# Deep Learning

# LVC 1: Introduction to Deep Learning

## Deep Learning

- Deep Learning is an important part of Machine Learning concerned with various algorithms which are inspired by the structure and function of the brain, called artificial neural networks.
- Deep Learning is considered over traditional machine learning in many instances. The main reason for this is that deep learning algorithms try to learn and process **very high** levels of features from data and importantly, **unstructured data**, such as images, text, and audio. This is one of the greatest advantages of deep learning over machine learning.

## Deep Learning: Reasons for Success

- **Lots of Data:**

  The standard principle in data science is that “more training data leads to better models".

  Due to the presence of vast amounts of data, better historic trends can be examined and better predictions can be achieved.

- **Computational resources**:

  Due to vast improvements in technology and greater improvements in computational power like higher-end CPUs (Central Processing Units), GPUs (Graphical Processing Units) and storage devices, deep learning models have a wider spectrum of chances to train on larger data sets more efficiently and on large scales.

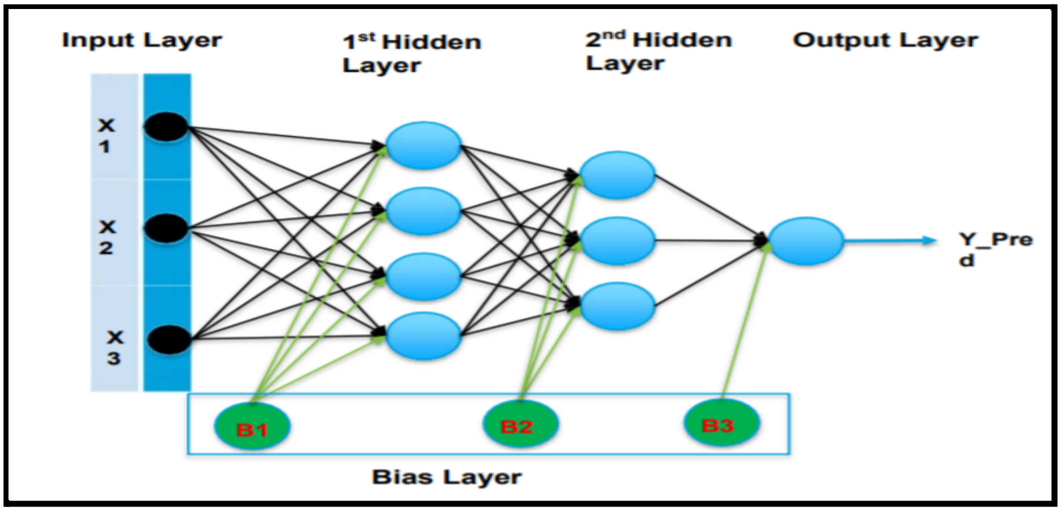
- **Large models are easier to train**:

  As we will observe, with the advent of deep learning models, the architectures have become very deep and large with several layers (hence giving rise to the term “deep” learning). With high-end GPUs and the ability to buy or rent computing power on the cloud, these models, with gradient-based approaches and batch size variations, have become a lot easier to train.

- **Flexible "lego style" architectures**:

  One very convenient aspect of deep learning models is that they are modular and easy to change. New architectures can easily be created by simply reusing existing models and just changing a few aspects such as the number of layers, the number of neurons, the activation or loss functions, and other hyperparametric choices that are decided by the user.

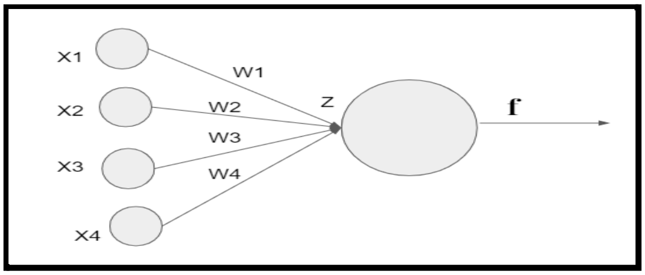
## Feedforward Neural Networks

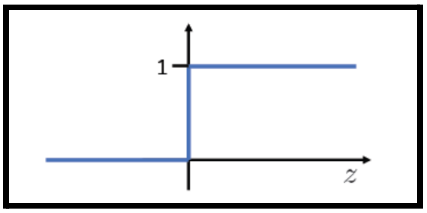


- The input layer is passive, does no processing, and only holds the input data to supply it to the **first hidden** layer. It is a vector that describes the features of the input.
- The layers that are not input or output layers are called the **Hidden Layers** of the neural network. Each neuron in the first hidden layer takes all the input attributes, **multiplies** them with the corresponding **weights**, adds a **bias**, and the output is transformed using a **nonlinear function**.
- The weights for a given hidden neuron are randomly initialized and all the neurons in the hidden layer will have weights associated with them.
- The output of each neuron is fed to output layer nodes or another set of hidden nodes in another hidden layer.
- The output value of each hidden neuron is sent to each output node in the output layer. The output for a classification problem will be (**YES/NO**) and for a regression problem, it will be a **real number**.

