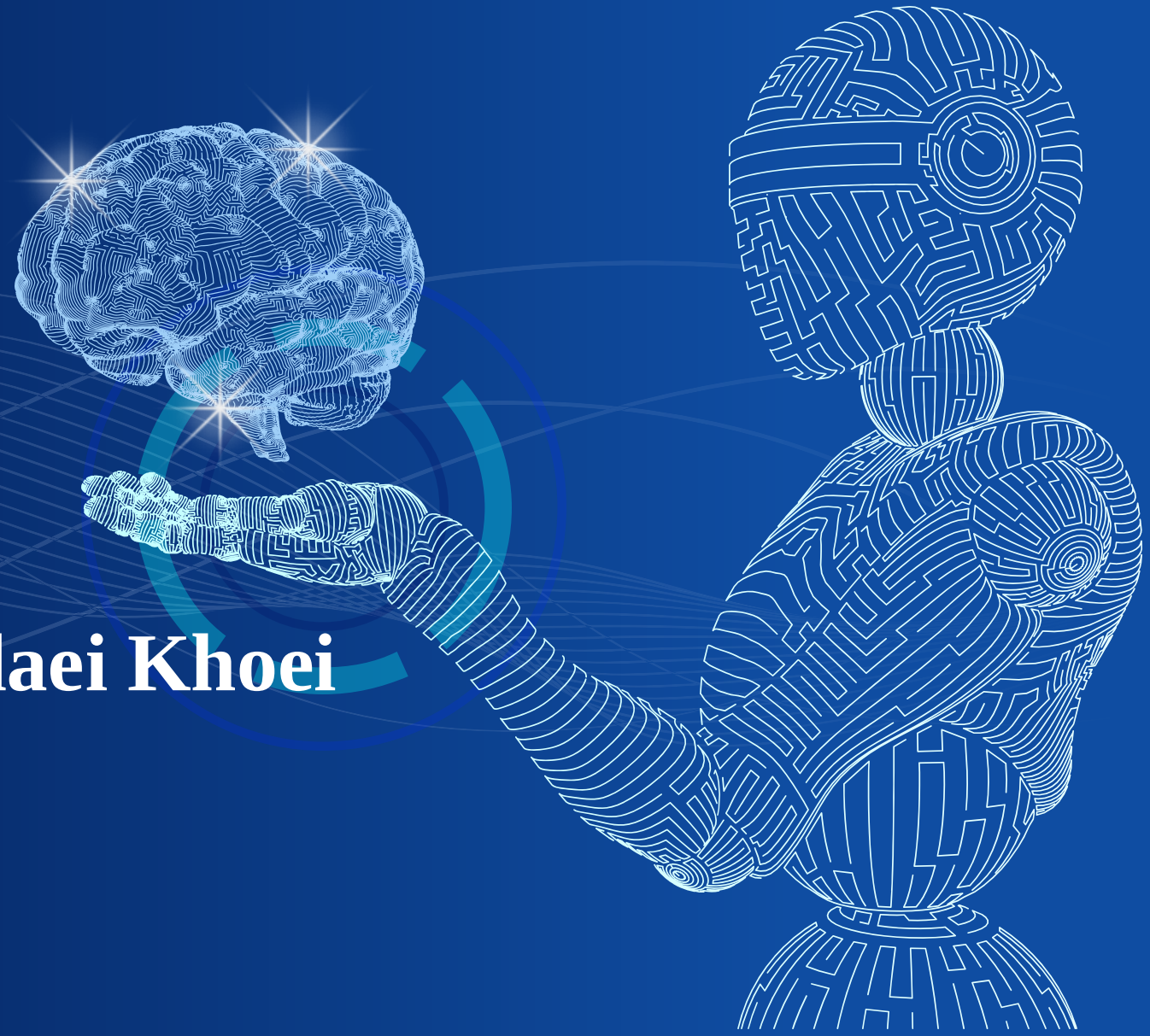
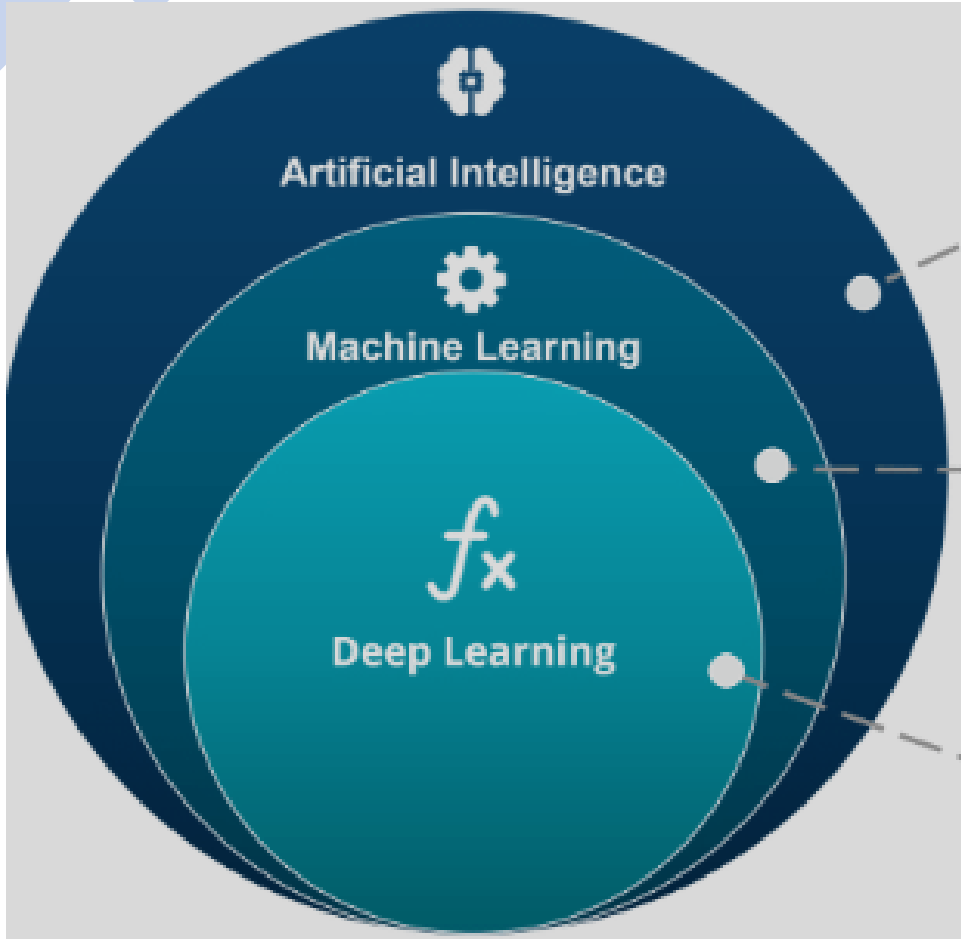
CS 7150: Deep Learning



Presented by: Dr. Tala Talaei Khoei



ARTIFICIAL INTELLIGENCE

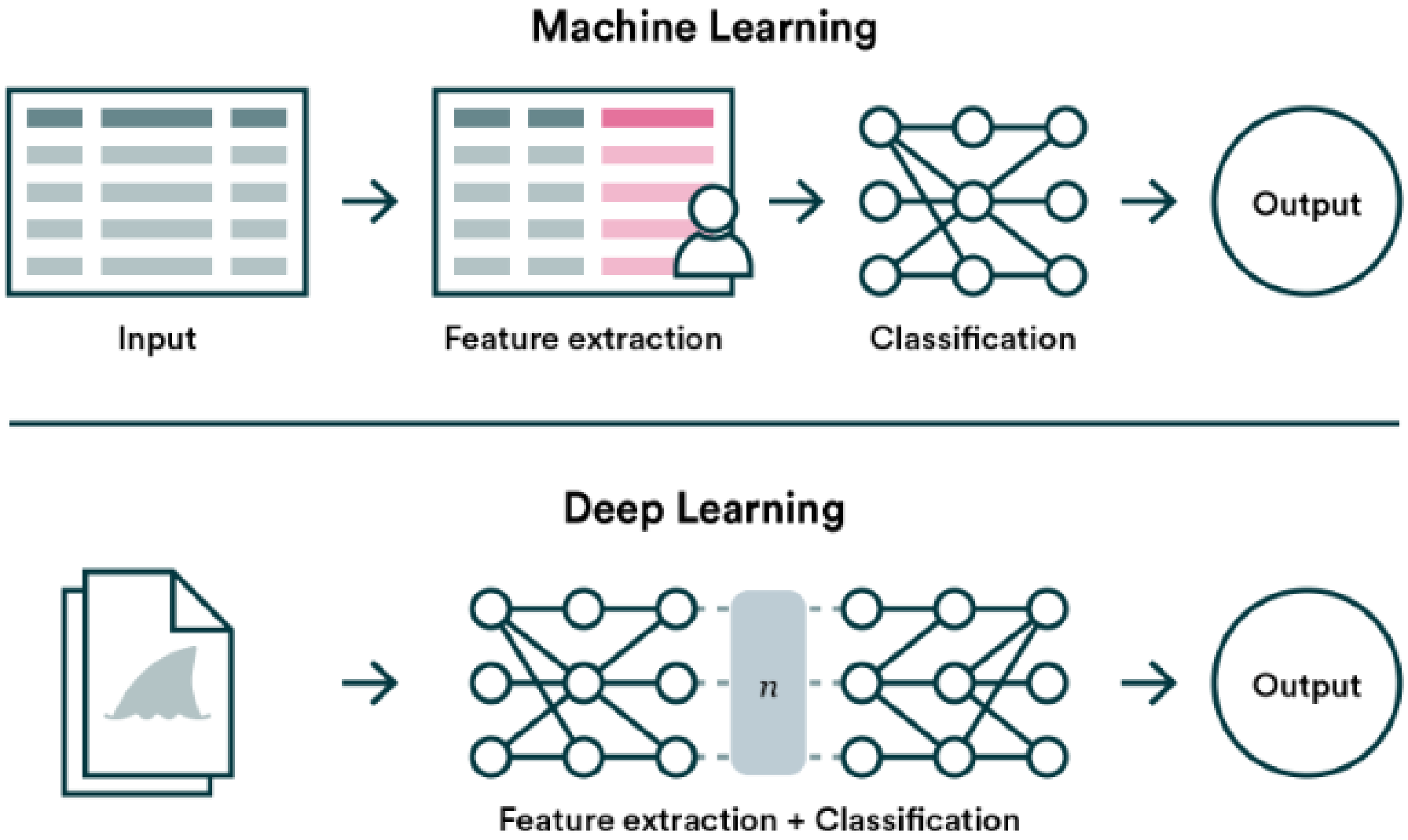
A technique which enables machines to mimic human behaviour

