**Module 3 Notes**

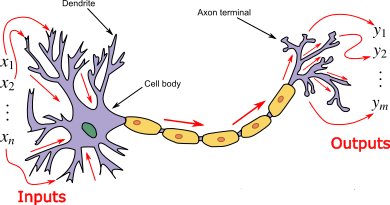
**Artificial Neural Network:**

Neural Networks is a computational learning system that uses a network of functions to understand and translate a data input of one form into a desired output, usually in another form. The concept of the artificial neural network was inspired by human biology and the way neurons of the human brain function together to understand inputs from human senses.

In simple words, Neural Networks are a set of algorithms that tries to recognize the patterns, relationships, and information from the data through the process which is inspired by and works like the human brain/biology.

**Biological Neural Networks**

* Models the brain and nervous system
* Highly parallel and distributed
* Process information
* Learning
* Very complex behaviours
* Consists of 1011 neurons densely connected
* A variety of different neurons exist (motor neuron, on-center off-surround visual cells…), with different branching structures.
* The connections of the network and the strengths of the individual synapses establish the function of the network.
* **Dendrites:** nerve fibres carrying electrical signals to the cell
* **Cell body:** computes a non-linear function of its inputs
* **Axon:** single long fiber that carries the electrical signal from the cell body to other neurons
* **Synapse:** the point of contact between the axon of one cell and the dendrite of another, regulating a chemical connection whose strength affects the input to the cell.



Signals “move” via electrochemical signals

The synapses release a chemical transmitter – the sum of which can cause a “threshold” to be reached – causing the neuron to “fire”

Synapses can be inhibitory or excitatory

Properties of ANN:

**Learning from examples** – labeled or unlabeled

**Adaptivity** – changing the connection strengths to learn things

**Non-linearity** – the non-linear activation functions are essential

**Fault tolerance** – if one of the neurons or connections is damaged, the whole network still works quite well

**History of ANN :**

**McCulloch** **& Pitts** (**1943**) are generally recognized as the designers of the first neural network

Many of their ideas still used today (e.g. many simple units combine to give increased computational power and the idea of a threshold)

**Hebb (1949)** developed the first learning rule (on the premise that if two neurons were active at the same time the strength between them should be increased)

During the 50’s and 60’s many researchers worked on the perceptron amidst great excitement.

1969 saw the death of neural network research for about 15 years – Minsky & Papert

Only in the mid 80’s (Parker and LeCun) interest was revived (in fact Werbos discovered algorithm in 1974)

**1943**: McCulloch and Pitts proposed a model of a neuron --> Perceptron

**1960s**: Widrow and Hoff explored Perceptron networks (which they called “Adelines”) and the delta rule.

**1962**: Rosenblatt proved the convergence of the perceptron training rule.

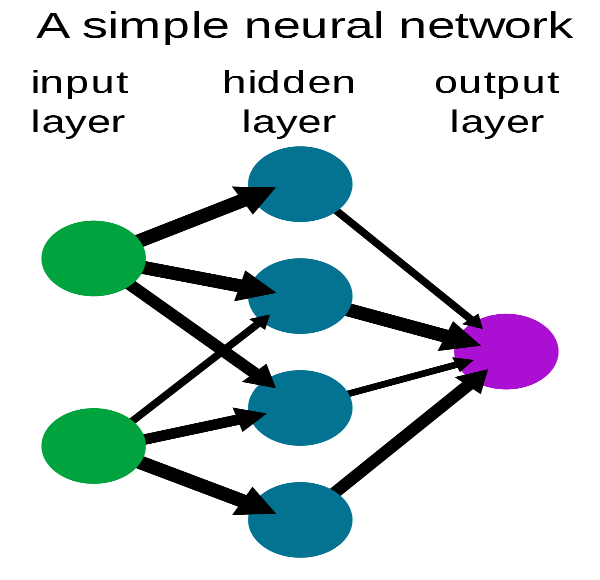
**1969**: Minsky and Papert showed that the Perceptron cannot deal with nonlinearly-separable data sets---even those that represent simple function such as X-OR.

