**Neural Networks**

Table of Contents

[Activation Functions 5](#_Toc106494587)

[Two-Class Classification 5](#_Toc106494588)

[Single Neuron 5](#_Toc106494589)

[Multiple Neurons 6](#_Toc106494590)

[Multiple Samples 8](#_Toc106494591)

[The Need for Activation Functions 9](#_Toc106494592)

[Multi-Class Classification 11](#_Toc106494593)

[Gradient Descent 11](#_Toc106494594)

[Forward Propagation 12](#_Toc106494595)

[Backward Propagation for Logistic Regression 13](#_Toc106494596)

[Backward Propagation for Neural Networks 14](#_Toc106494597)

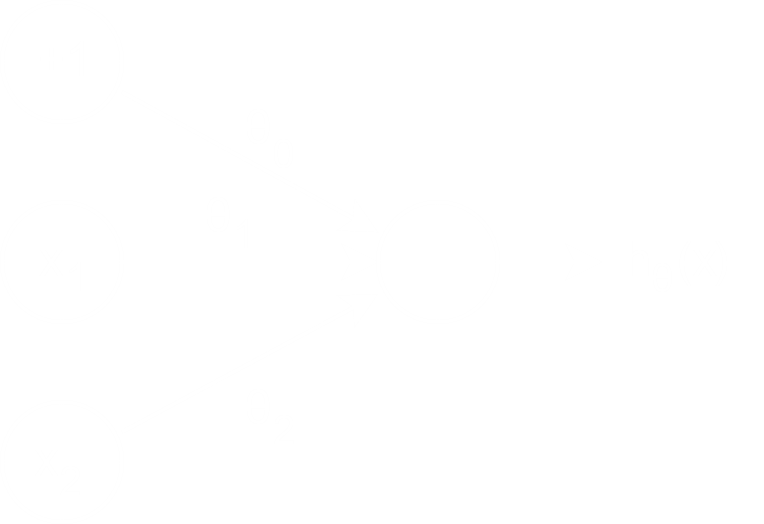
[Random Weight Initialization 17](#_Toc106494598)

One of the major applications of machine learning is in the field of computer vision. Consider a situation where we need to classify whether an image contains a car or not. This is an example of binary classification.

For classification problems, we used **logistic regression**, where the hypothesis was , where , a vector containing the feature values. If we have a 50x50 grey-scale image, we have a total of 2500 pixels. Each pixel represents a separate feature, so we have 2500 features. If we use an RGB image, then there are 3 pixels (one for each colour) for each of the original pixels, so a total of 7500 pixels. As you can probably tell, it becomes insanely difficult to deal with this many features if we try to use logistic regression. And this is for a tiny image. Images from even common smartphones nowadays take pictures of several million pixels.

To deal with problems like these, which have more features than we can deal with, we use **neural networks**. Neural networks were designed to mimic the human brain. They consist of multiple **neurons**. Each neuron takes an input and gives an output. Several neurons connect to each other to form the complete network.

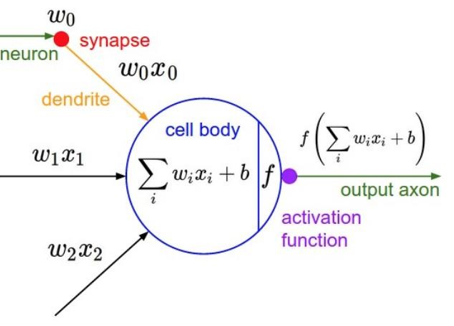
Some examples should make this clear. Suppose we have a single neuron that we want to use as a logical AND gate. An AND gate takes two inputs and gives one output.



Here, we have an extra input, which is just the value . This represents . For each input, we multiply it by some value . Thus, this is somewhat like the linear regression problem that we have seen previously. .

If we now set the values of , and to , and respectively, we will find that the output will act exactly like an AND gate. We will get a positive value only when both and are and a negative value otherwise.

