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. This is one of the greatest advantages of deep learning over machine learning.

Deep Learning: Reason 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 the vast improvements in technology and greater improvements in computational power like higher-end CPUs, GPU's and storage devices. Our deep learning models have a wider spectrum of chances to train on larger data sets more efficiently and on large scales.

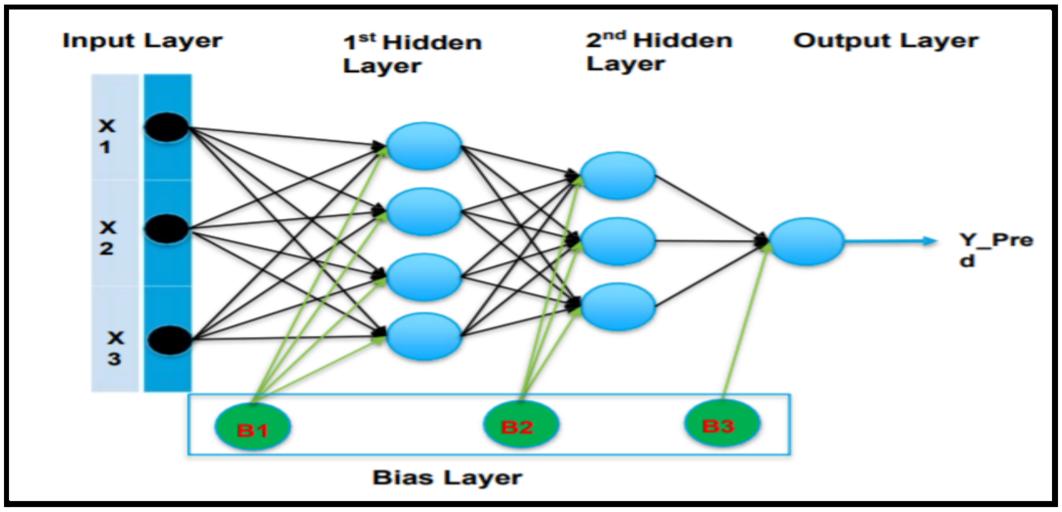
- **Large models are easier to train**:

  As we would even observe that with the advent of deep learning models, the architectures have become too deep and large with several layers. So these models are easier to train with our high-end GPUs and with our gradient-based approaches and batch size variations, this has become a lot easier.

- **Flexible neural "lego species"**:

  The most important feature of deep learning models is that they are easier to modify. New architectures are easier to get created as the different models in deep learning have only fewer changes like the number of layers, the number of neurons, and the activation or loss functions that, can be easily altered to create new and similar architectures.

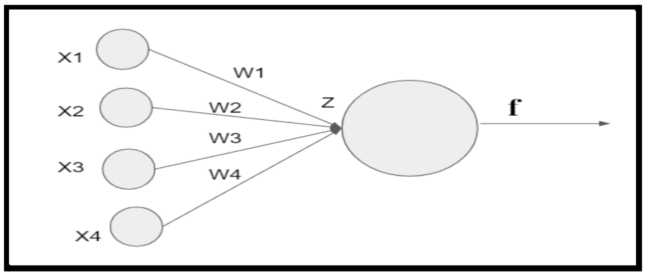
Feedforward Neural Network

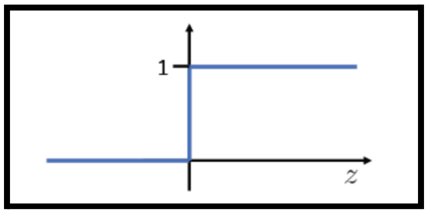


- The input layer is passive, does no processing, only holds the input data to supply it to the **first hidden** layer. It is a vector that describes the features of the input.
- Anything that is not Input and Output layers are called **hidden layers.** Each neuron in the first hidden layer takes all input attributes, **multiplies** with the corresponding **weights**, adds **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 their **weights.**
- 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. Output for a classification problem will be (**YES/NO**) and for a regression problem, it will be a **real number**.