**1970-1985**: Very little research on Neural Nets

**1986**: Invention of Backpropagation [Rumelhart and McClelland, but also Parker and earlier on: Werbos] which can learn from nonlinearly-separable data sets.

Since 1985: A lot of research in Neural Nets!

**Components / Architecture of Neural Network**

* Input layer
* Hidden layer
* Output layer

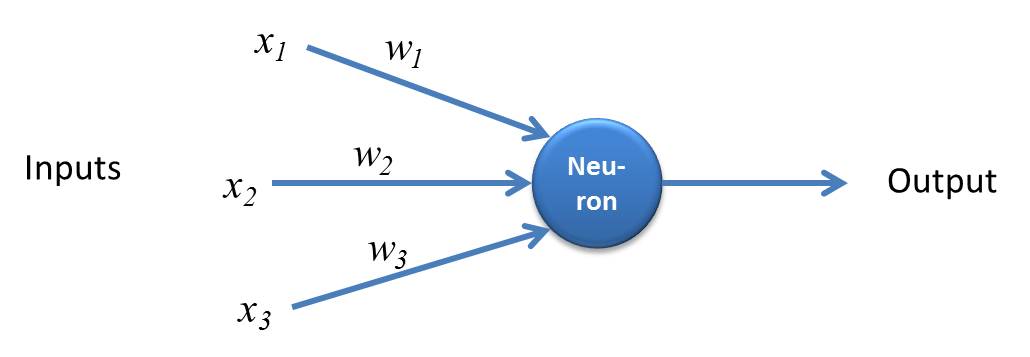
**Input Layer:**Also known as Input nodes are the inputs/information from the outside world is provided to the model to learn and derive conclusions from. Input nodes pass the information to the next layer i.e Hidden layer.

**Hidden Layer:** Hidden layer is the set of neurons where all the computations are performed on the input data. There can be any number of hidden layers in a neural network. The simplest network consists of a single hidden layer.

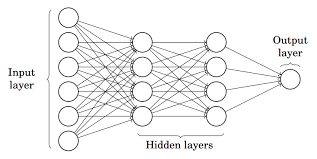
**Output layer:**The output layer is the output/conclusions of the model derived from all the computations performed. There can be single or multiple nodes in the output layer. If we have a binary classification problem the output node is 1 but in the case of multi-class classification, the output nodes can be more than 1.

## **Perceptron and Multi-Layer Perceptron**

**Perceptron** is a simple form of Neural Network and consists of a single layer where all the mathematical computations are performed.



Whereas, **Multilayer Perceptron** also known as **Artificial Neural Networks** consists of more than one perception which is grouped together to form a multiple layer neural network.



In the above image, The Artificial Neural Network consists of four layers interconnected with each other:

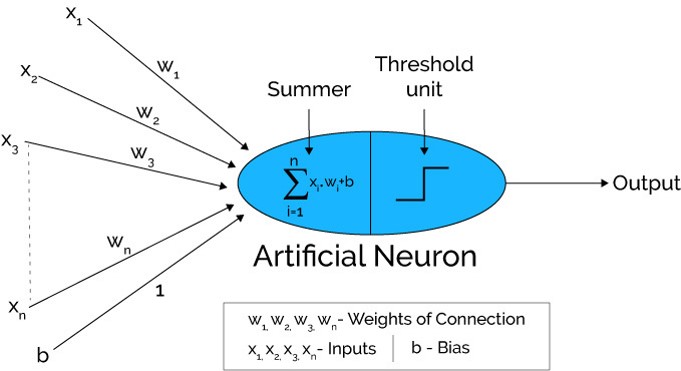
An input layer, with 6 input nodes

Hidden Layer 1, with 4 hidden nodes/4 perceptrons

Hidden layer 2, with 4 hidden nodes

Output layer with 1 output node

## Step by Step Working of the Artificial Neural Network



                                                                   Source: Xenonstack.com

1. In the first step, **Input units are passed i.e data is passed with some weights attached to it to the hidden layer**. We can have any number of hidden layers. In the above image inputs x1,x2,x3,….xn is passed.
2. Each hidden layer consists of neurons. All the inputs are connected to each neuron.
3. After passing on the inputs,**all the computation is performed in the hidden layer** (Blue oval in the picture)

