pass the result as input to the output layer, the overall function becomes linear. Linear functions, when combined together, can only produce another linear function. But what if we require a more complex function than just a linear one?

Activation Functions

Activation functions serve two primary purposes. First, they introduce non-linearity, enabling the network to approximate complex relationships between inputs and outputs. Second, they help normalize output ranges, such as scaling values between 0 and 1. Non-linearities are essential to make the network learn.

Common activation functions include the **hyperbolic tangent (*tanh*)**, **sigmoid**, and **rectified linear unit (ReLU)**.

During forward propagation, we pass the pre-activation values ($Z^{[1]}$) through an activation function:

$$A^{[1]} = g(Z^{[1]})$$

Moving to the next layer, we use the output values $A^{[1]}$ from the previous layer as the "input values" and calculate the pre-activation values for the next layer

$$Z^{[2]} = W^{[2]}A^{[1]} + B^{[2]}$$

Once again, we apply the activation function:

$$A^{[2]} = g(Z^{[2]})$$

Here, $A^{[2]}_{3\times 1}$ represents the output values generated by the model in the first run (since our model only has two layers. We can observe that the dimensions of the activation vector changed to that of the pre-activation vector.

It is important to note that the activation functions for different layers in the network can differ depending on the specific requirements and characteristics of the problem to be solved.

Now we have obtained the output values from an unbiased network. How can we use these values?

Cost Function

The cost function or error function allows the network to find the difference between the given output and its correct label. Using the cost function, we can train the model to adjust the weights and biases to give a more accurate prediction in the next run by minimising the error.

Let us denote the true/desired outputs as denoted as y and the predicted outputs as $\widehat{y}$. After finding the final activation values, i.e. predicted outputs, we will find the **error** between them and the true outcomes using the cost function.

Let C be the cost function. The cost function can depend on either the outputs, the inputs, or both. The type of cost function chosen depends on the specific problem and the desired behaviour of the neural network.

Some of the common cost functions used are as follows

1. **Mean Squared Error:** Mean Squared Error is commonly used for regression problems.

2. **Binary cross-entropy:** Binary cross-entropy is used for binary classification problems, where the goal is to classify inputs into two distinct classes.

3. **Categorical Cross-Entropy:** Categorical cross-entropy is used for multi-class classification problems. It calculates the difference between the predicted class probabilities and the true one-hot encoded labels.

Now that we have figured out the output values and the error, we can now improve the performance of the network by adjusting its parameters, i.e. weights and biases, and hence reducing the error.

This process is commonly called model optimisation or training and is done through a process called backpropagation.

Backpropagation

Backpropagation of errors, or backpropagation, is a fundamental algorithm that is used to train neural networks.

It aims to reduce the error by adjusting the weights of the connections and biases, or we can say backpropagation aims to minimise the cost function by adjusting the network's weights and biases.

The process propagates or runs the error back through the network from the output layer to the input layer to calculate how much each weight and bias contributed to the overall error.

To calculate how much the weights and biases need to be changed to minimise the error, we take out the **gradient** of the cost function.

The gradient of the cost function tells how much the parameters in question need to change (in either a positive or negative direction) to minimise the cost function. That is, we need to find the gradient of the cost function with respect to the weights and biases in the layers.

Let us recall the general equation for pre-activation values

$$Z^{[L]} = W^{[L]} A^{[L-1]} + B^{[L]}$$

Assuming the cost function to be C, we need to find its gradients with respect to pre-activation values Z, weights W and biases B. Consider the number of inputs in the training dataset to be N.

Let us assume the output vector to be as follows.

$$A^{[L]} = \begin{bmatrix} \widehat{y}_1 \\ \widehat{y}_2 \\ .. \\ .. \\ \widehat{y}_j \\ .. \\ \widehat{y}_m \end{bmatrix}_{m \times 1}$$

Recall that we modelled a NN with n neurons in the input layer and m neurons in the output layer, and the $(L-1)^{th}$ layer has n neurons, and the first layer, i.e. the first hidden layer has m neurons.

In the L^{th} Layer

Recall that we assumed the connection/weight between the k^{th} neuron in the $(L-1)^{th}$ layer and the j^{th} neuron in the next L^{th} layer as $w_{jk}^{[L]}$.

We will now focus on finding the gradient of the cost function with respect to a particular weight and bias in the L^{th} layer.

Let C_{1*} represent the "cost" given out by the cost function when the first sample is forward propagated through the network. Hence, C_{i*} represents the cost for the i^{th} sample from the dataset consisting of N samples.

As we discussed before, the cost function depends on either the inputs, the outputs, or both, which means that we can take the gradient of the cost function with respect to $A^{[L]}$, the output vector.