Let's understand the operations happening inside a single node or neuron:

## A Unit in a Neural Network



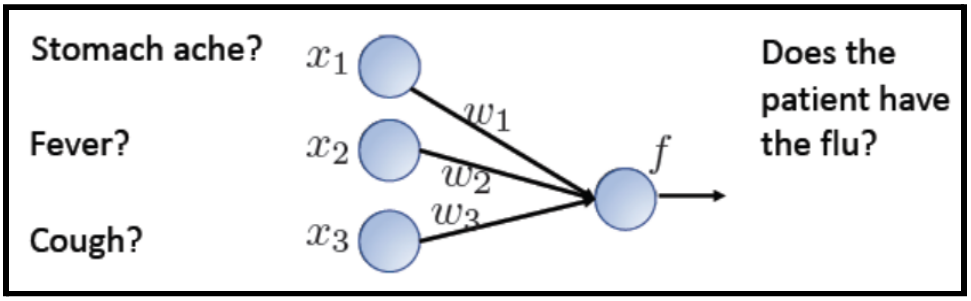


- A neuron in a neural network works in three steps:
    1. First, it multiplies the input signals with their corresponding weights.
    2. Second, it adds up the above products (obtaining a weighted sum of the inputs).
    3. Lastly, it applies an activation function on this weighted sum.
- So the input to a neuron is a **summation** of the products of the **input values** and their respective **weights**.

$$Z = X_1 * W_1 + X_2 * W_2 + X_3 * W_3 + X_4 * W4$$

- If the weight is **zero**, the respective input will obviously be **ignored** by the neuron, as it's being multiplied by zero (hence giving zero) and will not contribute to the weighted sum.
- In one kind of neuron (seen in early historical implementations of neural networks), the "activation function" is simply a threshold function, with a certain threshold value (seen in the diagram above on the right side) The function would simply output 1 if the input was greater than or equal to the threshold value; otherwise it would output 0.
- The summation of inputs and weights from earlier gives us a **score** - so in the above kind of neuron, if this score is **greater** than the threshold it will output **1**, otherwise, it outputs **0.**
- The above **operation** is similar to the **Linear Classifier** operation, where it takes the weighted combination of inputs and compares it with a **threshold.** The parameters of the linear classifiers are these **weights** and **biases.**
- So, a neuron in a neural network can be seen to work as a linear classifier of the inputs. There are many linear classifiers (neurons) in the hidden layers which have different weights, and each neuron **detects** a different **pattern** but is essentially just a kind of linear classifier.

# More Intuition: A Simple Example



- Now let's say we have three inputs - Stomach ache ($X_1$), Fever ($X_2$), and Cough ($X_3$), and the output we're trying to obtain is: **Does the patient have the flu or not?**
- Let's assume that the inputs are in **1**'s and **0**'s, **1 - Yes,** and **0 - No.**
- Stomach ache is not usually a symptom of flu, so the respective **weight** could be **decreased** (negative number), fever and cough do tend to be **symptoms** of flu so the corresponding weights for those two inputs could be set high to begin with (positive numbers).
- Let's assume this neuron has a threshold activation function with a **threshold value of 5.**
- For this example, consider the below observations with three features $X_1$, $X_2$, and $X_3$:

| | **Input** | **Value** | **Weights** |
|---|---|---|---|
| X1 | Stomach Ache | YES (1) | -1 |
| X2 | Fever | NO (0) | 3 |
| X3 | Cough | YES (1) | 3 |

Let's calculate the equation for these inputs and weights.

$$X_1 * W_1 + X_2 * W_2 + X_3 * W_3$$
$$= 1 *- 1 + 0 * 3 + 1 * 3$$
$$= - 1 + 0 + 3$$
$$= 2 < 5$$

Since the output of the neuron is less than the threshold, the threshold activation function will fire 0, or in other words, it predicts that the given person is not suffering from flu.

Let's consider another set of observations with three features $X_1$, $X_2$, and $X_3$:

| | Input | Value | Weights |
|---|---|---|---|
| X1 | Stomach Ache | NO (0) | -1 |
| X2 | Fever | YES (1) | 3 |
| X3 | Cough | YES (1) | 3 |

$$X_1 * W_1 + X_2 * W_2 + X_3 * W_3$$
$$= 0 *- 1 + 1 * 3 + 1 * 3$$
$$= 0 + 3 + 3$$
$$= 6 > 5$$

After calculating the linear expression, the output here is 6, which is greater than the threshold value of 5. That means the neuron will fire 1, or in other words, it predicts that this person is in fact suffering from the flu.

**NOTE:** The **weights** and **thresholds** chosen in the example are randomly selected for explanation purposes.

For the above example, we used a threshold activation function for the neuron (also called the Step Function), where it uses a threshold value to determine the output. If the output is greater than the threshold, this neuron fires 1 and if smaller than the threshold, 0 is fired.

However, the machine learning research community soon came to understand this was not necessarily the best way to determine the weights of a neural network and make predictions. One of the drawbacks is the sharp change from 0 to 1 at a specific threshold point value - this is a function that is not mathematically differentiable, and may not always be appropriate for real-world decision making. Rather, a smoother function that outputs, for example, the probability of someone having the flu, before we decide on a threshold, maybe a more appropriate activation function for a neuron.

Let’s take a look at some of these smoother activation functions that have recently found more prominence in Deep Learning.

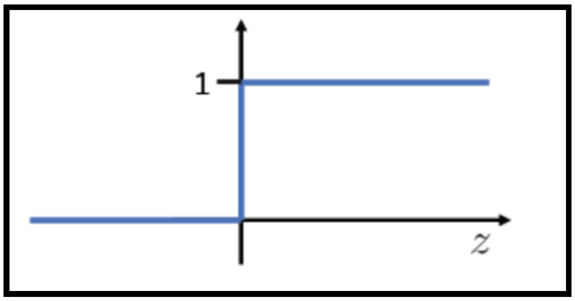
# Activation Functions

The activation function decides whether a neuron should be “activated” or not by calculating the weighted sum, and also further adding a bias term to it (different from the purely weighted sum example we saw earlier). The purpose of the activation function is to **introduce non-linearity** into the output of a neuron. The activation function is critical to the overall functioning of the neural network. Without it, the whole neural network will mathematically become equivalent to one single neuron which is just a linear combination of the input features. An activation function is hence one of the critical components that give neural networks the ability to deal with complex problems.