MACHINE LEARNING

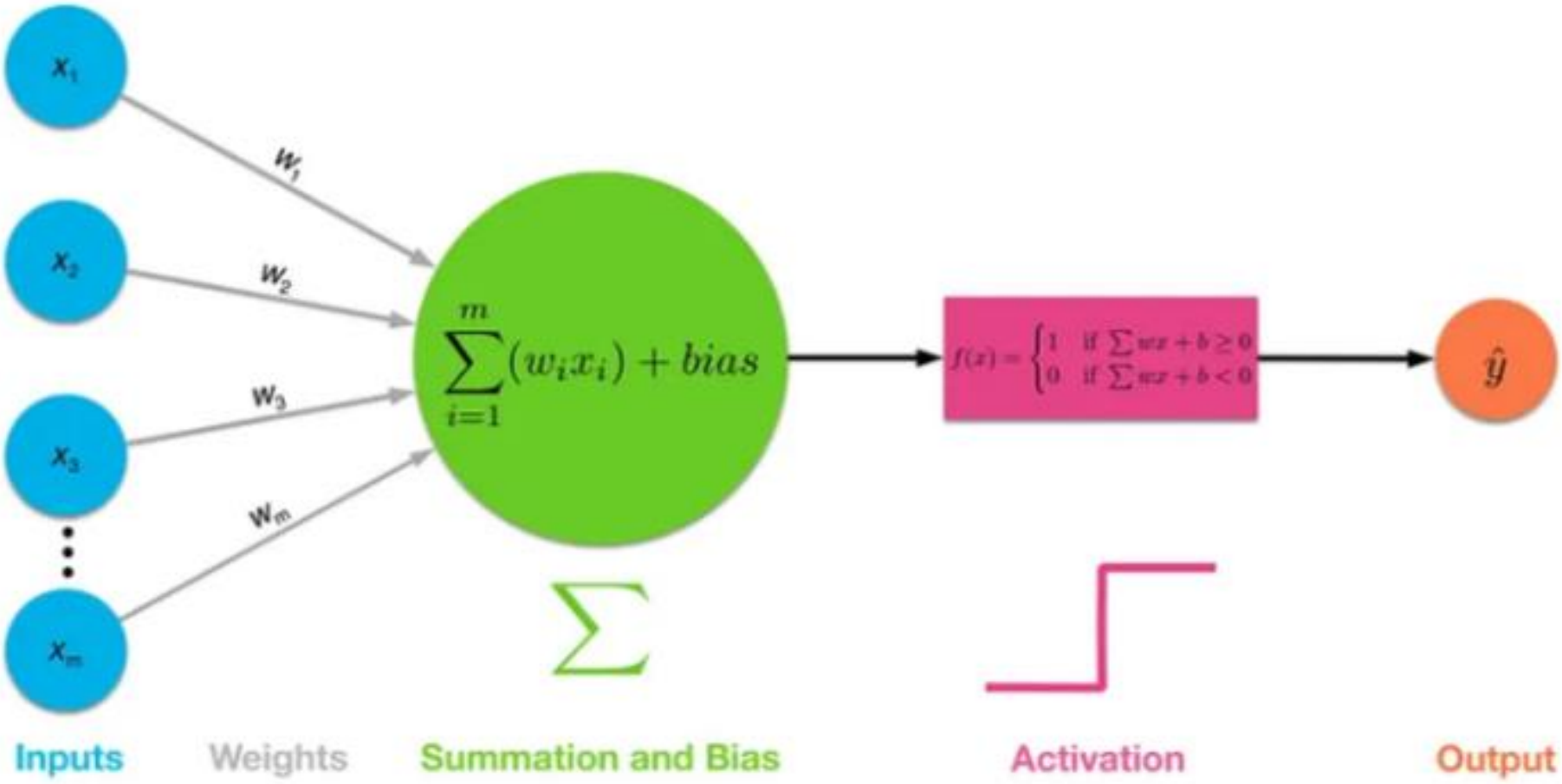
Subset of AI technique which use statistical methods to enable machines to improve with experience

DEEP LEARNING

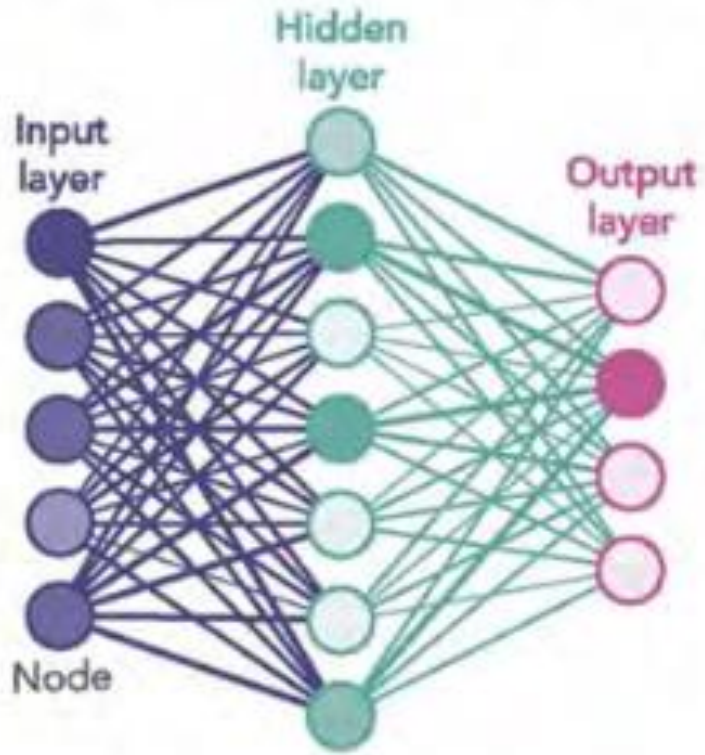
Subset of ML which make the computation of multi-layer neural network feasible


Machine Learning
Input
Feature extraction
Classification
Output
Deep Learning
n
Feature extraction + Classification
Output