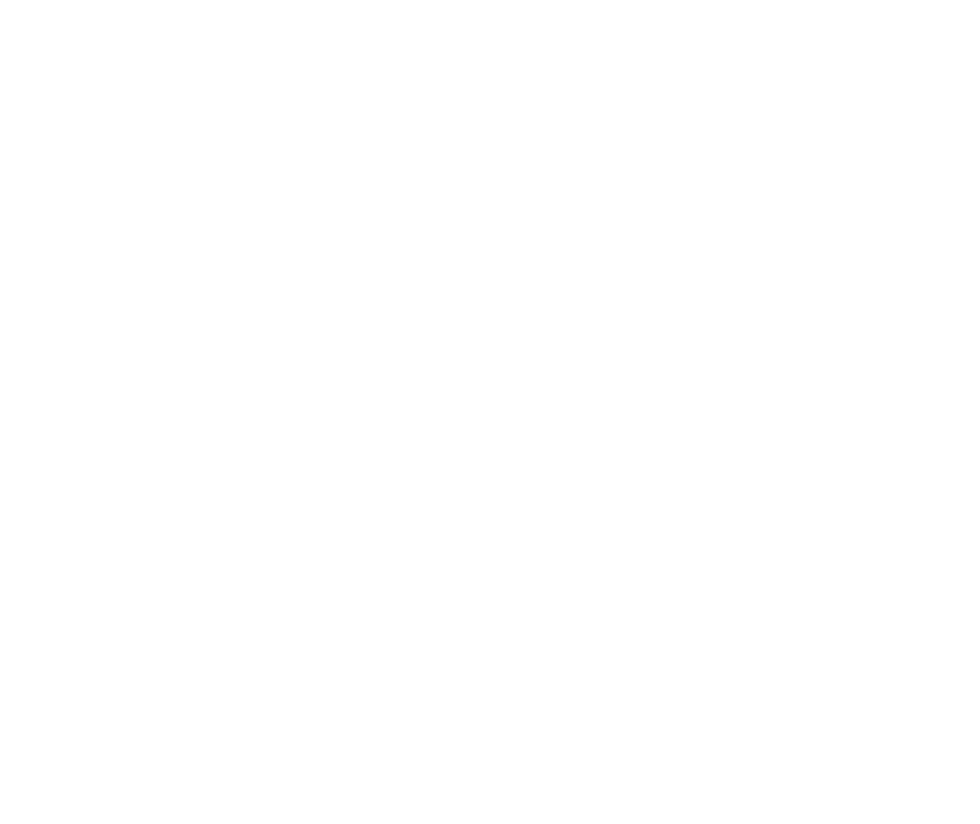
However, an AND gate is rarely if ever used alone. More likely, it is computing just a small part of a bigger problem, consisting of many such logic gates. The neurons in a neural network essentially do exactly this. They compute the result of one part of the larger problem. The problems we will be seeing involve each neuron performing logistic regression, as below. Thus, the entire neural network will involve many iterations of logistic regression.



Note: Variable names in image are different.

Similarly, we can design OR gates, NOT gates and so on using single neurons. The behaviour of a single neuron thus depends on the **parameter values**, since each neuron gets the same inputs.

However, what if we want to design an XNOR gate. You will find that it is not possible to design this using a single neuron. Even in real life, it requires combining two logic gates, AND and NOR. These in turn go to an OR gate, which gives the final output.



We can also arrange neurons in a neural network like this. There is an **input layer**, consisting of all the input features. All of these features go to every neuron in the second layer. This layer is called the **hidden layer**, because we never see the outputs for the neurons in this layer. We can have **multiple hidden layers**. Finally, we get to the **output layer**, which consists of just one neuron in this case.

The more layers we have in our neural network, the **deeper** it is said to be. How deep a neural network should be or how many neurons there should be in each layer are design decisions.

## Activation Functions

So far, we have been using the **sigmoid function** in logistic regression.

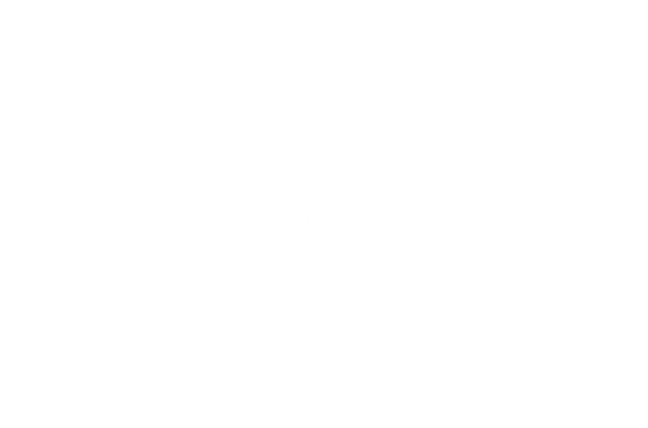
This is called an **activation function**. It takes a linear parameter and turns it into a non-linear output.