**Some of the common types of Activation Functions**:

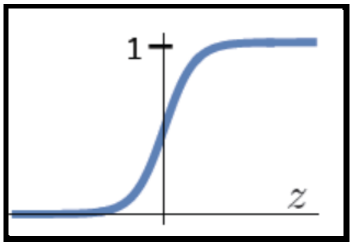
1. The Binary Step Function (which we have seen earlier)
2. The Sigmoid Function
3. The Tanh Function
4. The ReLU Function

- **The Binary Step Function**: This activation function is a basic threshold-based classifier. In this, we use some threshold value to decide whether the neuron should be **activated** or **not**. In the figure below, the threshold value has been set to **0.**



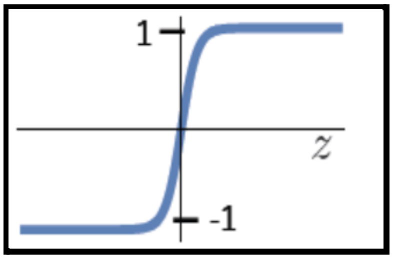
$$\boxed{f(z) = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{otherwise} \end{cases}}$$

- **The Sigmoid Function**: The Sigmoid Function is the most common activation function used in the **output** layer for **binary** classification problems. The main reason why Sigmoid is used is that it gives a smooth output between **0** and **1,** which can be interpreted as a likelihood or probability of an outcome happening (Ex: having the flu). Therefore, it is especially used for models where we have to **predict the probability** as an output. Usually, it is used in the output layer of a binary classification, where the result is either 0 or 1, so the result can be predicted **1** if a value is greater than **0.5** and **0** otherwise.



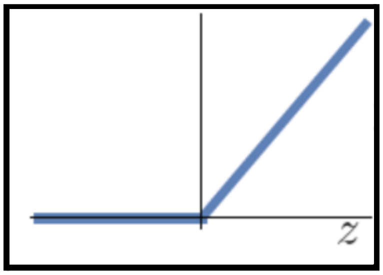
$$\boxed{f = \frac{e^z}{1 + e^z}}$$

- **The Tanh Function:** The Tanh Function **(Tangent Hyperbolic Function)** is a mathematically shifted version of the Sigmoid function. Its output values lie between **-1 to 1,** and it is usually preferred over the Sigmoid function for the hidden layers of neural networks.



$$f = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

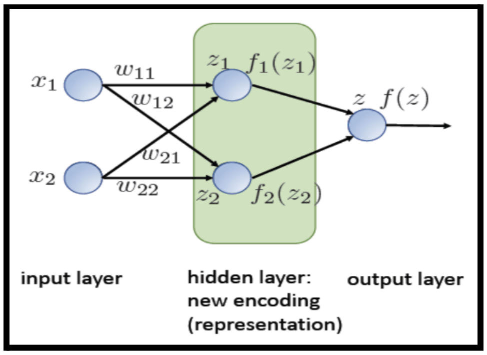
- **ReLU (Rectified Linear Unit)**: ReLU is **the most** widely used activation function in Deep Learning today. Chiefly implemented in the **hidden layers** of the neural networks, it ranges from **0** to ∞. One of ReLU's advantages is that it is **less computationally expensive** than Tanh and Sigmoid because it involves a much **simpler** mathematical operation.



$$\boxed{f = Max(0, Z)}$$

Every activation function has its advantages and disadvantages, and selecting the type of activation function to use is a hyperparameter which depends on the problem statement you need to solve.

# Putting Things Together:
# 1 Hidden Layer with Two Neurons



- In this example, we have two neurons (units) in the hidden layer. Each neuron in the hidden layer learns the weights affiliated with it. During the training process, each neuron will learn to coordinate with the other to some extent, but they are completely different from each other.

- The two neurons are essentially independent of each other having different weights, and each input is weighted differently. For example, the hidden layer neuron $Z_1$ uses a weight $W_{11}$ for the input $X_1$ and weight $W_{21}$ for the input $X_2$. Similarly, the hidden layer neuron $Z_2$ relies on weights $W_{12}$ and $W_{22}$ for the inputs $X_1$ and $X_2$.

- Now the network has an ensemble of classifiers that allows it to create a new representation of the data, that is different from the direct input features $X_1$ and $X_2$. The decision making is then done on this new, higher-level representation.

$$Z_1 = X_1 * W_{11} + X_2 * W_{21}$$

$f_1(Z_1)$ applies the activation function on $Z_1$ above

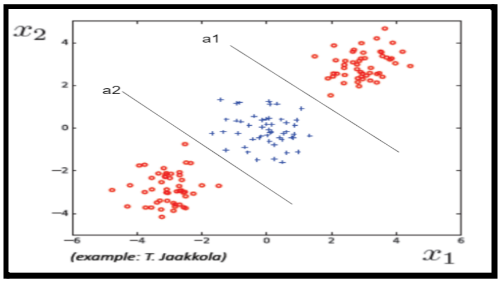
$$Z_2 = X_1 * W_{12} + X_2 * W_{22}$$

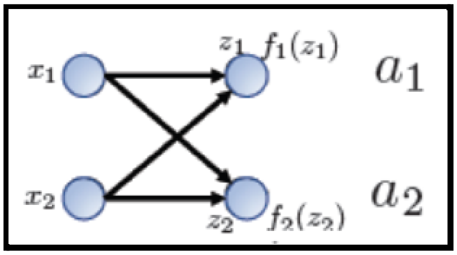
$f_2(Z_2)$ applies the activation function on $Z_2$ above

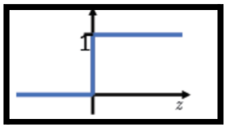
- The effect of this mechanism is each neuron ends up capturing different characteristics of the input.