Let's see what operations are happening in a single neuron (unit)

A Unit in a Neural Network



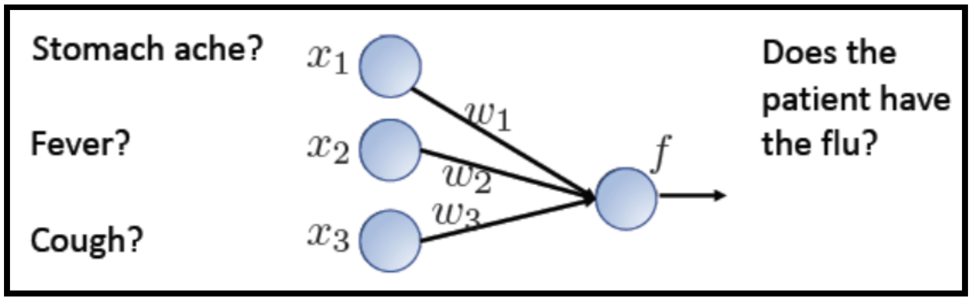


- A neuron in a neural network works in three steps:
    1. First, it multiplies the input signals with corresponding weights.
    2. Second, the weighted sum of the inputs.
    3. Applies activation function on the weighted sum.
- The input to a neuron is a **summation** of **inputs** and respective **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 be **ignored** by the neuron.
- The summation of inputs and weights will give a **score**, if the score is **greater** than the threshold it will output **1** otherwise it will output **0**.
- The above **operation** is similar to the **Linear classifier** operation, where it takes the weighted combination of inputs and compares it with the **threshold**. The parameters of the Linear classifiers are these **weights** and **bias**(threshold).
- So, a neuron in a neural network works as a linear classifier of the inputs, now there are many linear classifiers in the hidden layers which have different weights, each neuron **detects** a different **pattern** but essentially is a linear classifier.

More intuition: a simple example



- Now we have three inputs Stomach ache (X_1), Fever (X_2), and Cough (X_3), and the Output we have is Does the patient have the flu or No flu.
- Let's assume that the inputs are in **1**'s and **0**'s, 1-**Yes,** and 0-**No**.
- Stomach ache is **not** a symptom of flu so, the respective **weight** will be **decreased** (Negative number), fever and cough are **symptoms** of flu so the corresponding weights for those two inputs will be high (Positive number)

- 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
X2	Fever	NO	3
X3	Cough	YES	3

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

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

Hence the output of a neuron is less than the threshold, it fires 0 which means 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	-1
X2	Fever	YES	3
X3	Cough	YES	3

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

After calculating the equation the output from the neuron will be 6, which is greater than the threshold value, and fires 1, which means this person is suffering from flu.

NOTE: These **weights** and **thresholds** (chosen as 5) in the example are taken randomly for explanation purposes.

For the above example, we used the Threshold/ Step activation function, where it uses a threshold value to determine the output, if the output is greater than the threshold it fires 1 and smaller than the threshold fires 0, due to this high jump it's not an appropriate way to determine the weights of a neural network.
So let's have a look at some of the smoother activation functions.

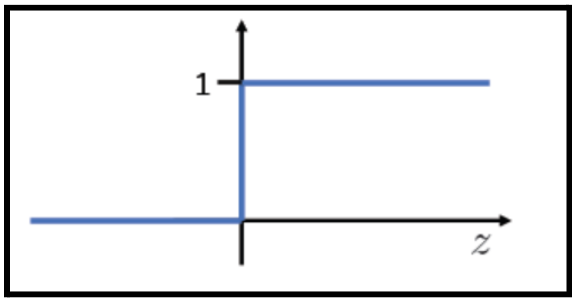
Activation function

The activation function decides whether a neuron should be activated or not by calculating the weighted sum and further adding bias with it. 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. An activation function is one of the critical components that give the neural networks ability to deal with complex problems.

Some of the common types of Activation Functions:

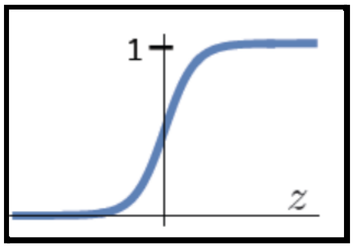
1. Binary Step function
2. Sigmoid function
3. Tanh function
4. Relu function

- **Binary Step function**: This activation function is very basic and it comes to mind every time if we try to bound output. It is a threshold-based classifier. In this, we decide some threshold values to decide whether the neuron should be **activated** or **deactivated**. Here, we set the threshold value to **0**. It is very simple and useful to classify binary problems or classifiers.