By applying the chain rule to the gradient, we obtain the following equation.

$$1)\ \frac{\partial C_{1*}}{\partial w_{jk}^{[L]}} = \frac{\partial Z_j^{[L]}}{\partial w_{jk}^{[L]}} \cdot \frac{\partial A_j^{[L]}}{\partial Z_j^{[L]}} \cdot \frac{\partial C_{1*}}{\partial A_j^{[L]}}$$

Let us now discuss each term in detail.

1. $$\frac{\partial C_{1*}}{\partial A_j^{[L]}}$$

   Here, $A_j^{[L]} = \widehat{y}_j$ which is a parameter in C, the cost function. Hence the gradient of the cost function can be computed with respect to $A_j^{[L]}$ without any issues. For the sake of understanding, let us take the cost function to be **Square Mean Error**. Hence the gradient turns out as follows

   $$C_{1*} = \frac{(\widehat{y}_1 - y_1)^2 + (\widehat{y}_2 - y_2)^2 + .. + (\widehat{y}_j - y_j)^2 + .. + (\widehat{y}_m - y_m)^2}{M}$$

   $$\boxed{\frac{\partial C_{1*}}{\partial A_j^{[L]}} = \frac{2}{M}(\widehat{y}_j - y_j)}$$

2. $$\frac{\partial A_j^{[L]}}{\partial Z_j^{[L]}}$$

   We can calculate this part by recalling $A_j^{[L]} = g(Z_j^{[L]})$. Hence we find the gradient to be as follows

   $$\boxed{\frac{\partial A_j^{[L]}}{\partial Z_j^{[L]}} = g'(Z_j^{[L]})}$$

   where g is the activation function, as discussed in the sections before.

3.

$$\frac{\partial Z_j^{[L]}}{\partial w_{jk}^{[L]}}$$

By recalling the general equation for a neuron, we get the gradient as follows

$$Z_j^{[L]} = N_j^{[L]} = w_{(j,1)}^{[L]} x_1^{[L-1]} + w_{(j,2)}^{[L]} x_2^{[L-1]} + ... + w_{(j,k)}^{[L]} x_k^{[L-1]} + ... + w_{(j,n)}^{[L]} x_n^{[L-1]} + b_j^{[L]}$$

$$\boxed{\frac{\partial Z_j^{[L]}}{\partial w_{jk}^{[L]}} = x_k^{[L-1]}}$$

Therefore, we get the required gradient with respect to $w_{jk}^{[L]}$ as follows

$$\frac{\partial C_{1*}}{\partial w_{jk}^{[L]}} = \frac{\partial Z_j^{[L]}}{\partial w_{jk}^{[L]}} \cdot \frac{\partial A_j^{[L]}}{\partial Z_j^{[L]}} \cdot \frac{\partial C_{1*}}{\partial A_j^{[L]}}$$

$$\boxed{\frac{\partial C_{1*}}{\partial w_{jk}^{[L]}} = x_k^{[L-1]} \cdot g'(Z_j^{[L]}) \cdot \frac{2}{M}(\widehat{y}_j - y_j)}$$

Now, we do the same for the gradient of the cost function with respect to the bias term.

$$2) \ \frac{\partial C_{1*}}{\partial b_j^{[L]}} = \frac{\partial Z_j^{[L]}}{\partial b_j^{[L]}} \cdot \frac{\partial A_j^{[L]}}{\partial Z_j^{[L]}} \cdot \frac{\partial C_{1*}}{\partial A_j^{[L]}}$$

Since two of the three terms are identical for the gradient with respect to the weight and bias term, we will only focus on the first term, which is yet to be defined.

1.

$$\frac{\partial Z_j^{[L]}}{\partial b_j^{[L]}}$$

By recalling the general equation for a neuron, we get the gradient as follows

$$Z_j^{[L]} = N_j^{[L]} = w_{(j,1)}^{[L]} x_1^{[L-1]} + w_{(j,2)}^{[L]} x_2^{[L-1]} + ... + w_{(j,k)}^{[L]} x_k^{[L-1]} + ... + w_{(j,n)}^{[L]} x_n^{[L-1]} + b_j^{[L]}$$

$$\boxed{\frac{\partial Z_j^{[L]}}{\partial b_j^{[L]}} = 1}$$

Therefore, we get the required gradient with respect to $b_j^{[L]}$ as follows

$$\frac{\partial C_{1*}}{\partial b_j^{[L]}} = \frac{\partial Z_j^{[L]}}{\partial b_j^{[L]}} \cdot \frac{\partial A_j^{[L]}}{\partial Z_j^{[L]}} \cdot \frac{\partial C_{1*}}{\partial A_j^{[L]}}$$

$$\boxed{\frac{\partial C_{1*}}{\partial b_j^{[L]}} = 1 \cdot g'(Z_j^{[L]}) \cdot \frac{2}{M}(\widehat{y}_j - y_j)}$$

We have found the gradients with respect to a weight and a bias term in the L^{th} layer for one single sample from the database containing N samples.