- The input to the output layer (from the hidden layer) is some high-level representation of the inputs $X_1$and $X_2$ that the hidden layer has computed. The range of values potentially given by the output layer, depends on the activation function being used in the output layer. If the Sigmoid function is used, the output value will be in the range 0 to 1, for Tanh the output will range from -1 to 1, and for ReLU it will be any real number greater than or equal to 0. ( i.e. $Max(0, Z)$ )

## Let's use the above architecture with an example to explain this in more detail:



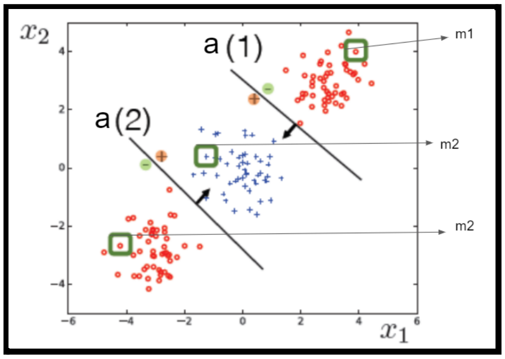


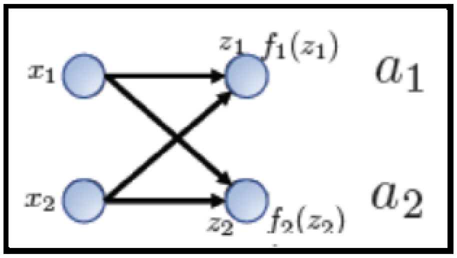


In the above example, the dataset has two classes - Red and Blue. The above graph is a 2D representation of the data. $X_1$ and $X_2$ are the inputs that are passed to two neurons (units) in the hidden layer. $Z_1$and $Z_2$ are the weighted sums of inputs, and $f_1$ and $f_2$ are the activation function (step function) being applied on the weighted sum of inputs. The output from the first and second neurons are $a_1$and $a_2$ respectively.

Now, let’s visualize the operation happening in each of the neurons in the hidden layer.

$a_1$ and $a_2$ are the two linear classifiers, each of which is a decision boundary where one side of the boundary has 1 (+ve) and the other side has 0 (-ve). In this example (see figure below), the positive side of each decision boundary is denoted by an arrow pointing in a certain direction (a point lying in that direction from the line is in the positive direction (hence 1), and the negative side of the decision boundary would be the other direction away from where the arrow is pointing (hence 0).

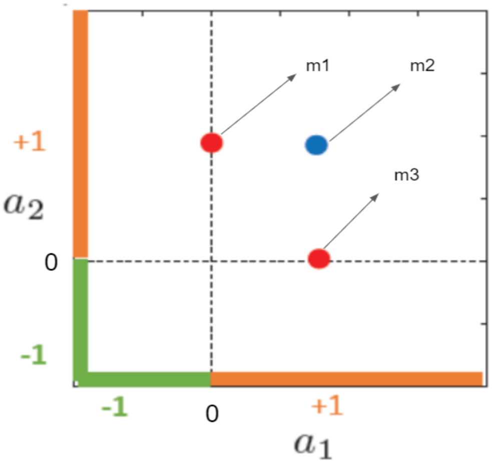


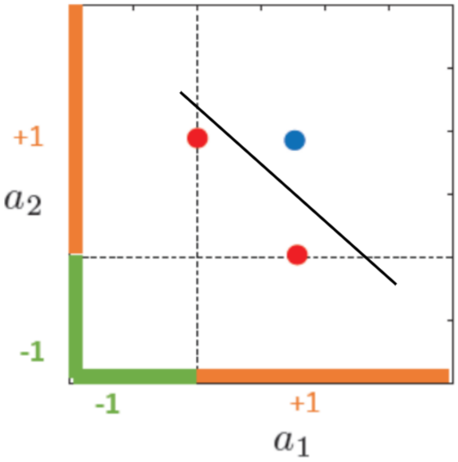


- So, when data point $m_1$ is passed to the hidden layer, the output of the first neuron $a_1$ will be **0** as it is on the negative side of the classifier, and $a_2$ will be **1** as it is on the positive side of the classifier.
- When the data point $m_2$ is given to the hidden layer, the output for $a_1$ and $a_2$ will be **1** and **1** respectively, as it lies on the positive side of both the classifiers.
- For the data point $m_3$, the outputs for $a_1$ and $a_2$ will accordingly be **1** and **0** respectively.

After applying the above activation function on the input data, the new representation of the data is shown in the below figure.

All the **red** data points in the dataset will be represented either in the place of $m_1$ (0,1) or $m_3$ (1,0), and the **blue** points will be represented by the position of $m_2$ (1,1).





With this new "encoding", (a deep learning term for a representation) of the data, the differently colored data points in the dataset can now be easily **separated** using a **single straight line**, something that was ostensibly not possible in the original representation of the inputs, where two straight lines were required to separate the two colors.

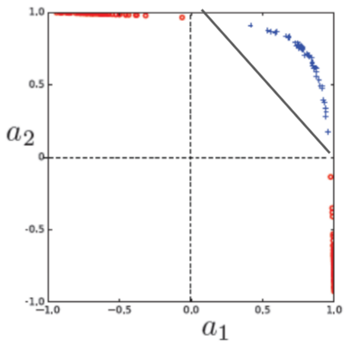
This is the power of deep learning - it internally creates new and alternate representations / encodings of the original dataset, that can turn out to be easier to draw decision boundaries and make predictions on, when compared to the input feature space.