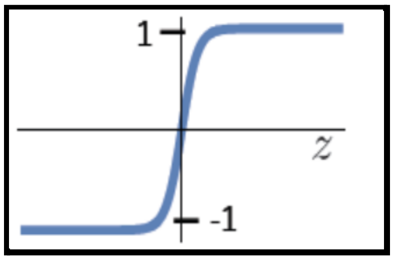
$$\boxed{f(z) = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{otherwise} \end{cases}}$$

- **Sigmoid**: A sigmoid activation function is the most common activation function used in an **output** layer for **binary** classification problems. The main reason why sigmoid is used is that it gives output between **0** and **1**. 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.



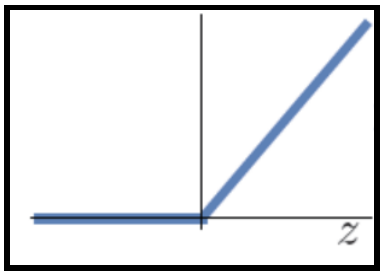
$$\boxed{f = \frac{e^z}{1+e^z}}$$

- **Tanh:** Tanh function is also known as **Tangent Hyperbolic** function. It's a mathematically shifted version of the sigmoid function. Usually, it is used in hidden layers of a neural network as its values lie between **-1 to 1.**



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

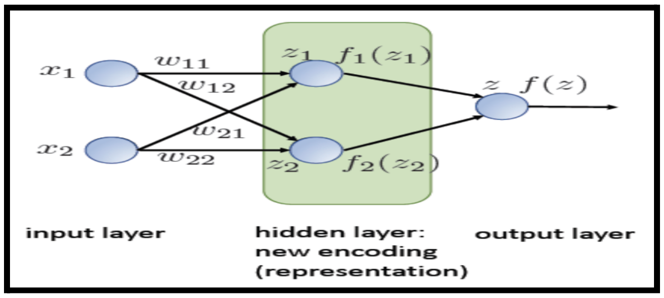
- **ReLu (Rectified Linear Unit)**. It is the **most** widely used activation function. Chiefly implemented in **hidden layers** of the Neural network. It ranges from **0** to∞. ReLu is **less** computationally expensive than Tanh and sigmoid because it involves **simpler** mathematical operations. In simple words, RELU **learns** much **faster** than sigmoid and Tanh functions.



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

Each activation function has its advantages and disadvantages, selecting which type of activation function to use is a hyperparameter and it also depends upon the problem statement to solve.

Putting things together: 1 Hidden layer with two neurons



- Now we have two neurons (units) in the hidden layer, each neuron in the hidden layer learns its weights. During the training, each neuron will learn to coordinate with each other to some extent but they are completely different from each other.

- Two neurons are essentially independent of each other, having different weights and each input is weighted differently. Weights W_{11} and W_{12} are assigned to input X_1 which have different values. Weights W_{21} and W_{22} are assigned to input X_2 which have different values.

- Now the network has an ensemble of classifiers where it gives a new representation of data not in terms of input but making high-level decisions. The higher-level decision means applying the activation function on Z .

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

$f_1(Z_1)$ Appling activation function on Z_1

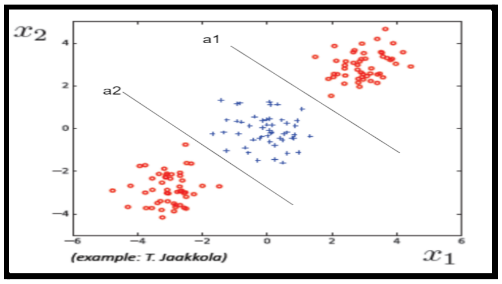
$$Z_2 = X_1 * W_{12} + X_2 * W_{22}$$

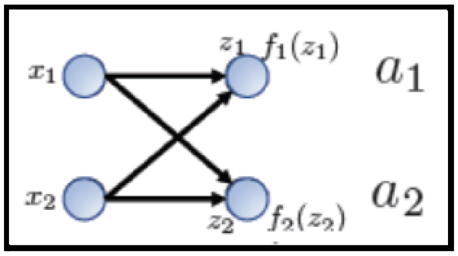
$f_2(Z_2)$ Appling activation function on Z_2

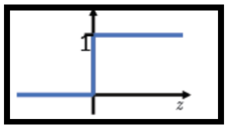
- Each neuron tells different characteristics of the input, let's say we have an image as an input where one of the neurons will be detecting the edges, etc.