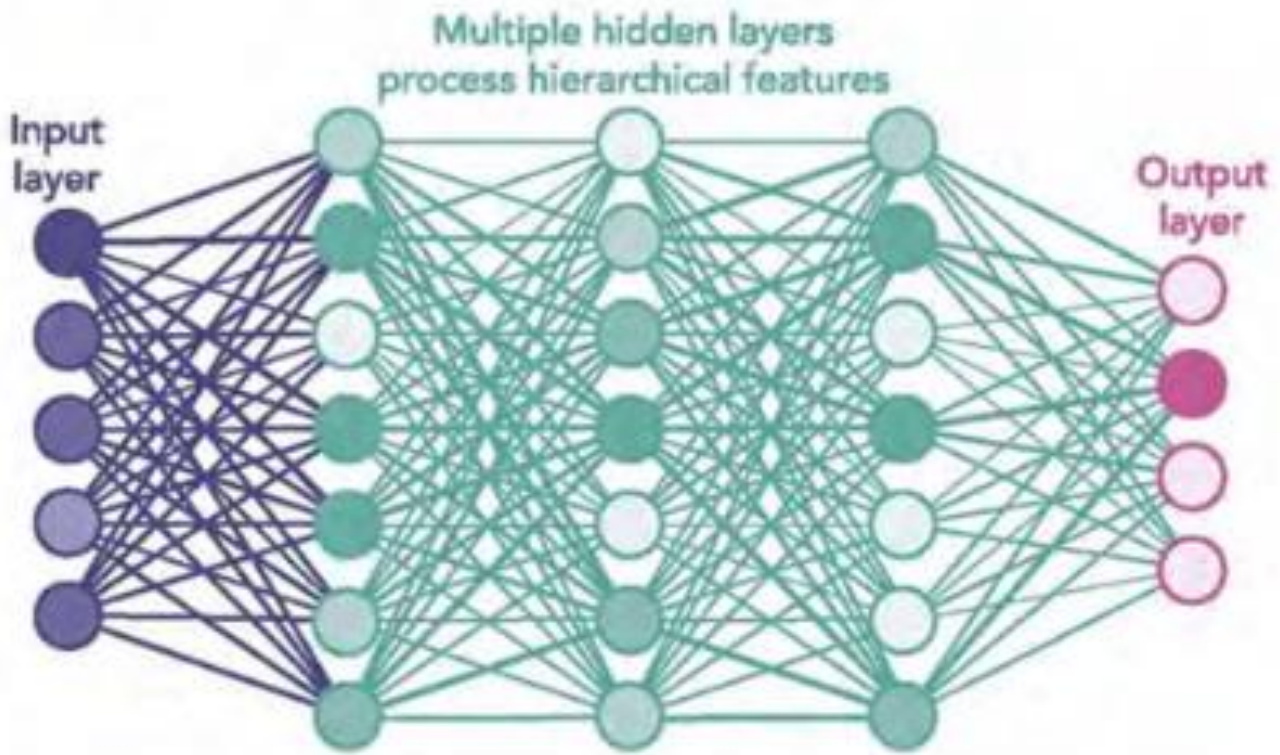
Single-layer Perceptron Network



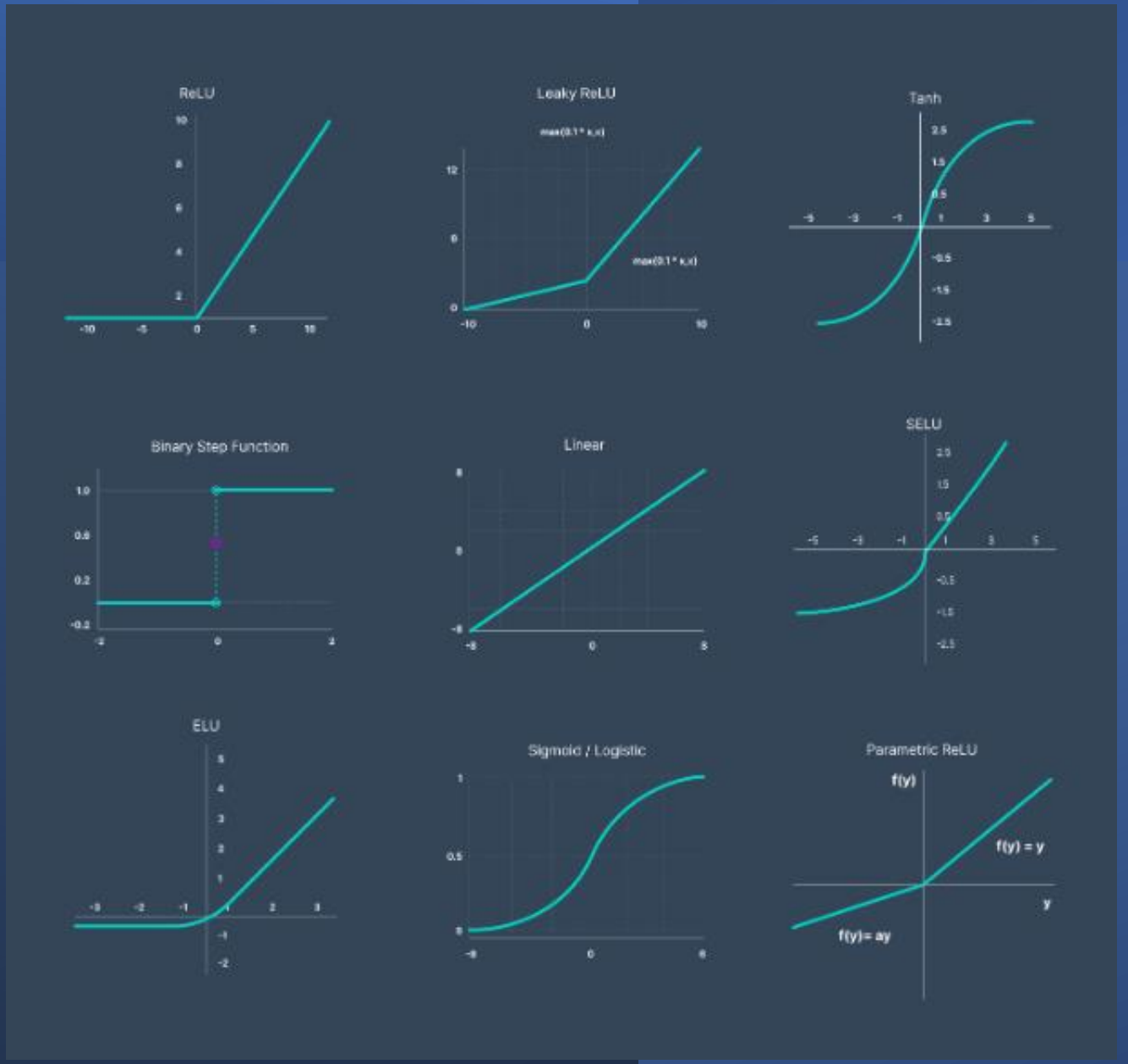
SHALLOW NEURAL NETWORK



DEEP NEURAL NETWORK



Activation Functions

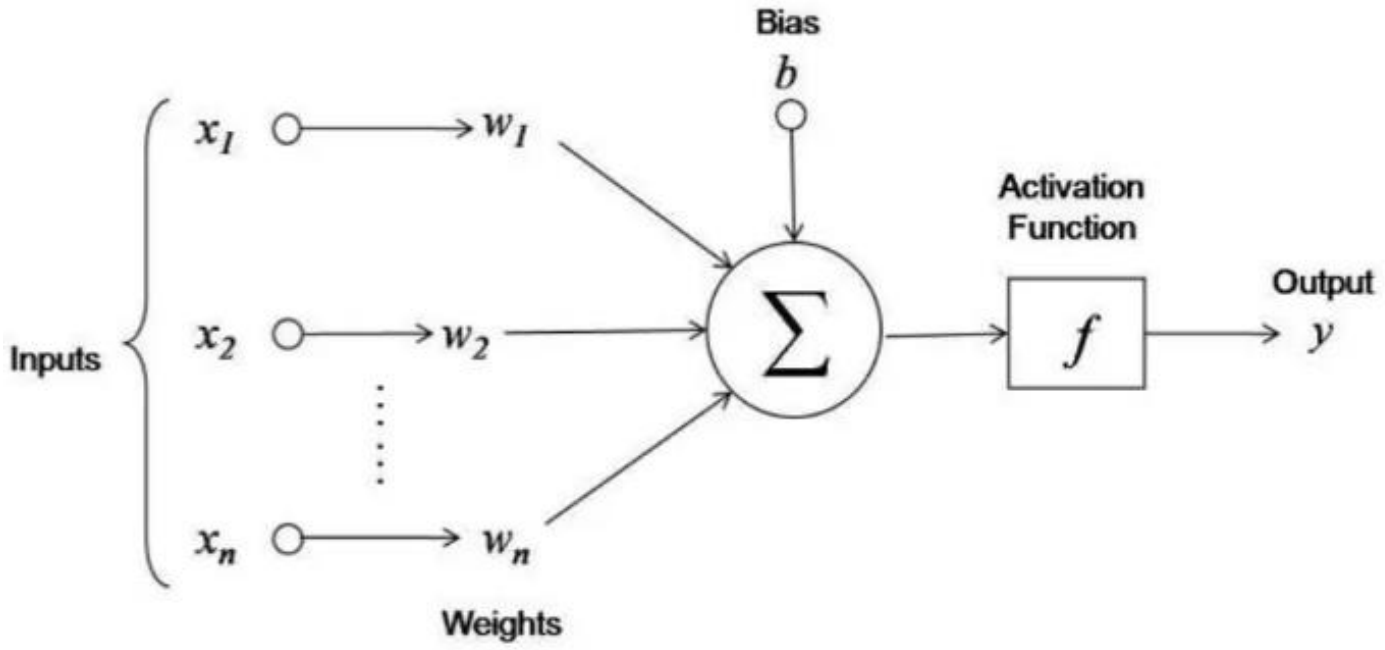


Activation Functions

- Activation function in neural networks is a mathematical function that determines the output of a neuron based on its input.
- As the name suggests, it is some kind of function that should "activate" the neuron.
- Whether it will be convolutional neural networks or recurrent neural networks, the activation function decides how to proceed.
- Just as neurons in the brain receive signals from the body and make decisions on how to process them, neurons in artificial neural networks work in a similar manner.
- They act as transfer functions, receiving input values and producing corresponding output values.

How do activation functions work?