The above is obviously not restricted to one particular type of activation function.

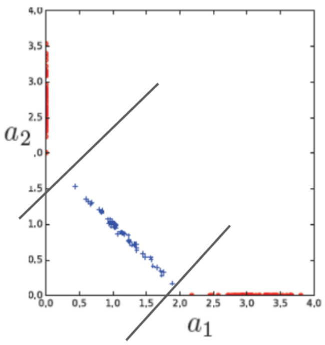
Let's look at the encodings we would get if we were to use a different activation function for the above dataset:

1. The Tanh Activation Function
2. The ReLU Activation Function

**The Tanh Activation Function**



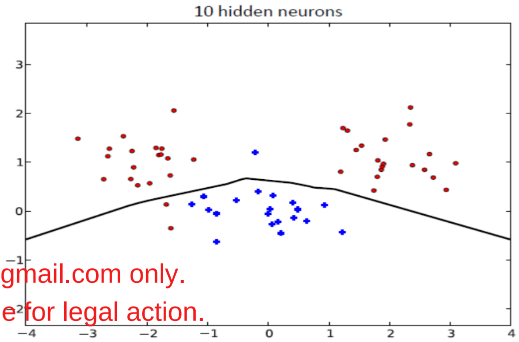
**The ReLU Activation Function**



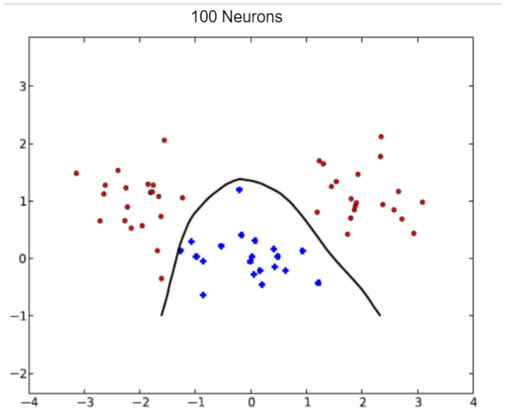
**Note:** In both the above activation function encodings, **each datapoint has its own unique representation** in the new vector space, which was not the case earlier with the variant of the threshold / step activation function we used at first. This turns out to be a more useful kind of representation, as we need to maintain the differences between individual data points to build more powerful predictive models, and this is one academic reason step functions are not preferred as activation functions in the hidden layers of deep neural networks.

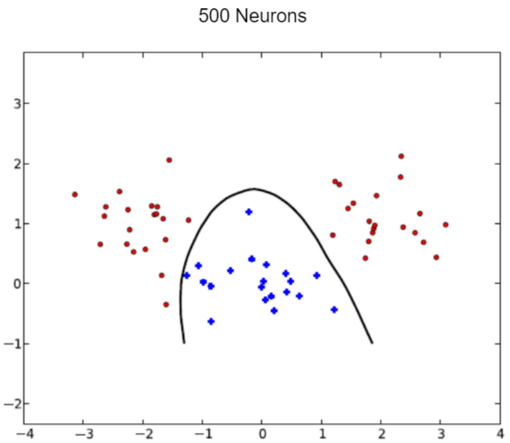
## What happens with more hidden neurons?

If we **increase** the number of **neurons** in the hidden layer, the decision boundary will become **more smooth.**



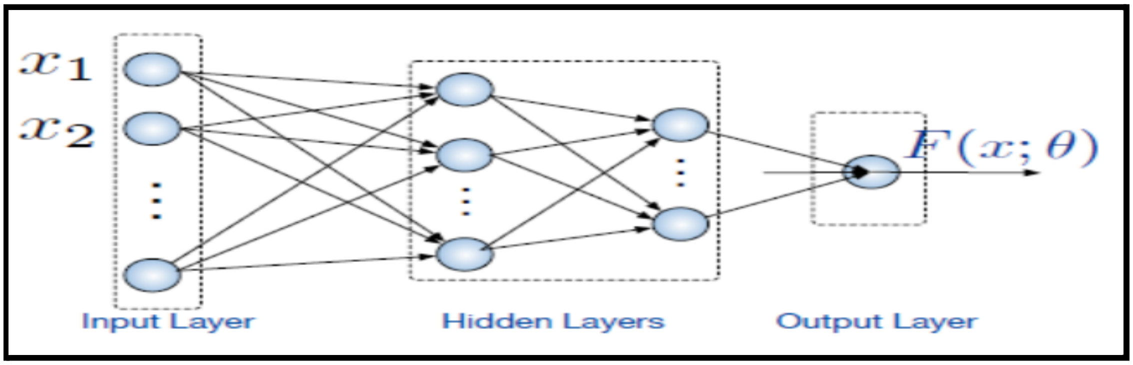
Now with **10** hidden neurons in the hidden layer, that corresponds to having 10 different **hyperplanes** which are trying to separate out the data points. This decision boundary is **smoother** than the one with only **2** neurons in the hidden layer.





- The wider the hidden layer (the more the number of neurons in the hidden layer), the smoother the decision boundary will be.
- But increasing the number of neurons in our neural network may lead to overfitting problems, and is hence not always the right solution.

## Hierarchical Representations: Multiple Layers



- In a fully-connected feedforward neural network, every neuron in the hidden layer is connected to **all the neurons** in the **next** layer, and every neuron in the hidden layer accepts an input from **all the neurons** from the **previous** layer.

- Each of these neurons in the hidden layer will typically **act** as a **linear classifier** and can make simple decisions. It takes the weighted combination of inputs and applies activation functions on it. When these ensembled neurons are put together, it **solves complicated decisions** by breaking them down into simpler ones.