- Input to the output layer is a high-level representation of the input X_1 and X_2. The output layer outputs the value depending on the activation function being used in the output layer. If sigmoid is used its value will be between 0 to 1, Tanh will be -1 to 1 and Relu will be 0 to a real number other than negative i.e.$Max(0, Z)$

Let's use the above architecture with an example.



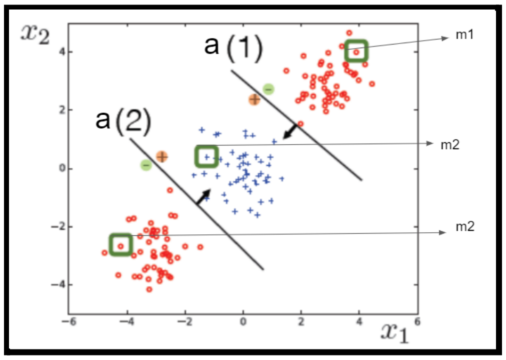


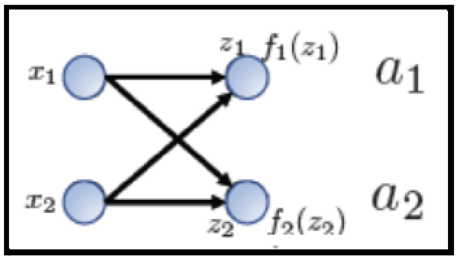


In the above example, the dataset has two classes [Red, Blue]. The above graph is a 2D representation of the data. X_1 and X_2 are the inputs that are passed to two neurons (units) in the hidden layer. Z_1 and Z_2 are the weighted sums of inputs and f_1 and f_2 are the activations function [Threshold/Step] applying on the weighted sum of inputs, Output from the first and second neuron is a_1 and a_2 respectively.

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

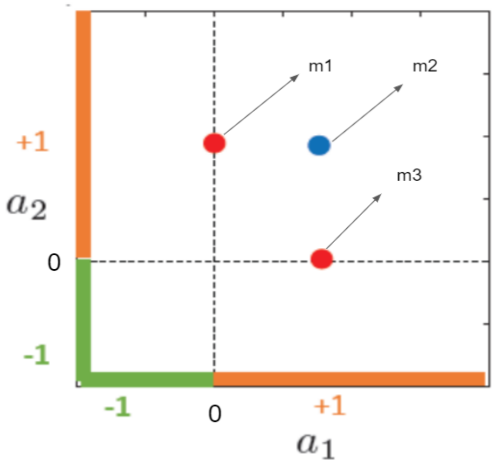
a_1 and a_2 are the two linear classifiers which each of them is a decision boundary where one side of the boundary has 1 (+ve) and the other side has 0 (-ve).



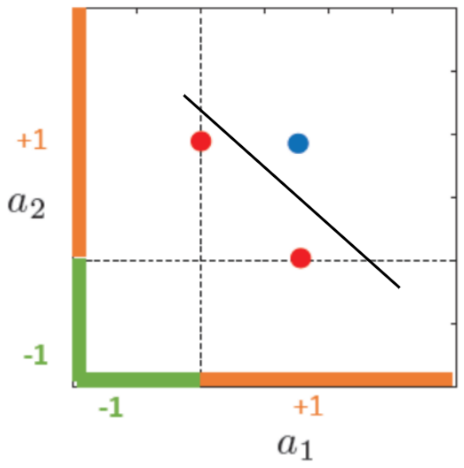


- When data point m_1 is passed to the hidden layer the output of the first neuron a_1 will be **0** as it is (-ve) side of the classifier and a_2 will be **1** as it is (+ve) side of the classifier.
- Output for a_1 and a_2 when data point m_2 is given to the hidden layer are **1** and **1** respectively as it is lying on the positive side of both the classifiers.
- For data point, m_3 outputs for a_1 and a_2 will be **1** and **0** respectively.