To find the average cost for the weight and the bias term, we add up all the costs from each sample as follows.

$$\frac{\partial C}{\partial w_{jk}^{[L]}} = \frac{1}{N}\sum_{i=1}^{N}\frac{\partial C_{i*}}{\partial w_{jk}^{[L]}}$$

$$\frac{\partial C}{\partial b_{j}^{[L]}} = \frac{1}{N}\sum_{i=1}^{N}\frac{\partial C_{i*}}{\partial b_{j}^{[L]}}$$

Now that we have understood the general procedure to find the cost for a weight and a bias term, we can apply the weight matrices and bias vectors instead of individually finding the gradients for each weight and bias term to make implementation easier and computations faster.

1. $$\frac{\partial C_{1*}}{\partial A^{[L]}} = \frac{2}{M}(\widehat{y} - y)$$

   where $\widehat{y} = A^{[L]}$ and y are the predicted output vector and true/desired output vector.

2. $$\frac{\partial A^{[L]}}{\partial Z^{[L]}} = g'(Z^{[L]})$$

   where $A^{[L]}$ and $Z^{[L]}$ are the predicted output vector and pre-activation vector for L^{th} layer.

3. $$\frac{Z^{[L]}}{W^{[L]}} = A^{[L-1]}$$

   where $W^{[L]}$ represents the weight matrix for the L^{th} layer.

4. $$\frac{Z^{[L]}}{B^{[L]}} = \overrightarrow{1}_{m\times 1}$$

   where, $\overrightarrow{1} = (1, 1, .., 1)^T$ is called "vector of ones".

Hence, the final equations are as follows

1. For the weight matrix, we get

$$\frac{\partial C_{1*}}{\partial W^{[L]}} = \frac{\partial Z^{[L]}}{\partial W^{[L]}} \cdot \frac{\partial A^{[L]}}{\partial Z^{[L]}} \cdot \frac{\partial C_{1*}}{\partial A^{[L]}}$$

$$\frac{\partial C_{1*}}{\partial B^{[L]}} = A^{[L-1]} \cdot g'(Z^{[L]}) \cdot \frac{2}{M}(\widehat{y} - y)$$

2. For bias vector we get

$$\frac{\partial C_{1*}}{\partial B^{[L]}} = \frac{\partial Z^{[L]}}{\partial B^{[L]}} \cdot \frac{\partial A^{[L]}}{\partial Z^{[L]}} \cdot \frac{\partial C_{1*}}{\partial A^{[L]}}$$

$$\frac{\partial C_{1*}}{\partial B^{[L]}} = \overrightarrow{1} \cdot g'(Z^{[L]}) \cdot \frac{2}{M}(\widehat{y} - y)$$

Now, we can average the gradients of the cost from all the samples from the dataset for a layer as follows.

$$\frac{\partial C}{\partial W^{[L]}} = \frac{1}{N} \sum_{i=1}^{N} \frac{\partial C_{i*}}{\partial W^{[L]}}$$

$$\frac{\partial C}{\partial B^{[L]}} = \frac{1}{N} \sum_{i=1}^{N} \frac{\partial C_{i*}}{\partial B^{[L]}}$$

In the $(L-1)^{th}$ Layer and beyond

Until now, we have discussed the costs for individual weights and biases in the L^{th} layer and weights and biases in the whole L^{th} layer. We will now discuss the costs for the layers from $(L-1)^{th}$ to the first layer.

Let us consider the $(L-1)^{th}$ layer. Then the gradient of the cost function with respect to its weight matrix and bias vector is as follows

1. $$\frac{\partial A^{[L]}}{\partial Z^{[L]}} = g'(Z^{[L]})$$

2. $$\frac{\partial Z^{[L]}}{\partial A^{[L-1]}} = W^{[L]}$$

3. $$\frac{\partial A^{[L-1]}}{\partial Z^{[L-1]}} = g'(Z^{[L-1]})$$

4. $$\frac{\partial Z^{[L-1]}}{\partial W^{[L-1]}} = A^{[L-1]}$$

5. $$\frac{\partial Z^{[L-1]}}{\partial B^{[L-1]}} = \overrightarrow{1}_{n \times 1}$$

   Here, $n \times 1$ because the $(L-1)^{th}$ consists of n neurons.

Now that we have defined all the terms, we can find the gradient of the cost function for the $(L-1)^{th}$ layer.