Let’s take an image of an object such as a car, as an input for example:

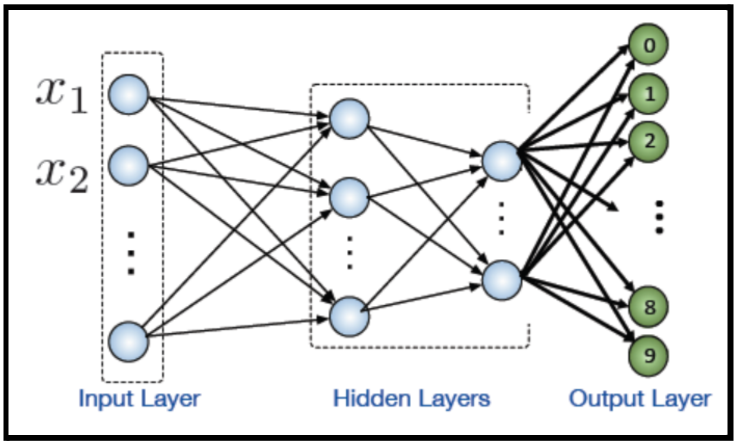
- The image data after passing through the first hidden layer may **detect just** an **edge**. So each of the neurons in the first hidden layer could act as an **edge detector**. For example, these neurons may detect the outer lines of the car, the edges of the wheel, etc.

- For the next layer, **the input** will be **different edges detected** by the **previous** layer, and the neurons in this hidden layer will detect an **object that is a combination of all of those edges.** For example - the wheels or the body of the car.

- Now, the input for the next layer will have some small objects, which will mean that this hidden layer will now predict even higher-level objects, such as the whole car itself.

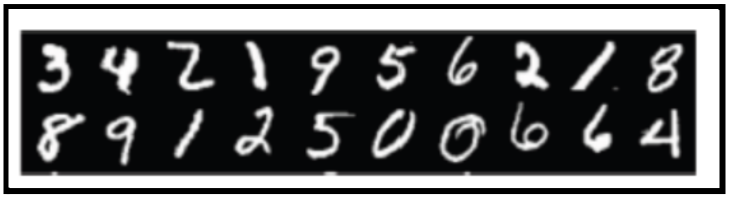
What if we want to predict multi-class classification which is not just 1’s and 0’s?
Let’s look at one multi-class classification problem and understand how neural networks can be used to solve such prediction problems.

## Multi-Class Prediction



- For a multi-class classification problem, the operations of the neurons in the respective hidden layers will be exactly like in the binary classification problem. The only difference is in the output layer, where instead of just one output neuron, the number of neurons in the output layer will depend on the number of classes.

- Let's take the MNIST data set for example, where the network has to predict a number between 0 to 9. In this case, there will be 10 neurons in the output layer, as there are ten digits from 0 to 9.



- For multi-class classification problems, **the Softmax function** is used as the activation function in the output layer. This works well when the output layer is one-hot encoded.

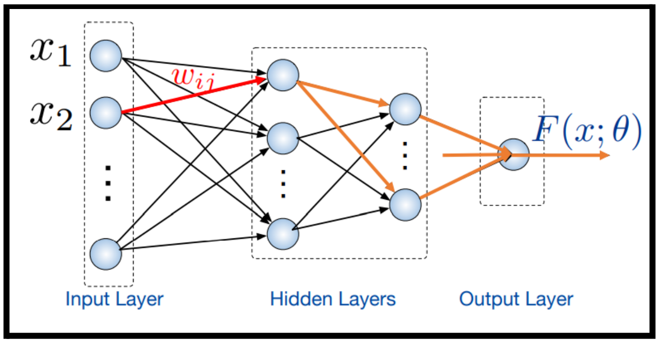
  **What is one-hot encoding?** Let say one of the data points belongs to the number 2 class, for this, the output layer would look as follows (1 means that class is present, and 0 means it's not)

  2 → 0010000000
  These digits represent the digits from 0 to 9. **2** lies in the third position (0, 1 and then 2), and hence that position has 1 (meaning that class is present). The other positions only have 0 (because those classes are not present).

  Similarly, the one-hot encoded version of 3 → 0001000000

- The output of the neurons will be a probability value for each of the 10 classes. So each of the ten neurons in the output layer, outputs the probability of occurrence of that class digit. The highest of the probabilities is considered a prediction of occurrence of that particular digit.

## How to train a neural network?



## Training a neural network is done in two alternating steps: Forward Propagation and Backward Propagation

### Let’s look at what happens in Forward Propagation:

- Forward propagation steps –
  - Calculate the weighted input to the hidden layer by multiplying $X_1$ (input) by the hidden weight $W_{11}$.
  - Apply the activation function and pass the result to the next layer.
  - In the next layer, repeat the above step by replacing $X1$ with the previous layer output.
  - This is repeated all the way till the final output layer.
- The final output from the neural network is called $\widehat{Y}$ (predicted).
- The next step is to calculate the loss.
  $Loss\ = (Y - \widehat{Y})^{2}$
- If the loss is too high, then the weights in the neural networks will be updated during backpropagation.
- The loss should be reduced such that both $Y\ and\ \widehat{Y}$ should be the same. The loss function can be reduced by using an optimizer.

### Let’s now look at Backpropagation:

- Backpropagation is the process of learning that the neural network employs to re-calibrate the weights and biases at every layer and every node to minimize the error in the output layer.

- During the first pass of forward propagation, the weights and bias are random numbers. The random numbers are generated within a small range say 0 – 1.