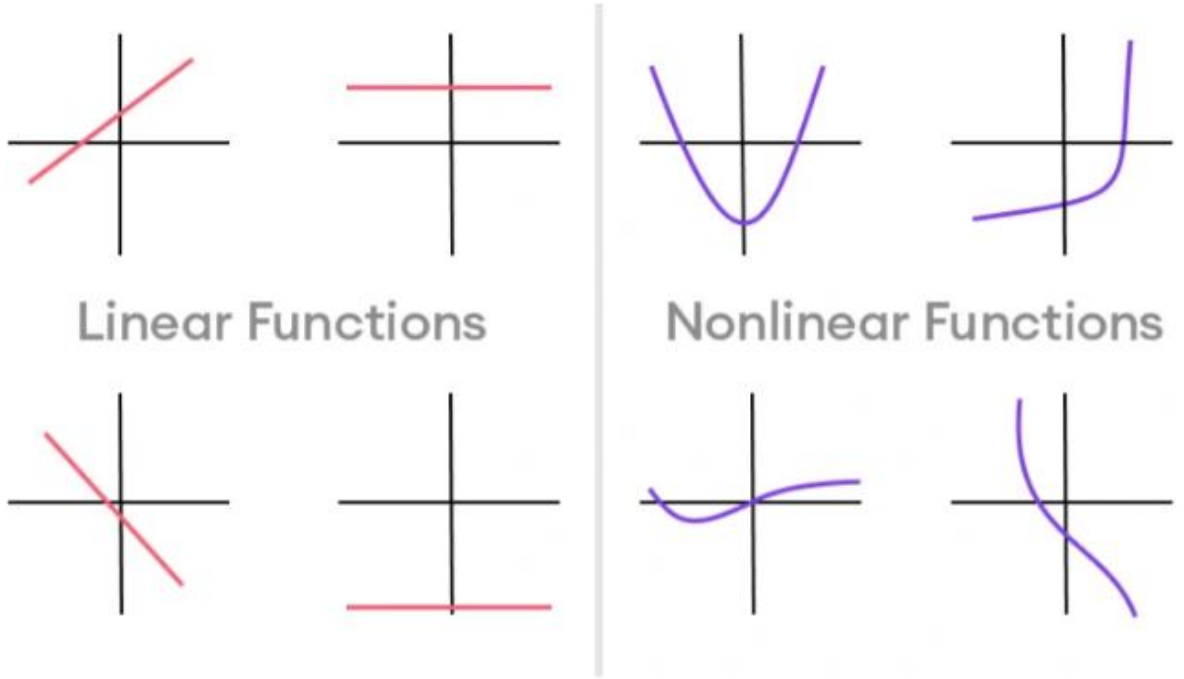
- Regardless of network architecture, an activation function will take the values generated by a given network layer (in a fully connected network, this would be the sum of weights and biases) and apply a certain transformation to these values to map them to a specific range.



- As you see in figure, after taking a weighted sum of the inputs plus the bias ($W_1X_1 + W_2*X_2 + \ldots + Wn*Xn + b$), we pass this value to some activation function f, which then gives us the output of the given neuron. In this case, each of the X_i values is the output of a neuron from the previous layer, while W_i is our neuron's weights assigned to each input X_i.

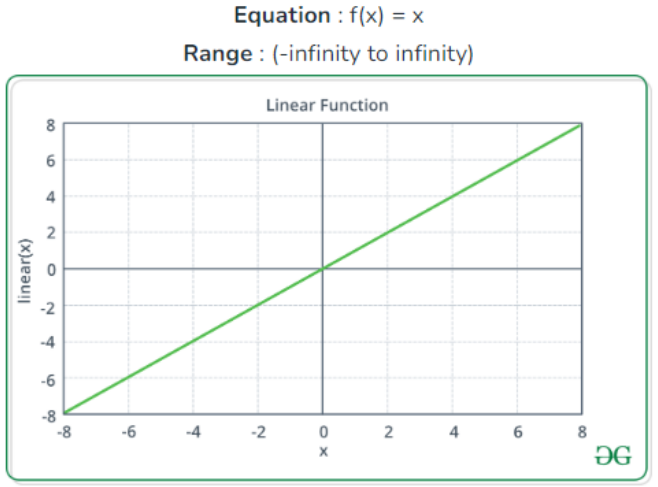
Why use an activation function?

- While not all activation functions are non-linear, the overwhelming majority are, and for a good reason.
- Nonlinear activation functions help introduce additional complexity into neural networks and facilitate them to “learn” to approximate a much larger swathe of functions.
- If not for nonlinear activation functions, neural networks would only be able to learn linear and affine functions since the layers would be linearly dependent on each other and would just comprise a glorified affine function.
- Another important aspect of activation functions is that they allow us to map an input stream of unknown distribution and scale it to a known one (e.g., the sigmoid function maps any input to a value between 0 and 1).
- This helps stabilize the training of neural networks and also helps map the values to our desired output in the output layer (for non-regression tasks).



Types of Activation Functions

1. Linear Activation Function



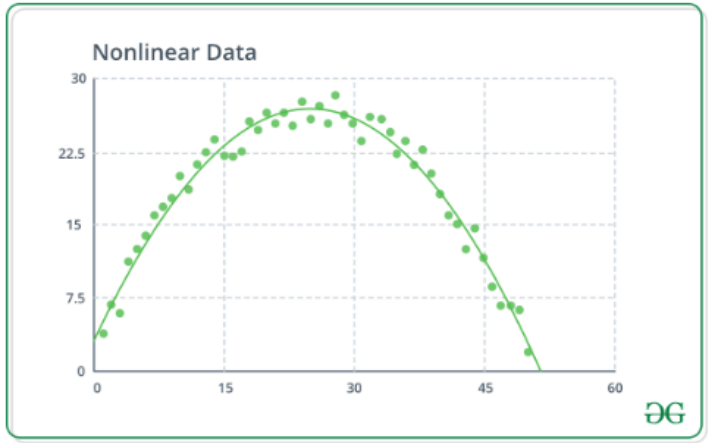
2. Non-Linear Activation Function

- It makes it easy for the model to generalize with a variety of data and to differentiate between the output.
- Non-linear means that the output cannot be reproduced from a linear combination of the inputs.

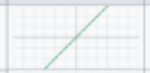
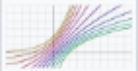
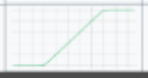
The main terminologies needed to understand for nonlinear functions are:

✓ **Derivative:** Change in y-axis w.r.t. change in x-axis. It is also known as slope.

✓ **Monotonic function:** A function which is either entirely non-increasing or non-decreasing



The Nonlinear Activation Functions are mainly divided on the basis of their range or curves as follows:

Name	Plot	Equation	Derivative	Range	Order of continuity
Identity		$f(x) = x$	$f'(x) = 1$	$(-\infty, \infty)$	C^∞
Binary step		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x \neq 0 \\ ? & \text{for } x = 0 \end{cases}$	$\{0, 1\}$	C^{-1}
Logistic (a.k.a. Sigmoid or Soft step)		$f(x) = \sigma(x) = \frac{1}{1+e^{-x}}$	$f'(x) = f(x)(1 - f(x))$	$(0, 1)$	C^∞
TanH		$f(x) = \tanh(x) = \frac{(e^x - e^{-x})}{(e^x + e^{-x})}$	$f'(x) = 1 - f(x)^2$	$(-1, 1)$	C^∞
ElliotSig Softsign		$f(x) = \frac{x}{1+\lvert x \rvert}$	$f'(x) = \frac{1}{(1+\lvert x \rvert)^2}$	$(-1, 1)$	C^1
Square Nonlinearity (SQNL)		$f(x) = \begin{cases} 1 & : x > 2.0 \\ x - \frac{x^2}{4} & : 0 \leq x \leq 2.0 \\ x + \frac{x^2}{4} & : -2.0 \leq x < 0 \\ -1 & : x < -2.0 \end{cases}$	$f'(x) = 1 \mp \frac{x}{2}$	$(-1, 1)$	C^2
Rectified linear unit (ReLU)		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$[0, \infty)$	C^0
Bipolar rectified linear unit (BReLU)		$f(x_i) = \begin{cases} ReLU(x_i) & \text{if } i \bmod 2 = 0 \\ -ReLU(-x_i) & \text{if } i \bmod 2 \neq 0 \end{cases}$	$f'(x_i) = \begin{cases} ReLU'(x_i) & \text{if } i \bmod 2 = 0 \\ -ReLU'(-x_i) & \text{if } i \bmod 2 \neq 0 \end{cases}$	$(-\infty, \infty)$	C^0
Leaky rectified linear unit (Leaky ReLU)		$f(x) = \begin{cases} 0.01x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0.01 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$(-\infty, \infty)$	C^0
Parameteric rectified linear unit (PReLU)		$f(\alpha, x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(\alpha, x) = \begin{cases} \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$(-\infty, \infty)$	C^0
Randomized leaky rectified linear unit (RReLU)		$f(\alpha, x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(\alpha, x) = \begin{cases} \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$(-\infty, \infty)$	C^0
Exponential linear unit (ELU)		$f(\alpha, x) = \begin{cases} \alpha(e^x - 1) & \text{for } x \leq 0 \\ x & \text{for } x > 0 \end{cases}$	$f'(\alpha, x) = \begin{cases} f(\alpha, x) + \alpha & \text{for } x \leq 0 \\ 1 & \text{for } x > 0 \end{cases}$	$(-\alpha, \infty)$	$\begin{cases} C_1 & \text{when } \alpha = 1 \\ C_0 & \text{otherwise} \end{cases}$
Adaptive piecewise linear (APL)		$f(x) = \max(0, x) + \sum_{s=1}^{S} a_i^s \max(0, -x + b_i^s)$	$f'(x) = H(x) - \sum_{s=1}^{S} a_i^s H(-x + b_i^s)^1$	$(-\infty, \infty)$	C^0
Bent identity		$f(x) = \frac{\sqrt{x^2+1}-1}{2} + x$	$f'(x) = \frac{x}{2\sqrt{x^2+1}} + 1$	$(-\infty, \infty)$	C^∞
Sigmoid Linear Unit (SiLU) (AKA SiL and Swish-1)		$f(x) = x \cdot \sigma(x)$	$f'(x) = f(x) + \sigma(x)(1 - f(x))$	$[\approx -0.28, \infty)$	C^∞
SoftExponential		$f(\alpha, x) = \begin{cases} -\frac{\ln(1-\alpha(x+\alpha))}{\alpha} & \text{for } \alpha < 0 \\ x & \text{for } \alpha = 0 \\ \frac{e^{\alpha x}-1}{\alpha} + \alpha & \text{for } \alpha > 0 \end{cases}$	$f'(\alpha, x) = \begin{cases} \frac{1}{1-\alpha(\alpha+x)} & \text{for } \alpha < 0 \\ e^{\alpha x} & \text{for } \alpha \geq 0 \end{cases}$	$(-\infty, \infty)$	C^∞
Soft Clipping		$f(\alpha, x) = \frac{1}{\alpha} \log \frac{1+e^{\alpha x}}{1+e^{\alpha(x-1)}}$	$f'(\alpha, x) = \frac{1}{2} \sinh\left(\frac{p}{2}\right) \operatorname{sech}\left(\frac{px}{2}\right) \operatorname{sech}\left(\frac{p}{2}(1-x)\right)$	$(0, 1)$	C^∞

Note: For details of each technique, refer to the uploaded document in Canvas

How to choose an activation function?