Computation performed in hidden layers are done in two steps which are as follows :

* First of all, **all the inputs are multiplied by their weights**. Weight is the gradient or coefficient of each variable. It shows the strength of the particular input. After assigning the weights, a bias variable is added. **Bias** is a constant that helps the model to fit in the best way possible.

**Z1 = W1\*In1 + W2\*In2 + W3\*In3 + W4\*In4 + W5\*In5 + b**

W1, W2, W3, W4, W5 are the weights assigned to the inputs In1, In2, In3, In4, In5, and b is the bias.

* Then in the second step, the**activation function is applied to the linear equation Z1.** The activation function is a nonlinear transformation that is applied to the input before sending it to the next layer of neurons. The importance of the activation function is to inculcate nonlinearity in the model.

1. The whole process described in point 3 is performed in each hidden layer. After passing through every hidden layer, **we move to the last layer i.e our output layer which gives us the final output.**

**The process explained above is known as forwarding Propagation.**

1. After getting the predictions from the output layer, the **error is calculated i.e the difference between the actual and the predicted output.**

If the error is large, then the steps are taken to minimize the error and for the same purpose, **Back Propagation is performed.**

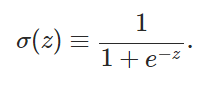
**Why Bias?**

* The bias node is a node that is always 'on'. That is, its value is set to 1 without regard for the data in a given pattern.
* It is analogous to the intercept in a regression model, and serves the same function.
* If a neural network does not have a bias node in a given layer, it will not be able to produce output in the next layer that differs from 0, when the feature values are 0
* Consider a simple example: You have a feed forward perceptron with 2 input nodes A and B, and 1 output node Y.
* “A” and “B” are binary features and set as, A=B=0.
* Multiply these 2 0’s by whatever weights you like, wA and wB, sum the products and pass it through whatever activation function you prefer. Without a bias node, only one output value is possible, which may yield a very poor fit.

**Perceptron Limitation**

* Suppose if we make a small change in some weight (or bias) in the network. What we'd like is - for this small change in weight to cause only a small corresponding change in the output from the network.
* The problem is that this isn't what happens when our network contains perceptrons. In fact, a small change in the weights or bias of any single perceptron in the network can sometimes cause the output of that perceptron to completely flip, say from 0 to 1.
* We can overcome this problem by introducing a new type of artificial neuron called a **sigmoid** neuron.
* Sigmoid neurons are similar to perceptrons, but modified so that small changes in their weights and bias cause only a small change in their output.

**Sigmoid Function:**

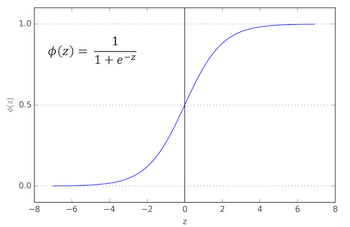
****

Also just like a perceptron, the sigmoid neuron has weights for each input, w1, w2, … and an overall bias, b. But the output is not 0 or 1. Instead, it's σ(w⋅x+b) and σ is called the sigmoid function (Activation Function or Transfer Function).