There can be several types of activation functions, which results in different performances for the overall model. Some of the common ones are:

* Sigmoid
* Hyperbolic Tangent
* Rectified Linear Unit (ReLU)
* Leaky ReLU

## Two-Class Classification

### Single Neuron

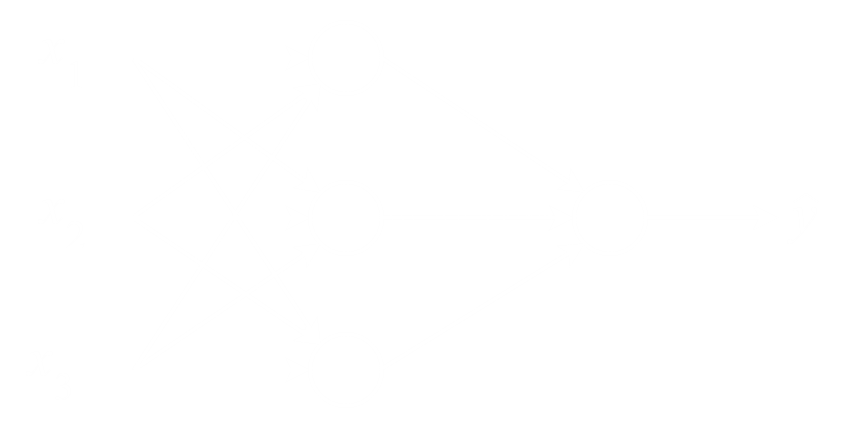


In a neural network, the task of a **single neuron** is to perform **logistic regression** on its inputs. Thus, for the neural network above, the neuron gives us two things:

Here, and . The result of the first equation gives us .

### Multiple Neurons

If we now ramp this up to include a **single hidden layer**, the neural network will look like this:



For each of the nodes in the hidden layer:

We get three separate **numerical values** here. If we want to consider the output for the **entire hidden layer** at once, we can have a vector like this:

Here, and .

The **superscripts** in the squared brackets indicate the **layer number**. For the weights, the **subscripts** indicate the **neuron number** and the **weight number** respectively.

Similarly, considering the second layer:

The output for our neural network is .

For the first layer, if we simply denote as , then the outcomes from each layer can be generalized as:

The network we have created here is called a **Feed-Forward Neural Network**, because the output for one layer is fed forward as the next layer’s input. It is also a **fully connected network**, since all the inputs for a layer are connected to all the neurons in the layer.

The number of hidden layers and the number of neurons in each layer are both **hyperparameters** for our system. If we have more layers, we will be able to design more complex systems.

### Multiple Samples

In the above example, we have considered just one sample. If we consider all samples, each with features,

This is an matrix.

Even when we consider this, the weight matrix does not change. It remains an matrix, where is the number of neurons in the layer. We simply denote it as .

The vector is interesting though. The same vector is repeated times for the examples. This is in fact done automatically during implementation using **Python broadcasting**.

As a result of all this, also becomes a matrix, with each column representing the outputs for all the neurons in the layer for each of the samples.

This is an matrix.

Finally, we have .

Thus, for th layer,

### The Need for Activation Functions

We keep using , the **activation function**, but why do we need this at all? Why not just directly use in the next layer?

Consider a two-layer examples:

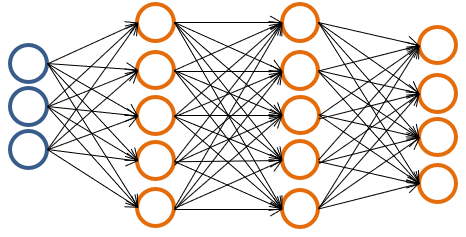
Now consider that we are using a **linear activation function**, i.e. . If we do this, we can directly write this:

Thus, regardless of how many layers we have in the model, we will end up with a single equation output, as though we have just one layer. However, the entire purpose of having more layers was to model more complex non-linear functions. Thus, if we do not use a non-linear activation function, our entire model will become linear.