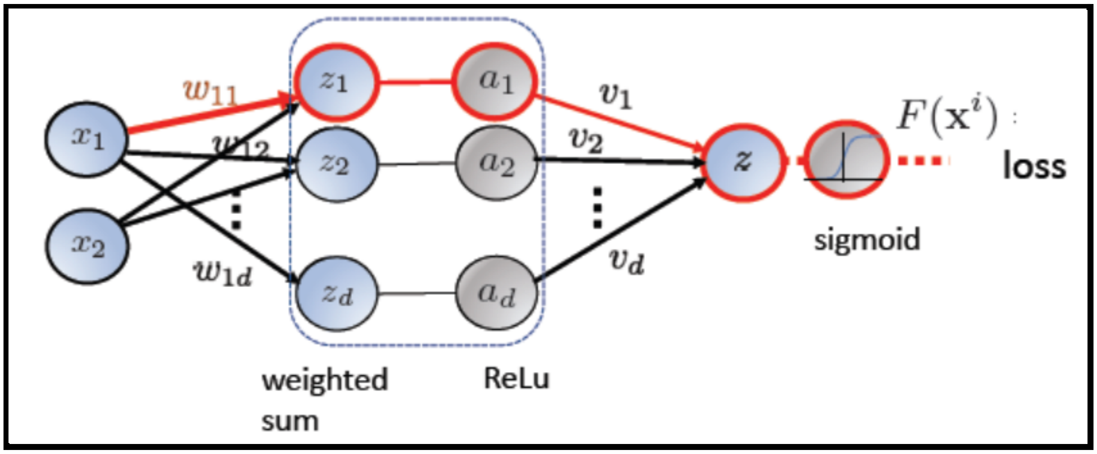
- Needless to say, the output of the first iteration is almost always incorrect. The difference between actual value/class and predicted value/class is the error.

- All the neurons in all the preceding layers have contributed to the error and hence need to get their share of the error and correct their weights.

- The goal of backpropagation is to adjust weights and bias in proportion to the error contribution and an iterative process identifies the optimal combination of weights.

- At each layer, at each node, a gradient descent algorithm is applied to adjust the weights.

## How does a weight change impact the loss?



- In the above architecture if weight w11 is changed, then w11 affects the weighted sum ($Z_1$) and that affects the output of the activation function ($a_1$) which is fed to the output neuron and then applies an activation function (sigmoid) on it and finds loss.
- This is exactly how a change in weight influences the loss and compute partial derivatives.
- Let's have a look at the formula

$$W_{11\,new} = W_{11\,old} - \eta \frac{\delta L}{\delta W_{11\,old}}$$

$$\frac{\delta}{\delta W_{11\,old}} L(x_i, y^i, \theta) = \frac{\delta z_1}{\delta w_{11\,old}} * \frac{\delta a_1}{\delta z_1} * \frac{\delta z}{\delta a_1} * \frac{\delta L}{\delta z}$$

$$= x_1 * 1[z_1 > 0] * v_1 * F[(x^i) - y^i]$$

1. $F[(x^i) - y^i]$ - This is the error it computes, if $x^i$ $and\ y^i$ match no need to calculate derivative or weight update
2. $v_1$ - Subsequent weight assigned to output
3. $1[z_1 > 0]$ - Activation function, if this neuron is not activated the partial derivative will be zero
4. $x_1$ - Input
5. $\eta$ - Learning rate / Step size

Now, this is for one neuron in one hidden layer.
What if the neural network has many hidden layers and neurons in it?

The path for weight update of each weight becomes very long, from one weight there are multiple passes to the output. The weight update procedure is hence more complicated for deep neural networks. As there will be many layers in a deep neural network, the derivatives (collection of weights) will either be a very huge value or small value and this could lead to a vanishing gradient or exploding gradient problem.

To avoid this, the neural network may be initialized in such a way that the partial derivatives should be neither too small nor too big.

# Weight Initialization

## What is Weight Initialization?

- Weight Initialization is a procedure to assign weights to a neural network with some small random values during the training process in a neural network model.

## Why do we need Weight Initialization?

- The purpose of using weight initialization is to prevent the neural network from exploding or vanishing gradients during forward and backward propagation. If either of those occurs, the neural network will take a longer time to converge.

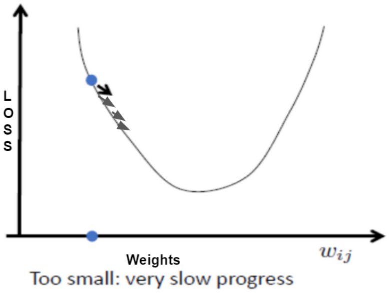
## Initializing all Weights to Zero

- This makes your model equivalent to a linear model.
- When you set all weights to 0, the derivative with respect to the loss function is the same for every weight in every layer, thus, all weights have the same values in the subsequent iteration.

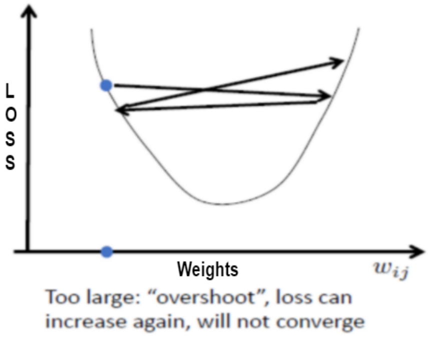
## Initializing all weights randomly

- Initializing weights randomly, following a standard normal distribution while working with a (deep) neural network can potentially lead to 2 issues — vanishing gradients or exploding gradients.

## What are vanishing gradients and exploding gradients?



- **Vanishing Gradient:** In the case of deep networks, for any activation function, the derivatives of weights will get smaller and smaller as we go backward with every layer during backpropagation. The weight update is minor and results in slower convergence. This makes the optimization of the loss function slow. In the worst case scenario, this may completely stop the neural network from training. This is called the **Vanishing Gradient** problem in Deep Learning.