**Similarity b/w Perceptron & Sigmoid Neuron:**

* Suppose z=w.x+b is a large positive number. Then e−z≈0 and so σ(z)≈1. In other words, when z=w.x+b is large and positive, the output from the sigmoid neuron is approximately 1, just as it would have been for a perceptron.
* Suppose on the other hand that z=w.x+b is very negative. Then e−z→∞, and σ(z)≈0. So when z=w.x+b very negative, the behaviour of a sigmoid neuron also closely approximates a perceptron.

Chart, histogram

Description automatically generated

**Activation Functions:**

* In computational network, activation function of a node defines the output of that node given the input or the set of inputs.
* It can be as simple as a step function that turns the neuron output on and off, depending on a rule or threshold. Or it can be a transformation that maps the input signals into output signals that are needed for the neural network to function.
* Increasingly, neural networks use non-linear activation functions, which can help the network learn complex data, compute and learn almost any function representing a question, and provide accurate predictions.

**Step-wise Activation Function**

* A binary step function is a threshold-based activation function. If the input value is above or below a certain threshold, the neuron is activated and sends exactly the same signal to the next layer.
* The problem with a step function is that it does not allow multi-value outputs—for example, it cannot support classifying the inputs into one of several categories.

**Linear Activation Function**

* A linear activation function takes the form: **A = cx**
* It takes the inputs, multiplied by the weights for each neuron, and creates an output signal proportional to the input.
* In one sense, a linear function is better than a step function because it allows multiple outputs, not just yes and no**.**
* A neural network with a linear activation function is simply a linear regression model. It has limited power and ability to handle complexity varying parameters of input data.

**Non-Linear Activation Function**

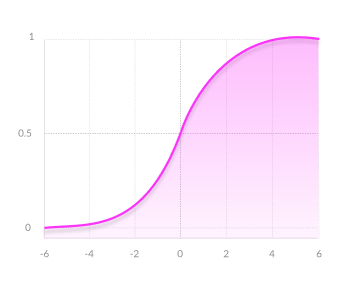
* Modern neural network models use non-linear activation functions.
* They allow the model to create complex mappings between the network’s inputs and outputs, which are essential for learning and modeling complex data, such as images, video, audio, and data sets which are non-linear or have high dimensionality.
* Non-linear functions address the problems of a linear activation function:
* They allow backpropagation because they have a derivative function which is related to the inputs.
* They allow “stacking” of multiple layers of neurons to create a deep neural network. Multiple hidden layers of neurons are needed to learn complex data sets with high levels of accuracy.

**Common Non-Linear Activation Functions**

* Sigmoid / Logistic
* TanH / Hyperbolic Tangent
* ReLU ( Rectified Linear Unit)
* Leaky ReLU
* Parametric ReLU
* Softmax

**Sigmoid Function:**

* **Output values bound** between 0 and 1, normalizing the output of each neuron.
* **Clear predictions**—For X above 2 or below -2, tends to bring the Y value (the prediction) to the edge of the curve, very close to 1 or 0. This enables clear predictions.

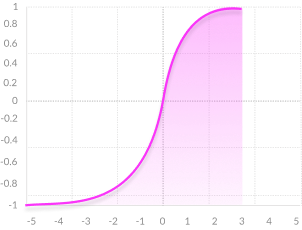


Disadvantages of Sigmoid:

* Vanishing gradient—for very high or very low values of X, there is almost no change to the prediction, causing a vanishing gradient problem. This can result in the network refusing to learn further, or being too slow to reach an accurate prediction.
* Outputs not zero centred
* Computationally expensive

**Tanh Function:**

* Zero centered—making it easier to model inputs that have strongly negative, neutral, and strongly positive values.
* Otherwise like the Sigmoid function.