The exact activation functions we use in different layers may be different. It is acceptable to use say a sigmoid function in one layer and a ReLU function in another. In this case, the activation function for a particular layer can be denoted as .

## Multi-Class Classification

The only difference between a single-class classification neural network and a multi-class classification neural network is that the output layer will have **multiple neurons**.



The output for this layer may look like this, for example:

Thus, since the output for the fourth neuron is the highest, we will say that the correct class is the fourth class.

## Gradient Descent

In gradient descent, there are two steps,forward propagation and backward propagation.

### Forward Propagation

We have already seen the **forward propagation** section.

For a two-layer example, , the output. For some value of the weights, we have gotten an output. From this, we can calculate the difference between the ground truth and our current output, also called the **loss**, .

This comes directly from our previous discussion on **logistic regression**. Once we have the loss, we can also find the **cost**, .

If we have **multiple outputs** instead of just a single one, we need to find the loss as the sum of the errors of all possible outputs.

### Backward Propagation for Logistic Regression

The reason we went through this trouble was to find the **cost** at the end. This enables us to determine what the **gradients** with respect to the parameters are and thus update our parameters.

For logistic regression:

Thus,

In the above equations, is the result of computing for .

Now, repeatedly writing the derivatives as fractions is troublesome, so we will often reduce the derivative notation to a simple for the example above, e.g. . We may similarly use , and so on.

Thus,

Since we have the derivates, we can update our parameters.

The process of finding derivatives using gradients in a backwards manner as we have done above is called **backward propagation**.

### Backward Propagation for Neural Networks

In a similar manner to how we performed backward propagation for logistic regression, we can perform backward propagation for **neural networks**. For a two-layer neural network:

(element-wise product)

(element-wise product)

Generally, for the last layer (layer 2 in this case), we tend to use a **sigmoid function** as the activation function. Thus,

This allows us to simplify as

For both and , we need to add an extra , since we are considering all examples. Remember that and we update as . Thus, we need to take the into account.

Thus,

Similarly, we also need to average over all examples when updating .

The extra parameters in the brackets are for the Python implementation. They ensure that the average is happening column-wise and that the dimensions are not broken.

Thus, in summary:

For the last layer, ,

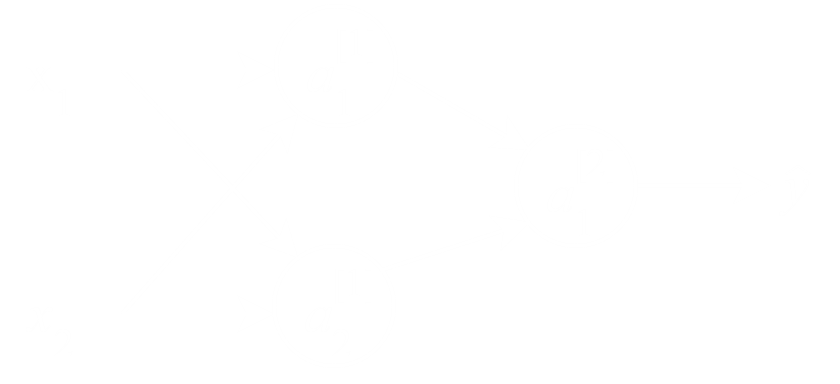
In general, for any layer ,

(element-wise product)

## Random Weight Initialization

It is important that we **initialize** our parameters **randomly** instead of initializing them to zero. More specifically, the weight matrix, , cannot be initialized to or to the same value. This will cause all the hidden units in each hidden layer to compute the **same value**, which removes the benefits of using a neural network.

Suppose we have the following neural network:



If all the weights are set to 0, then . Again, since the all the weights are , we will find that the derivatives for all the values will be the same. This will result in the values being updated to the same values. This value may be non-zero, but will definitely be the same. This will continue in the further iterations.

All the weights being the same will cause us to be unable to compute different features. We are only seeing one.

The solution is to initialize the weight matrix randomly. For the case where we have 2 hidden units in the first hidden layer and 3 input values,

We multiplied by to get random values that are close to to start with, but are not identical. If is large, will be large, which will cause the activation function to be at a point where the gradient is small for those values of (assuming we are using the sigmoid function or the hyperbolic tangent function). This will cause the gradient descent algorithm to run slowly.