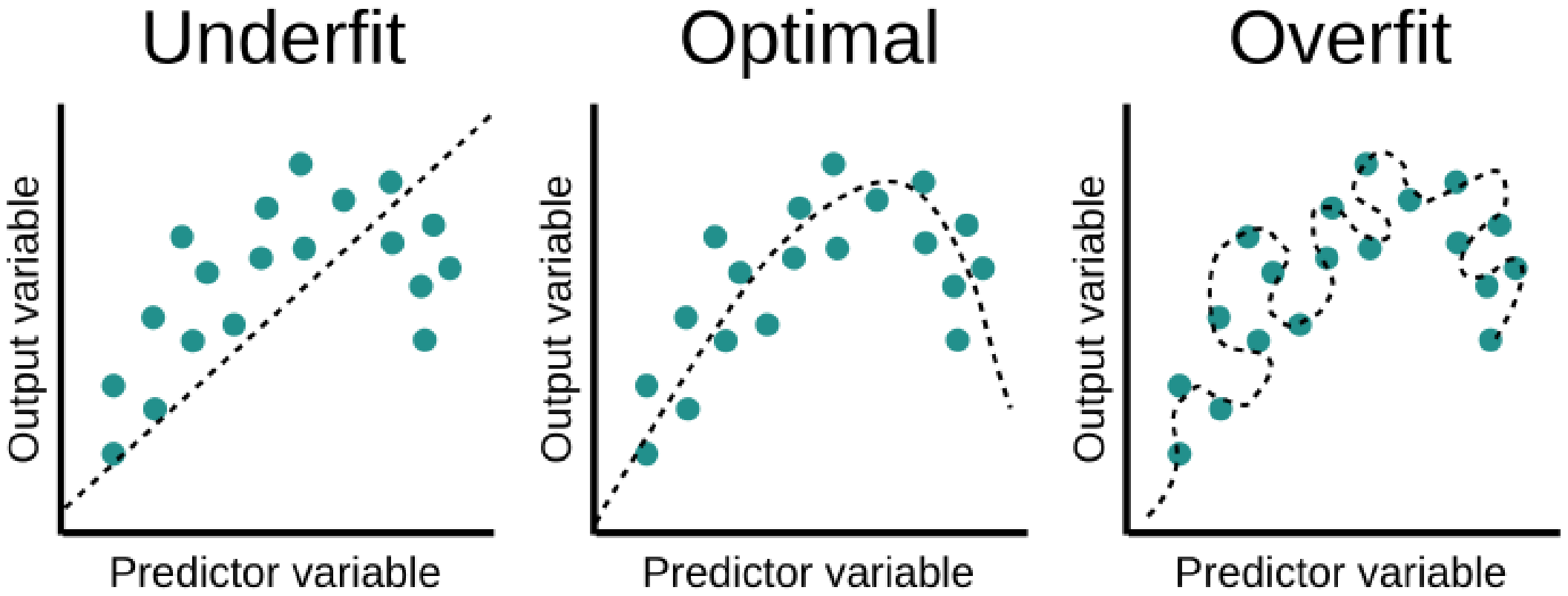
- ✓ Regression: Linear Activation Function
- ✓ Binary Classification: Sigmoid/Logistic Activation Function
- ✓ Multiclass Classification: SoftMax
- ✓ Multilabel Classification: Sigmoid

- ✓ The activation function used in hidden layers is typically chosen based on the type of neural network architecture.

- ✓ **Convolutional Neural Network (CNN)**: ReLU activation function.
- ✓ **Recurrent Neural Network**: Tanh and/or Sigmoid activation function

Example of Python implementation of Activation function for Neural Network

```python
import numpy as np

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def tanh(x):
    return np.tanh(x)

def relu(x):
    return np.maximum(0, x)

def leaky_relu(x, alpha=0.01):
    return np.maximum(alpha * x, x)

def softmax(x):
    exp_x = np.exp(x)
    return exp_x / np.sum(exp_x)
```


Underfit
Optimal
Overfit
Output variable
Predictor variable
Output variable
Predictor variable
Output variable
Predictor variable

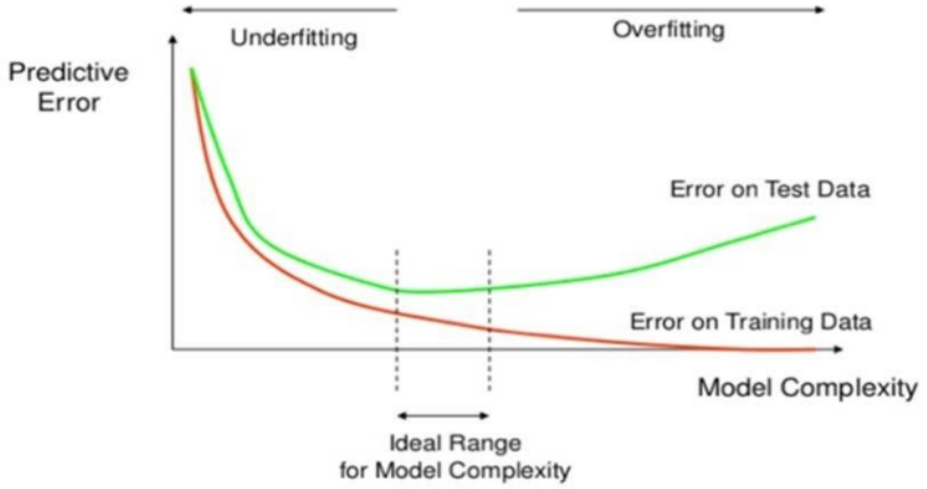
Correct fit vs. overfit models

- When a machine learning model is provided with training samples along with corresponding labels, the model will start to recognize patterns in the data and update the model parameters accordingly. The process is known as training. These parameters or weights are then used to predict the labels or outputs on another set of unseen samples that the model has not been trained on, i.e., the testing dataset. This process is called inference.

- If the model can perform well on the testing dataset, the model can be said to have generalized well, i.e., correctly understood the patterns provided in the training dataset. This type of model is called a correct fit model. However, if the model performs well on the training data and doesn't perform well on the testing data, it can be concluded that the model has memorized the patterns of training data but is not able to generalize well on unseen data. This model is called an overfit model.

- To summarize, overfitting is a phenomenon where the machine learning model learns patterns and performs well on data that it has been trained on and does not perform well on unseen data.

Note: The graph shows that as the model is trained for a longer duration, the training error lessens. However, the testing error starts increasing after a specific point. This indicates that the model has started to overfit.