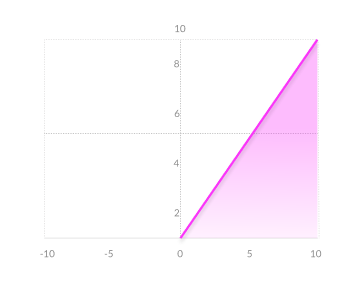
****

**Disadvantages :**

* **Vanishing gradient—**for very high or very low values of X, there is almost no change to the prediction, causing a vanishing gradient problem. This can result in the network refusing to learn further, or being too slow to reach an accurate prediction.
* Computationally expensive

**RelU Function:**

* **Computationally efficient—**allows the network to converge very quickly
* **Non-linear—**although it looks like a linear function, ReLU has a derivative function and allows for backpropagation

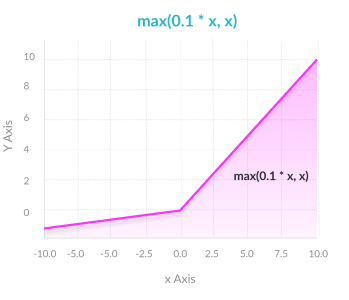


**Disadvantages:**

**The Dying ReLU problem—**when inputs approach zero, or are negative, the gradient of the function becomes zero, the network cannot perform backpropagation and cannot learn.

**Leaky RelU:**

* **Prevents dying ReLU problem—** this variation of ReLU has a small positive slope in the negative area, so it does enable backpropagation, even for negative input values
* Otherwise like ReLU

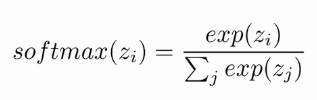


**Disadvantages:**

**Results not consistent—**leaky ReLU does not provide consistent predictions for negative input values.

**Softmax Function:**

* **Able to handle multiple classes** only one class in other activation functions—normalizes the outputs for each class between 0 and 1, and divides by their sum, giving the probability of the input value being in a specific class.
* **Useful for output neurons**—typically Softmax is used only for the output layer, for neural networks that need to classify inputs into multiple categories.



Here, the Z represents the values from the neurons of the output layer. The exponential acts as the non-linear function. Later these values are divided by the sum of exponential values in order to normalize and then convert them into probabilities.

Note that, when the number of classes is two, it becomes the same as the sigmoid activation function. In other words, sigmoid is simply a variant of the Softmax function.

**Matrix Calculus:**

Matrix calculus refers to a number of different notations that use matrices and vectors to collect the derivative of each component of the dependent variable with respect to each component of the independent variable. In general, the independent variable can be a scalar, a vector, or a matrix while the dependent variable can be any of these as well.

* **Jacobian Matrix: The Jacobian matrix is a matrix containing the first-order partial derivatives of a function. It gives us the slope of the function along multiple dimensions.**

**Diagram

Description automatically generated with low confidence**

* **Hessian matrices** belong to a class of mathematical structures that involve second order derivatives.

Table

Description automatically generated

## **Why Is The Hessian Matrix Important In Machine Learning?**

The Hessian matrix plays an important role in many machine learning algorithms, which involve optimizing a given function. While it may be expensive to compute, it holds some key information about the function being optimized. It can help determine the saddle points, and the local extremum of a function. It is used extensively in training neural networks and deep learning architectures.

**Gradient descent**

Gradient descent is a first-order iterative optimization algorithm for finding the minimum of a function. The goal of the gradient descent is to minimise a given function which, in our case, is the loss function of the neural network. To achieve this goal, it performs two steps iteratively.

1. Compute the slope (gradient) that is the first-order derivative of the function at the current point
2. Move-in the opposite direction of the slope increase from the current point by the computed amount

Diagram

Description automatically generated

# **Batch Gradient Descent**

In Batch Gradient Descent, all the training data is taken into consideration to take a single step. We take the average of the gradients of all the training examples and then use that mean gradient to update our parameters. So that’s just one step of gradient descent in one epoch.

Batch Gradient Descent is great for convex or relatively smooth error manifolds. In this case, we move somewhat directly towards an optimum solution.

Chart, line chart

Description automatically generated

The graph of cost vs epochs is also quite smooth because we are averaging over all the gradients of training data for a single step. The cost keeps on decreasing over the epochs.