After applying the 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 or m_3 and blue points in m_2 as we are using the threshold/step activation function.

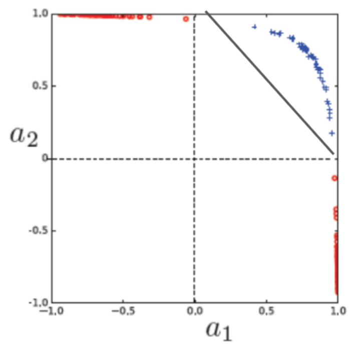


With this new encoding (representation) of the data, all the data points in the dataset can be **separated** using a **single straight line**.

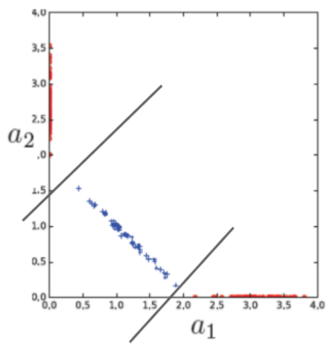
What if you use a different activation function for the above dataset.

1. TanH Activation function
2. ReLu Activation function

TanH Activation function



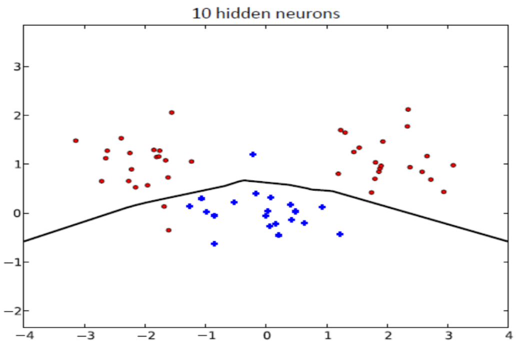
ReLu Activation function



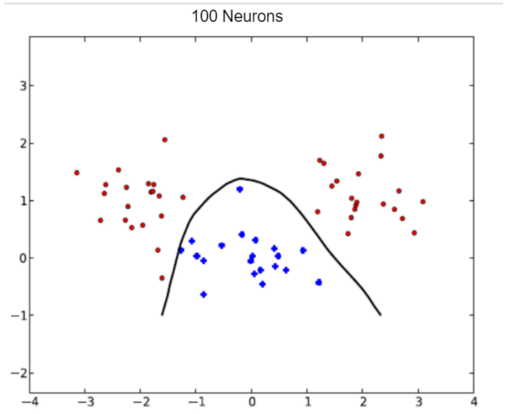
In the above two activation functions representation, **each** data has its representation which was not in the case of threshold/step activation.

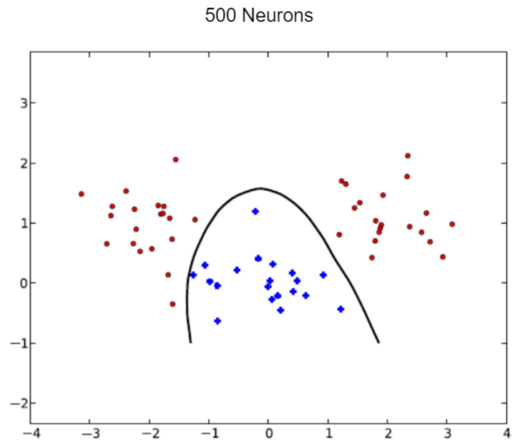
What happens with more hidden neurons?

If we **increase** the number of **hidden neurons** in the hidden layer the decision boundary will be more **smoothen**.