L
O
S
S
Weights
$w_{ij}$
Too large: "overshoot", loss can increase again, will not converge

- **Exploding Gradient**: This is exactly the opposite of vanishing gradients, where large error gradients accumulate and result in very large updates to model weights of neural networks during training. This will lead the model to slower convergence and result in oscillating around the minima or even overshooting the optimum again and again and the model will never learn. This is called the **Exploding Gradient** problem in Deep Learning.

# Optimizers

Optimizers are the algorithms used to change the parameters of the neural networks such as weights, bias, and learning rate to reduce the loss. The loss is reduced by updating these parameters during backpropagation.

## Types of Optimizers

- **Stochastic Gradient Descent (SGD):**

  Stochastic Gradient Descent (SGD) is an optimization algorithm used in Deep Learning. During the training period to find the derivative loss function, a random data point is selected instead of the whole data for each iteration.
  For Example: A dataset consists of 1000 data points, and to calculate the derivative loss

function it will be considering only one data point at a time.

In SGD, convergence to the global minima happens very slowly as it will take a single record in each iteration during forward and backward propagation.
As it takes one data point at a time there will be noise.

- **Gradient Descent with Momentum:**

  To overcome the noisy data produced by the SGD, there is another variant of Gradient Descent called Gradient Descent with Momentum, which smoothens the noisy data.

  Gradient Descent with Momentum uses exponentially weighted averages of gradients over the previous iteration to stabilize their convergence.

- **Adagrad (Adaptive Gradient Algorithm):**
  In the Adagrad optimizer, there is no concept of momentum, so it is simpler compared to SGD with momentum.

  The idea behind Adagrad is to use different learning rates for each parameter based on iteration. The reason behind the need for different learning rates is that the learning rate for sparse feature parameters needs to be higher compared to the dense feature parameters because the frequency of occurrence of sparse features is lower.

- **Adam (Adaptive Moment Estimation):**

  Adam can be thought of as a merger of Adagrad and SGD with momentum. Like Adagrad, it utilizes the squared gradients to scale the learning rate, and like SGD with momentum, it uses the moving average of the gradient as an alternative to the gradient itself.

  Algorithms like Stochastic Gradient Descent with Momentum, Adagrad, and Adam's optimizer are similar in performance, but Adam has a slight edge over the others due to the bias correction performed in Adam.

## Types of Loss Function: Classification

➢ **Categorical Cross-Entropy**

- It is a loss function used in multi-class classification, and the target labels are one-hot encoded.
- It is well suited for classification tasks with mutually exclusive choices, so one example can belong to one class with probability 1 and the other with probability 0.
- For example, in the MNIST digit recognition, the model gives higher probability predictions for the correct digit and lower probabilities for the other digits.

➢ **Binary Cross-Entropy**

- It is used in problems related to binary classification.
- It compares each of the probabilities predicted to the output class which can be either 0 or 1.

## Types of Loss Function: Regression

➢ **Mean Squared Error (MSE)**

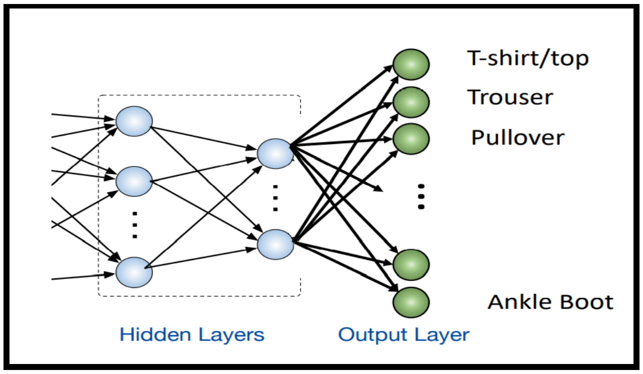
- It is calculated as the average of the squared differences between actual and predicted values.
- The result is always positive and the case with no error will always have MSE=0
- As it is sensitive to outliers, MSE is great for ensuring our trained models have no outlier predictions with huge errors, as that implies larger weights on outliers due to the square function.

➢ **Root Mean Squared Error (RMSE)**

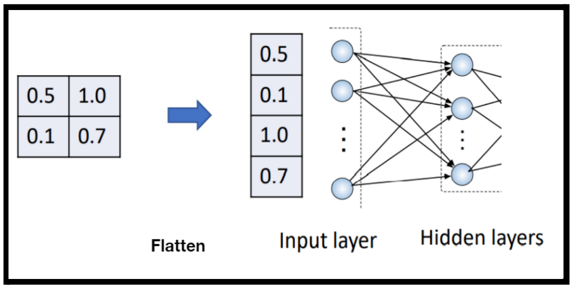
- It is similar to MSE with a root function over it.
- Here the errors are squared before the average is taken, so RMSE gives a high weightage to large errors.

Both MSE and RMSE have a range from 0 to ∞. The model is good if the value of these errors is low.

# Example: Fashion MNIST



- The Fashion MNIST dataset contains 60,000 (train) + 10,000 (test) article images.
- Each image in the dataset has a 28x28 pixel size.
- The dataset consists of 10 classes (T-shirt/top, Trouser, Pullover, Dress, Coat, Sandal, Shirt, Sneaker, Bag, Ankle Boot)
- We have images of clothes and each image to classify what type of dress it belongs to.
- For image data, the input to the neural network will be the pixel values of the image.
- As this dataset is a multi-class classification problem, the loss function will be the Categorical Cross-Entropy function.
- In this dataset, each image has 28x28 pixel values which will be flattened and passed to the neural network as shown in the below figure.


0.5
1.0
0.1
0.7
0.5
0.1
1.0
0.7
Flatten
Input layer
Hidden layers