# **Stochastic Gradient Descent**

In Batch Gradient Descent we were considering all the examples for every step of Gradient Descent. But what if our dataset is very huge. Deep learning models crave for data. The more the data the more chances of a model to be good. Suppose our dataset has 5 million examples, then just to take one step the model will have to calculate the gradients of all the 5 million examples. This does not seem an efficient way. To tackle this problem we have Stochastic Gradient Descent. In Stochastic Gradient Descent (SGD), we consider just one example at a time to take a single step. We do the following steps in **one epoch** for SGD:

1. Take an example
2. Feed it to Neural Network
3. Calculate it’s gradient
4. Use the gradient we calculated in step 3 to update the weights
5. Repeat steps 1–4 for all the examples in training dataset

Since we are considering just one example at a time the cost will fluctuate over the training examples and it will **not**necessarily decrease. But in the long run, you will see the cost decreasing with fluctuations.

Graphical user interface, shape

Description automatically generated with medium confidence

Also because the cost is so fluctuating, it will never reach the minima but it will keep dancing around it.

SGD can be used for larger datasets. It converges faster when the dataset is large as it causes updates to the parameters more frequently.

# **Mini Batch Gradient Descent**

We have seen the Batch Gradient Descent. We have also seen the Stochastic Gradient Descent. Batch Gradient Descent can be used for smoother curves. SGD can be used when the dataset is large. Batch Gradient Descent converges directly to minima. SGD converges faster for larger datasets. But, since in SGD we use only one example at a time, we cannot implement the vectorized implementation on it. This can slow down the computations. To tackle this problem, a mixture of Batch Gradient Descent and SGD is used.

Neither we use all the dataset all at once nor we use the single example at a time. We use a batch of a fixed number of training examples which is less than the actual dataset and call it a mini-batch. Doing this helps us achieve the advantages of both the former variants we saw. So, after creating the mini-batches of fixed size, we do the following steps in **one epoch:**

1. Pick a mini-batch
2. Feed it to Neural Network
3. Calculate the mean gradient of the mini-batch
4. Use the mean gradient we calculated in step 3 to update the weights
5. Repeat steps 1–4 for the mini-batches we created

Chart

Description automatically generated

Just like SGD, the average cost over the epochs in mini-batch gradient descent fluctuates because we are averaging a small number of examples at a time.

**Vanishing and Exploding Gradients:**

#### **Vanishing –**

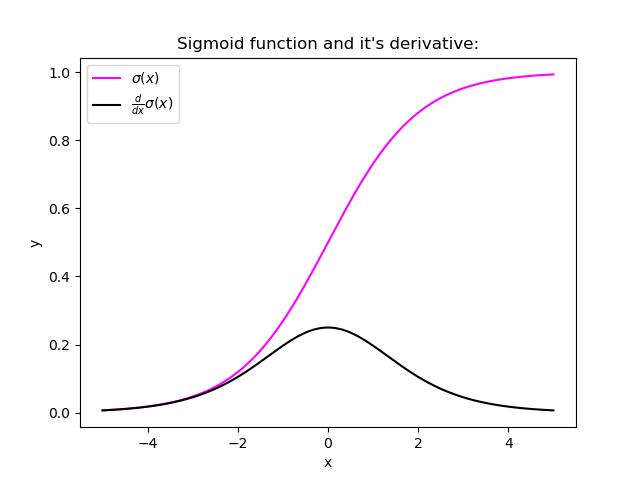
As the backpropagation algorithm advances downwards(or backward) from the output layer towards the input layer, the gradients often get smaller and smaller and approach zero which eventually leaves the weights of the initial or lower layers nearly unchanged. As a result, the gradient descent never converges to the optimum. This is known as the **vanishing gradients** problem.