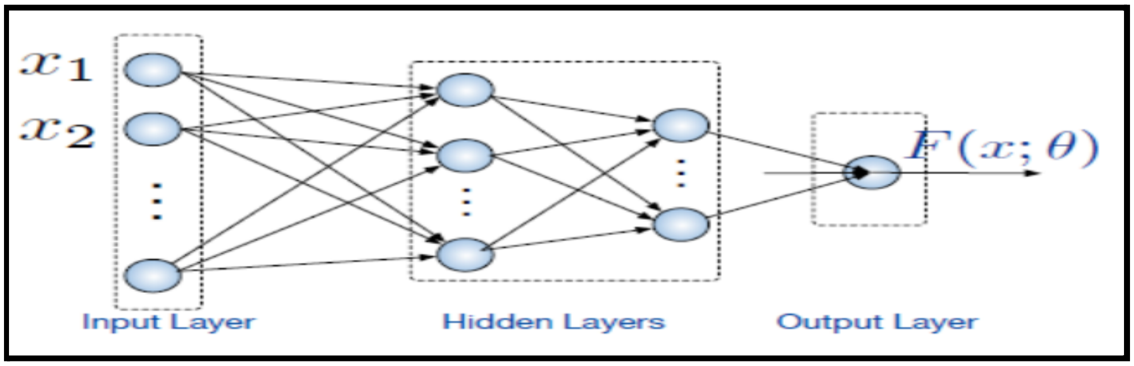
Now with **10** hidden neurons in the hidden layer corresponds to having 10 different **hyperplanes** which are trying to separate the data points. The above decision boundary is **smoother** than the decision boundary with **2** neurons.





- The wider the hidden layer (More neurons in the hidden layer) the smoother the decision boundary will be.
- But making our neural network deeper (More no neurons) will lead to overfitting problems.

Hierarchical representations: multiple layers



- Every neuron in the hidden layer is connected to **all** the **neurons** in the **next** layer and every neuron in the hidden layer **accepts** the input from the neurons from the **previously** hidden 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 into simpler ones.

Let's take Image (Car) as an input for example:

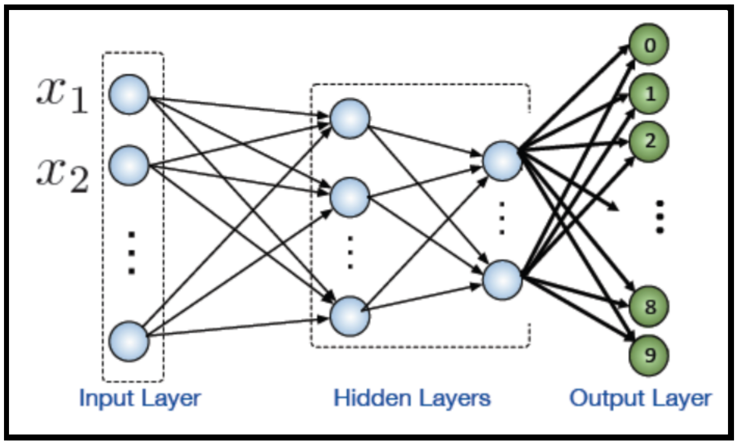
- Image data after passing through the first hidden layer could just **detect** an **edge**, each of the neurons in the first hidden layer acts as an **edge detector**. For example, it will detect the car outer line, **wheel** edges, etc

- 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 the edges. For example - wheels, the body of the car.

- Now the next layer input will have some small objects which will tend to this hidden layer to predict the bigger object i.e. car.

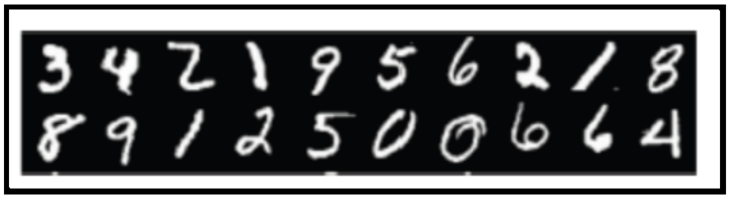
What if we want to predict the multiclass classification which is not just 1's and 0's? Let's look at one multi-class classification problem and understand how neural networks work.

Multi-Class Prediction



- For a multi-class classification problem, all the operations of neurons in the respective hidden layer will be the same as the binary classification problem, only the output layer will be different where the number of neurons in the output layer will have more than 2 neurons depending on the number of classes.