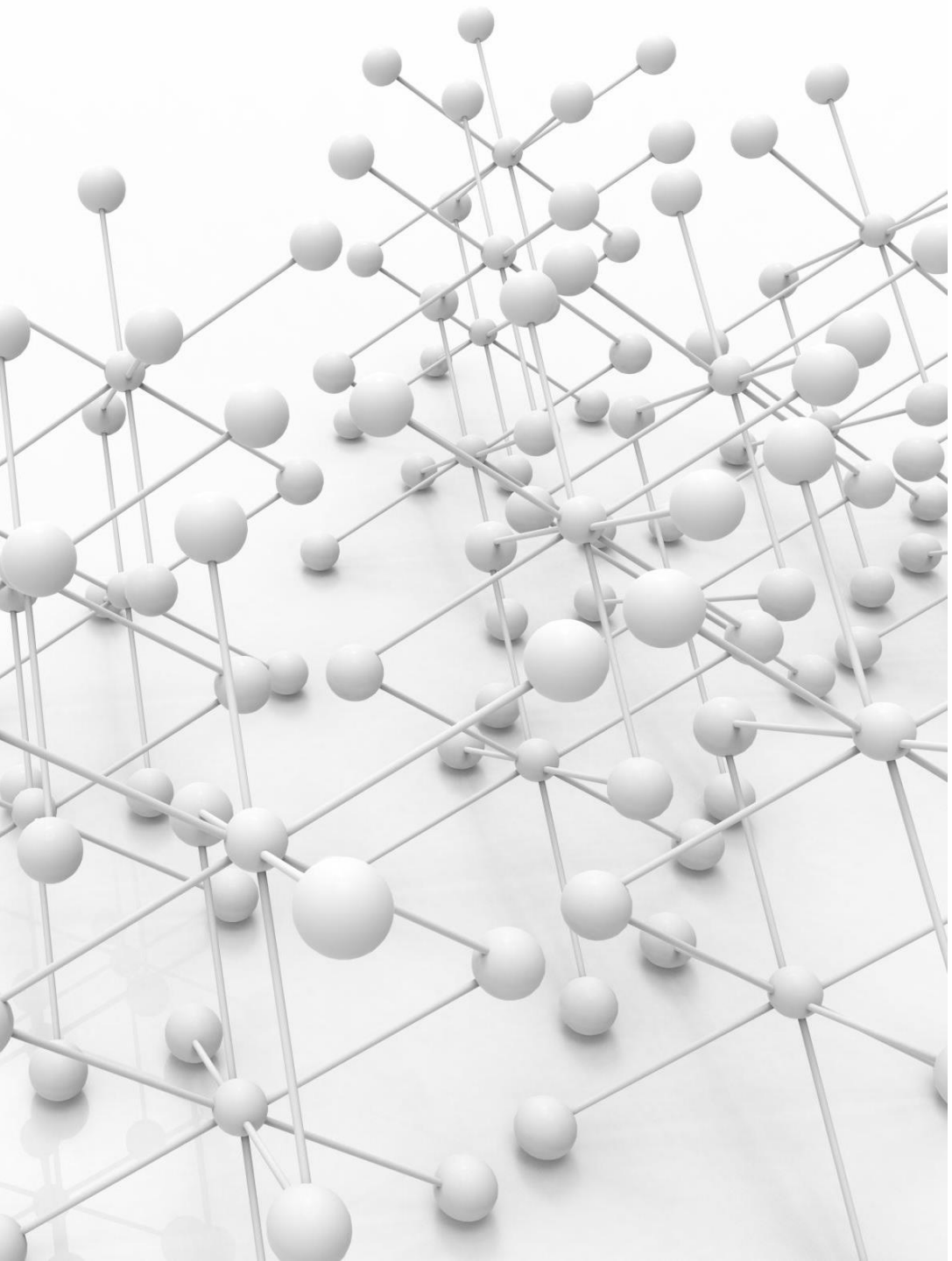
Methods to Handle Overfitting

Overfitting is caused when the training accuracy/metric is relatively higher than validation accuracy/metric. It can be handled through the following techniques:

1. *Training on more training data* to better identify the patterns.
2. *Data augmentation* for better model generalization.
3. *Early stopping*, i.e., stop the model training when the validation metrics start
4. decreasing or loss starts increasing.
5. *Regularization techniques*.

Note: In these techniques, data augmentation and more training data don't change the model architecture but try to improve the performance by altering the input data. Early stopping is used to stop the model training at an appropriate time - before the model overfits, rather than addressing the issue of overfitting directly. However, regularization is a more robust technique that can be used to avoid overfitting.


Regularization in
Neural Networks



Regularization

- Regularization is a set of techniques that can prevent overfitting in neural networks and thus improve the accuracy of a Deep Learning model when facing completely new data from the problem domain.

- In other words, regularization refers to the set of techniques that lower the complexity of the neural network model during training and prevent overfitting.

Types of regularization techniques

- Regularization is a technique used to address overfitting by directly changing the architecture of the model by modifying the model's training process. The following are the commonly used regularization techniques:

✓ ***L2 regularization***

According to regression analysis, L2 regularization is also called ridge regression. In this type of regularization, the squared magnitude of the coefficients or weights multiplied with a regularizer term is added to the loss or cost function. L2 regression can be represented with the following mathematical equation.

Loss:

$$\underbrace{\sum_{i=1}^{n}\left(y_i - \sum_{j=1}^{p} x_{ij} b_j\right)^2}_{Loss\ term} + \underbrace{\lambda \cdot \left(\sum_{j=1}^{p} b_j\right)^2}_{Regularizer\ term}$$

In the above equation,

y_i – $labels$

x_{ij} – $features$

b_j – $weights$

λ – $regularizer\ term\ (lambda)$

i – $no\ of\ training\ samples$

Note:

- Lambda is the hyperparameter that is tuned to prevent overfitting i.e. penalize the insignificant weights by forcing them to be small but not zero.
- L2 regularization works best when all the weights are roughly of the same size, i.e., input features are of the same range.
- This technique also helps the model to learn more complex patterns from data without overfitting easily.

✓ *L1 regularization*

L1 regularization is also referred to as lasso regression. In this type of regularization, the absolute value of the magnitude of coefficients or weights multiplied with a regularizer term is added to the loss or cost function. It can be represented with the following equation.

Loss:

$$\underbrace{\sum_{i=1}^{n}\left(y_i - \sum_{j=1}^{p} x_{ij} b_j\right)^2}_{Loss\ Term} + \underbrace{\lambda \cdot \sum_{j=1}^{p} |b_j|}_{Regularizer\ term}$$

In the above equation,

y_i – *labels*

x_{ij} – *features*

b_j – *weights*

λ – *regularizer term (lambda)*

i – *no of training samples*

Note:

- Since the L1 regularization adds an absolute value as a penalty to the cost function, the feature selection will be done by retaining only some important features and eliminating the lower or unimportant features.
- This technique is also robust to outliers, i.e., the model will be able to easily learn about outliers in the dataset.
- This technique will not be able to learn complex patterns from the input data.

Regularization Type	Advantages	Disadvantages
L1	1. Performs feature selection by shrinking less important feature weights to zero.	1. Not effective for datasets with many important features.
	2. Can be used for high-dimensional datasets with many irrelevant features.	2. The solution may not be unique, which can lead to instability in the model.
L2	1. Provides a smooth solution and improves the generalization performance of the model.	1. May not perform well for datasets with many irrelevant features.
	2. Can handle datasets with many important features.	2. Does not perform feature selection, and all features are included in the model with non-zero weights.

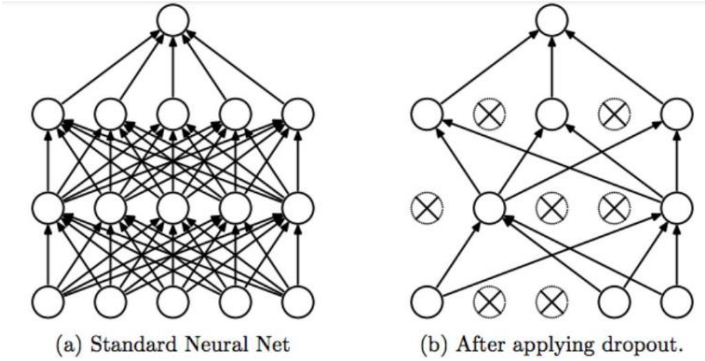
Dropout regularization

Dropout regularization is the technique in which some of the neurons are randomly disabled during the training such that the model can extract more useful robust features from the model. This prevents overfitting. You can see the dropout regularization in the following diagram:

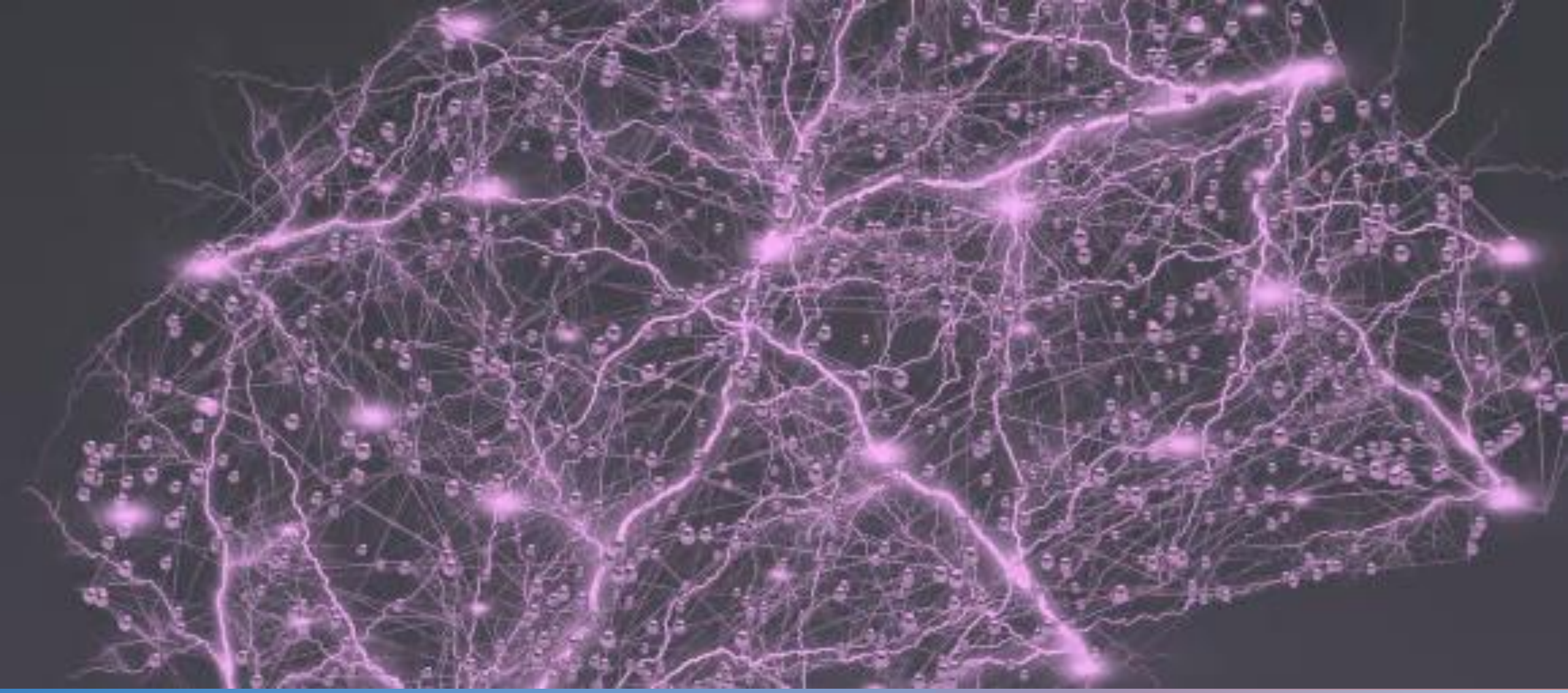
- In figure (a), the neural network is fully connected. If all the neurons are trained with the entire training dataset, some neurons might memorize the patterns occurring in training data. This leads to overfitting since the model is not generalizing well.
- In figure (b), the neural network is sparsely connected, i.e., only some neurons are active during the model training. This forces the neurons to extract robust features/patterns from training data to prevent overfitting.