#### **Exploding –**

On the contrary, in some cases, the gradients keep on getting larger and larger as the backpropagation algorithm progresses. This, in turn, causes very large weight updates and causes the gradient descent to diverge. This is known as the **exploding gradients** problem.

## ***Why do the gradients even vanish/explode?***

Certain activation functions, like the logistic function (sigmoid), have a very huge difference between the variance of their inputs and the outputs. In simpler words, they shrink and transform a larger input space into a smaller output space that lies between the range of [0,1].



Observing the above graph of the Sigmoid function, we can see that for larger inputs (negative or positive), it saturates at 0 or 1 with a derivative very close to zero. Thus, when the backpropagation algorithm chips in, it virtually has no gradients to propagate backward in the network, and whatever little residual gradients exist keeps on diluting as the algorithm progresses down through the top layers. So, this leaves nothing for the lower layers.

Similarly, in some cases suppose the initial weights assigned to the network generate some large loss. Now the gradients can accumulate during an update and result in very large gradients which eventually results in large updates to the network weights and leads to an unstable network. The parameters can sometimes become so large that they overflow and result in NaN values.

**Overfitting of Neural Networks:**

In Overfitting, the model tries to learn too many details in the training data along with the noise from the training data. As a result, the model performance is very poor on unseen or test datasets. Therefore, the network fails to generalize the features or patterns present in the training dataset.

### ****Methods to Avoid Overfitting of a Model****

You can identify that your model is not right when it works well on training data but does not perform well on unseen and new data. You can also track the performance of the model performance through concepts like [bias and variance](https://analyticsindiamag.com/how-an-unethical-optimisation-principle-can-help-businesses-prevent-ai-biases/). But how to solve this problem? Here are some of the techniques you can use to effectively overcome the overfitting problem in your neural network.

1. [**Data Augmentation**](https://analyticsindiamag.com/why-does-image-data-augmentation-work-as-a-regularizer-in-deep-learning/)**:**Diversity of data and a larger dataset is the easiest way to avoid overfitting of the model. Data augmentation allows you to increase the size of your dataset by performing processes like flipping, cropping, rotation, scaling and translation on the existing images. Data augmentation not only increases the dataset size but also exposes the model to different angles and lighting and reduces the bias in the dataset, thus avoiding chances of overfitting.

**2. Regularization Techniques:**This method involves adding an extra element to the loss function. This extra element acts as a critic which punishes the model for using higher weights than needed. As the complexity of the model increases, a penalty is added in the loss function that helps in limiting the flexibility of the model. The two popular methods of regularization are the L1 and L2 regularization methods.  L1 regularization reduces the weight values of less important features to zero and eliminates them from further calculations. L2 regularization aims to minimize the magnitude of weights by squaring the weight. The disadvantage is that if there are large numbers of outliers, the square increases the magnitude of the outliers as well and the model tends to not perform as well as it would with L1 regularization. With an increase in penalty value, the cost function performs weight tweaking and reduces the increase and therefore reduces the loss and overfitting.

**3. Dropouts:**Regularization techniques prevent the model from overfitting by modifying the cost function. Dropout, on the other hand, prevents overfitting by modifying the network itself. It works as follows. Every neuron apart from the ones in the output layer is assigned a probability p of being temporarily ignored from calculations. p is also called dropout rate and is usually initialized to 0.5. Then, as each iteration progresses, the neurons in each layer with the highest probability get dropped. This results in creating a smaller network with each epoch. Since in each iteration, a random input value can be eliminated, the network tries to balance the risk and not to favour any of the features and reduces bias and noise.

**4. Early Stopping:**Early stopping is a technique that can avoid over-training and hence overfitting of the model. An over-trained model has a tendency to memorize all the training data points. With early stopping, a large arbitrary number of training epochs is specified. The model is stopped from training further when the model performance stops improving on the validation dataset.

**Naïve-Bayes Classifier:** Refer to PPTs uploaded in LMS.

**SVM:** Refer to PPTs uploaded in MLS