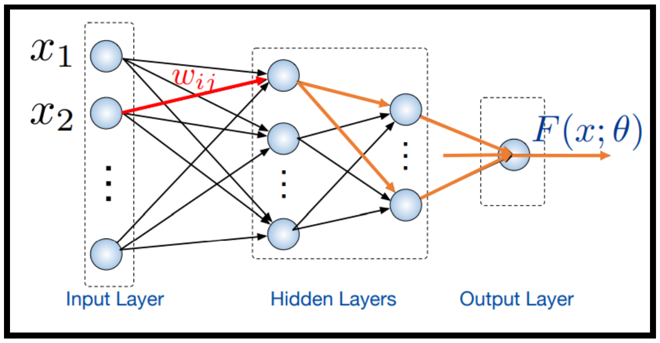
- Let's take the MNIST data set where the network has to predict the number between 0 to 9, so there will be 10 neurons in the output layer.



- Here, for multi-class classification problems, softmax will be used as an activation function in the output layer. Neural networks work well when the output layer is one-hot encoded.
  What is one-hot encoding? Let say one of the data points that belong to the number 2 class, for this, the output layer looks as follows (1 if the class is present and 0 otherwise)

  2→ 0010000000
  3→ 0001000000
- The output of these neurons will always be a probability value, as we get 10 probability values for each test data point so we will take the index of the highest probabilistic value as our label.

How to train a neural network?



Let's see what happens in forward propagation.

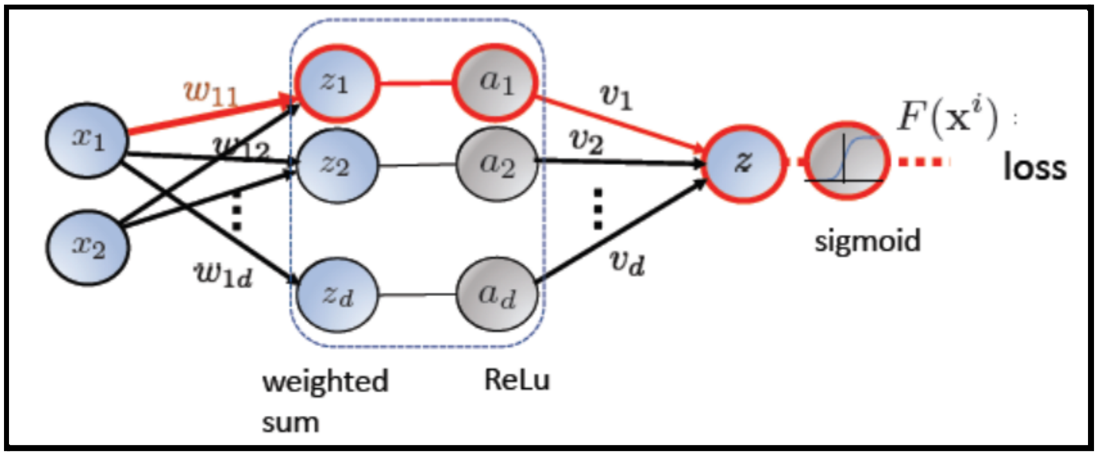
- Forward propagation steps –
  - Calculate the weighted input to the hidden layer by multiplying X_1 (input) by the hidden weight W_{11}.
  - Apply the activation function and pass the result to the final layer.
  - At the output layer, repeat the above step by replacing $X1$ with the hidden layer's output.
- The output from the hidden layer 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 weights in the neural networks will be updated during backpropagation.
- Loss should be reduced such that both Y and $\widehat{Y}$ should be the same, loss can be reduced by using an optimizer.

Let's have a 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 leads to affect the activation function (a_1) which is fed to the output neuron and then applies 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. η - 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 so long, from one weight there are multiple passes to the output. Weight update looks complicated for deep neural networks. As there will be many layers in a deep neural network, the particle derivatives (collection of weights) will either be a very huge value or small value and this leads to a vanishing gradient or exploding gradient problem.

To avoid this neural network has to be initialized in such a way that partial derivatives should not be very small or very 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 the weight initialization technique is to prevent the neural network from exploding or vanishing gradient during the forward and backward propagation. If either occurs, the neural network will take a longer time to converge.