1. For the weight matrix

$$\frac{\partial C_{1*}}{\partial W^{[L-1]}} = \frac{\partial Z^{[L-1]}}{\partial W^{[L-1]}} \cdot \frac{\partial A^{[L-1]}}{\partial Z^{[L-1]}} \cdot \frac{\partial Z^{[L]}}{\partial A^{[L-1]}} \cdot \frac{\partial A^{[L]}}{\partial Z^{[L]}} \cdot \frac{\partial C_{1*}}{\partial A^{[L]}}$$

$$\boxed{\frac{\partial C_{1*}}{\partial W^{[L-1]}} = A^{[L-1]} \cdot g'(Z^{[L-1]}) \cdot W^{[L]} \cdot g'(Z^{[L]}) \cdot \frac{2}{M}(\widehat{y} - y)}$$

2. For the bias vector

$$\frac{\partial C_{1*}}{\partial B^{[L-1]}} = \frac{\partial Z^{[L-1]}}{\partial B^{[L-1]}} \cdot \frac{\partial A^{[L-1]}}{\partial Z^{[L-1]}} \cdot \frac{\partial Z^{[L]}}{\partial A^{[L-1]}} \cdot \frac{\partial A^{[L]}}{\partial Z^{[L]}} \cdot \frac{\partial C_{1*}}{\partial A^{[L]}}$$

$$\boxed{\frac{\partial C_{1*}}{\partial B^{[L-1]}} = \overrightarrow{1} \cdot g'(Z^{[L-1]}) \cdot W^{[L]} \cdot g'(Z^{[L]}) \cdot \frac{2}{M}(\widehat{y} - y)}$$

We have calculated the gradients for one sample from the dataset consisting of N samples. Then the average cost gradients would be as follows.

$$\frac{\partial C}{\partial W^{[L-1]}} = \frac{1}{N} \sum_{i=1}^{N} \frac{\partial C_{i*}}{\partial W^{[L-1]}}$$

$$\frac{\partial C}{\partial B^{[L-1]}} = \frac{1}{N} \sum_{i=1}^{N} \frac{\partial C_{i*}}{\partial B^{[L-1]}}$$

Hence, in this way, we can apply the chain rule for all the layers from $L-1$ to the first layer and find the average cost gradients. Now that we have found the required gradients, we need to update the weights and biases such that the error function is minimized.

Weight and Bias Optimisation

Using the computed gradients, we can update the weights and biases of the network.

Hyperparameter:

A hyperparameter is a setting that is determined before the training process and remains constant during training. Unlike model parameters (weights, biases) learned from the training data, hyperparameters are set manually.

Finding the appropriate hyperparameter values is essential as they impact the model's performance and generalisation.

The learning rate is an example of a hyperparameter. Let us denote the learning rate as α.

$$W_{new}^{[L]} = W_{old}^{[L]} - \alpha \cdot \frac{\partial C}{\partial W_{old}^{[L]}}$$

$$B_{new}^{[L]} = B_{old}^{[L]} - \alpha \cdot \frac{\partial C}{\partial B_{old}^{[L]}}$$

$$W_{new}^{[L-1]} = W_{old}^{[L-1]} - \alpha \cdot \frac{\partial C}{\partial W_{old}^{[L-1]}}$$

$$B_{new}^{[L-1]} = B_{old}^{[L-1]} - \alpha \cdot \frac{\partial C}{\partial B_{old}^{[L-1]}}$$

$$..$$

$$..$$

$$W_{new}^{[1]} = W_{old}^{[1]} - \alpha \cdot \frac{\partial C}{\partial W_{old}^{[1]}}$$

$$B_{new}^{[1]} = B_{old}^{[1]} - \alpha \cdot \frac{\partial C}{\partial B_{old}^{[1]}}$$

The learning rate α determines the step size in the parameter update, controlling how fast the network learns.

Epochs

Now, the weights and biases are tuned to mitigate the error. We run the same training dataset multiple times to fine-tune the weights and biases.

The number of epochs represents the times the model goes over the training dataset to update the weights and performance. The relationship between the number of epochs and model performance can be complicated.

Increasing the number of epochs allows the model to learn more from the training data, potentially improving accuracy. However, there is a point where the model starts to memorise the training data instead of learning general patterns, leading to overfitting.

This issue also comes with the number of neurons in the hidden layers. A general guideline is to keep the neurons in a hidden layer less than $2\times$ the number of input neurons. A commonly used starting number is $\frac{2}{3}$ of the input neurons + output neuron number.

However, the optimal number of neurons in a hidden layer can vary depending on the specific problem and dataset. Finding the right balance between model complexity and performance may require experimentation and fine-tuning.

We have completed the training of our model Neural Network. We can pass the test dataset through the input layers to test the model. The neural network will process the incoming data with fine-tuned weights and biases and intelligently predict the output.