The following are the characteristics of dropout regularization:

- ✓ Dropout randomly disables some percent of neurons in each layer. So for every epoch, different neurons will be dropped leading to effective learning.
- ✓ Dropout is applied by specifying the ‘p’ values, which is the fraction of neurons to be dropped.
- ✓ Dropout reduces the dependencies of neurons on other neurons, resulting in more robust model behavior.
- ✓ Dropout is applied only during the model training phase and is not applied during the inference phase.
- ✓ When the model receives complete data during the inference time, you need to scale the layer outputs ‘x’ by ‘p’ such that only some parts of data will be sent to the next layer. This is because the layers have seen less amount of data as specified by dropout.



(a) Standard Neural Net (b) After applying dropout.



Batch Normalization in Deep Neural Networks

Batch Normalization

- Batch normalization is a deep learning approach that has been shown to significantly improve the efficiency and reliability of neural network models. It is particularly useful for training very deep networks, as it can help to reduce the internal covariate shift that can occur during training.

- Batch normalization is a supervised learning method for normalizing the interlayer outputs of a neural network. As a result, the next layer receives a "reset" of the output distribution from the preceding layer, allowing it to analyze the data more effectively.

- The term "internal covariate shift" is used to describe the effect that updating the parameters of the layers above it has on the distribution of inputs to the current layer during deep learning training. This can make the optimization process more difficult and can slow down the convergence of the model.

- Since normalization guarantees that no activation value is too high or too low, and since it enables each layer to learn independently from the others, this strategy leads to quicker learning rates.

Internal Covariate Shift *is the change in the distribution of network activations due to the change in network parameters during training.*

Batch normalization is a technique for training very deep neural networks that standardizes the inputs to a layer for each mini-batch. This has the effect of stabilizing the learning process and dramatically reducing the number of training epochs required to train deep networks.

Goals of Batch Normalization

- Batch normalization is a technique used to improve the performance of a deep learning network by first removing the batch mean and then splitting it by the batch standard deviation.

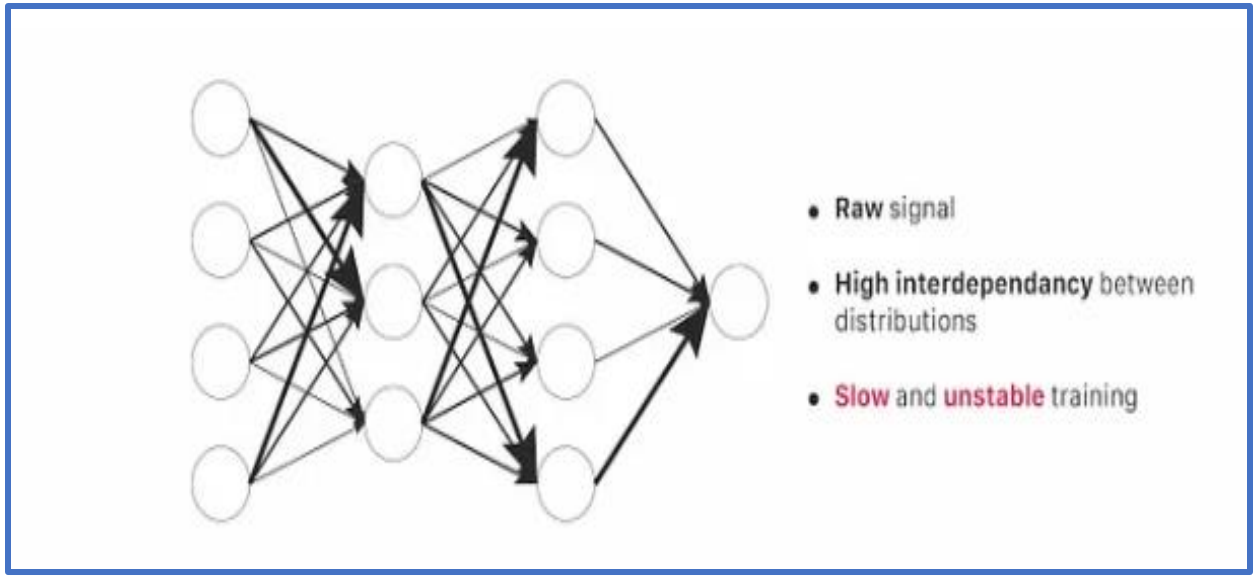
- The goal of batch normalization is to stabilize the training process and improve the generalization ability of the model. It can also help to reduce the need for careful initialization of the model's weights and can allow the use of higher learning rates, which can speed up the training process.

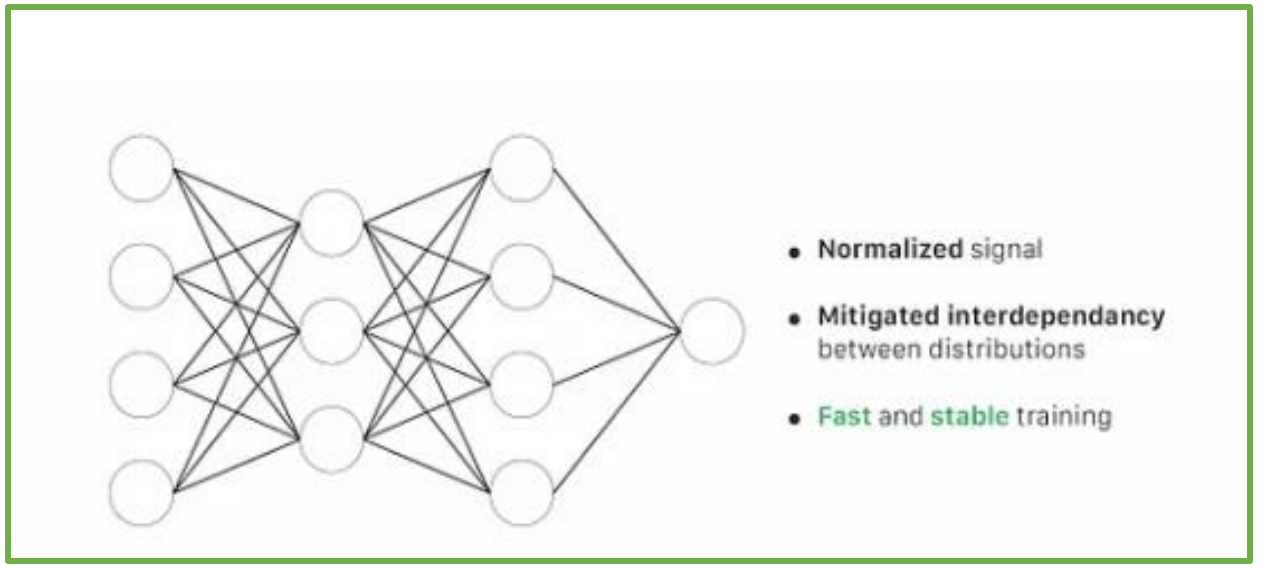
- It is common practice to apply batch normalization prior to a layer's activation function, and it is commonly used in tandem with other regularization methods like a dropout. It is a widely used technique in modern deep learning and has been shown to be effective in a variety of tasks, including image classification, natural language processing, and machine translation.

Batch Normalization

- Batch-Normalization (BN) is an algorithmic method which makes the training of Neural Networks faster and more stable.

- It consists of normalizing activation vectors from hidden layers using the first and the second statistical moments (mean and variance) of the current batch. This normalization step is applied right before (or right after) the nonlinear function.





Advantages of batch normalization

Stabilize the training process. Batch normalization can help to reduce the internal covariate shift that occurs during training, which can improve the stability of the training process and make it easier to optimize the model.

Improves generalization. By normalizing the activations of a layer, batch normalization can help to reduce overfitting and improve the generalization ability of the model.