Initializing all weights to Zero

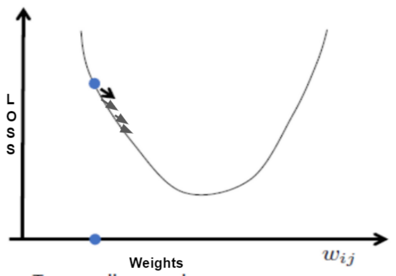
- This makes your model equivalent to a linear model.
- When you set all weights to 0, the derivative with respect to loss function is the same for every weight in every layer, thus, all the weights have the same values in the subsequent iteration.

Initializing all weights randomly

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

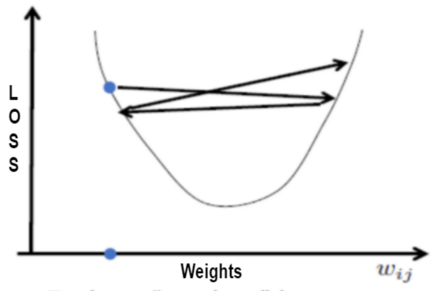
What are vanishing gradients and exploding gradients?

- **Vanishing Gradient:** In the case of deep networks, any activation function, 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, it may completely stop the neural network from training.



Too small: very slow progress

- **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.



Too large: "overshoot", loss can increase again, will not converge

Optimizer

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

Types of Optimizers

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

  Stochastic Gradient descent 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 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 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 will smoothen the noisy data.

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

- **Adagrad (Adaptive Gradient Algorithm):**
  In Adagrad optimizer, there is no momentum concept so it is much 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 features parameters needs to be higher compared to the dense features parameter because the frequency of occurrence of sparse features is lower.

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

  Adam can be defined as the merger of Adagrad and SGD with momentum.
  Like Adagrad, it utilizes the squared gradients to scale the learning rate, and
  Similarly, 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 always one-hot encoded.
  - It is well suited for classification tasks so that one example can belong to one class with probability 1 and the other with probability 0.
  - For example, In 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:

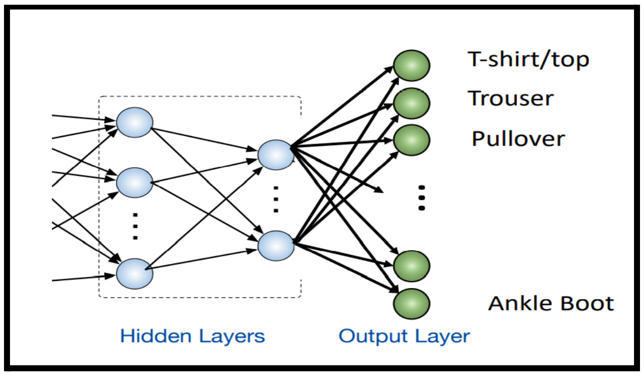
Regression

- **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 will be great for ensuring our trained models have no outlier predictions with huge errors as it 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.

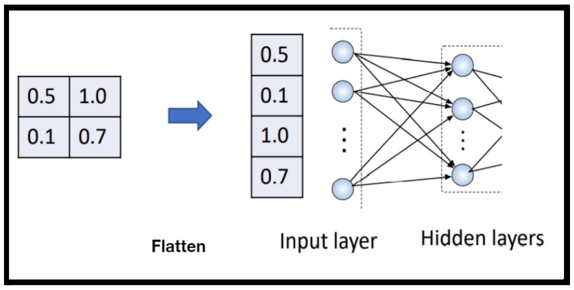
- Here the errors are squared before the average is taken so RMSE always gives a high weight to large errors.

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

Example Fashion MNIST



- Fashion MNIST dataset contains 60,000 (train) + 10,000 (test) article images.
- Each image in the dataset has a 28x28 pixel size.
- 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 cloth it belongs to.
- For image data, the input to the neural network will be the pixel values of the image.
- 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.



- As this dataset is a multi-class classification problem, the loss function will be categorical cross-entropy. Image as an input to a Neural network is called Convolutional Neural Network (CNN) and we will see more about images in the next session.