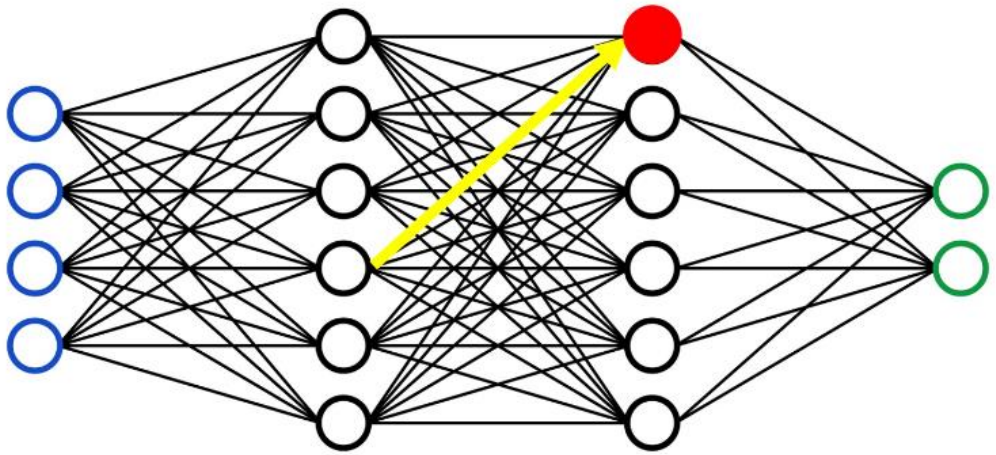
Reduces the need for careful initialization. Batch normalization can help reduce the sensitivity of the model to the initial weights, making it easier to train the model.

Allows for higher learning rates. Batch normalization can allow the use of higher learning rates that can speed up the training process.

When to Use Batch Normalization?

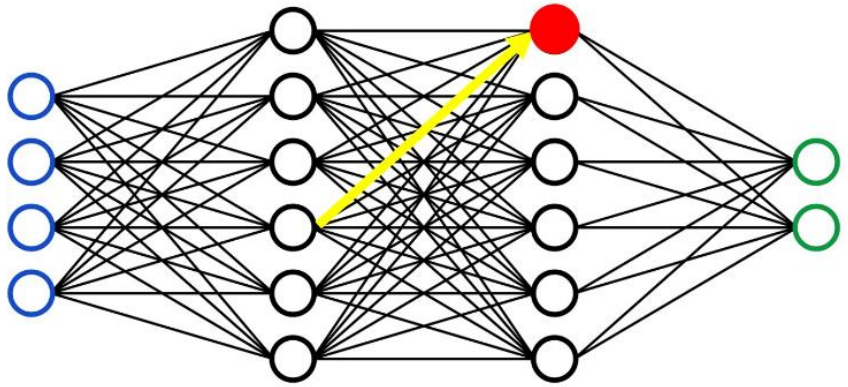
- We can use Batch Normalization in Convolution Neural Networks, Recurrent Neural Networks, and Artificial Neural Networks.

- In practical coding, we add Batch Normalization before the activation function of the output layer or before the activation function of the input layer.



How does Batch Normalization Work?

- When we normalize a dataset and start the training process, the weights in our model become updated over each epoch. So, what will happen if, during training, one of the weights ends up becoming drastically larger than the other weight? This large weight can again cause the output from its corresponding neuron to be extremely large. Then, this imbalance will again continue to cascade through the neural network causing the problem where features with larger values will have a bigger impact on the learning process compared with the features with smaller values.

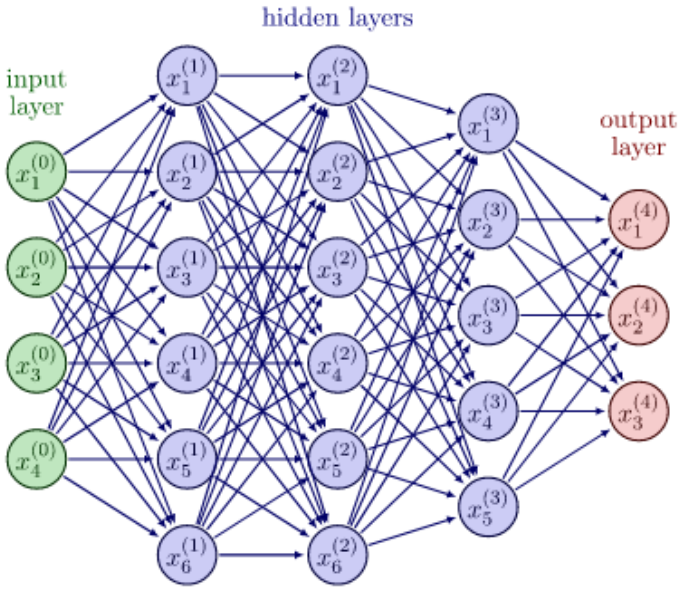


- That is the reason why we need to normalize not just the input data, but also the data in the individual layers of the network. When applying batch norm to a layer we first normalize the output from the activation function. After normalizing the output from the activation function, batch normalization adds two parameters to each layer. The normalized output is multiplied by a "standard deviation" parameter, and then a "mean" parameter is added to the resulting product.

- To sum up, when we apply standard normalization, the mean and standard deviation values are calculated with respect to the entire dataset. On the other hand, with batch normalization, the mean and standard deviation values are calculated with respect to the batch. This addition of batch normalization can significantly increase the speed and accuracy of our model.

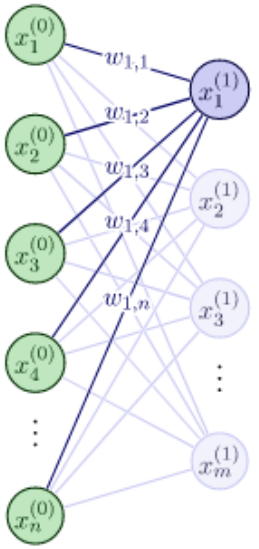
Batch normalization
Input
Normalize the outputs
Normalize the outputs
Normalize the outputs
Output

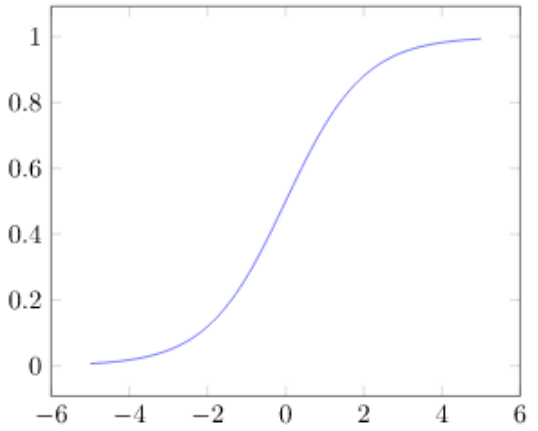
The Universal Approximation Theorem

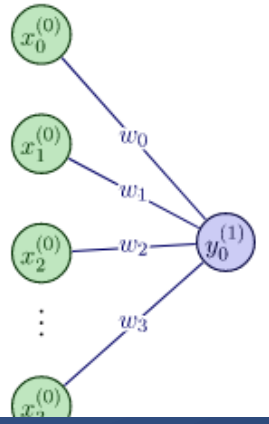


$$W^{(l)} = \begin{pmatrix} w^{(l)}_{1,1} & w^{(l)}_{1,2} & \dots & w^{(l)}_{1,n} \\ w^{(l)}_{2,1} & w^{(l)}_{2,2} & \dots & w^{(l)}_{2,n} \\ \dots & \dots & \ddots & \\ w^{(l)}_{m,1} & w^{(l)}_{2} & \dots & w^{(l)}_{m,n} \end{pmatrix}$$

$$\begin{aligned} \psi(x) &:= \sigma\left(\sum_{i=1}^{n} x_i w_i - b\right) \\ &= \sigma(\underbrace{w^\top \bullet x - b}_{\in \mathbb{R}}), \end{aligned}$$







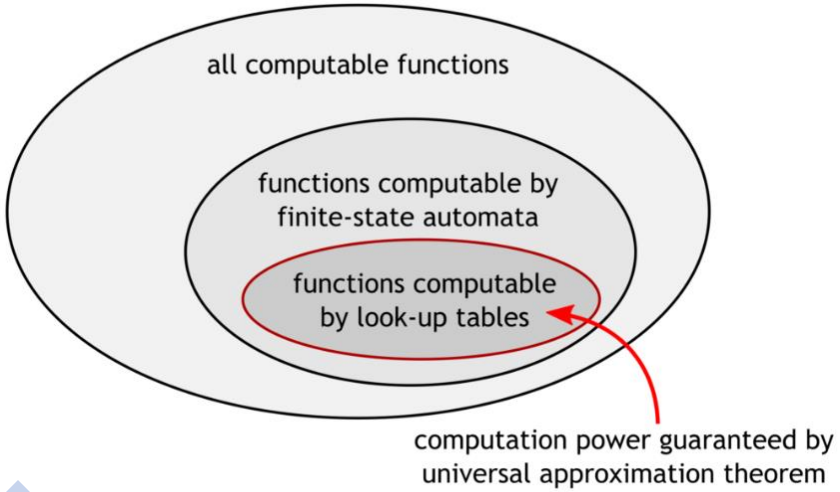
$$\int \sigma(w^\top x - b)\, d\mu(x) = 0 \quad \forall w \in \mathbb{R}^n, b \in \mathbb{R}$$

Universal Approximation Theorem

- Any neural network architecture aims at finding any mathematical function $y = f(x)$ that can map attributes(x) to output(y). The accuracy of this function i.e. mapping differs depending on the distribution of the dataset and the architecture of the network employed.
- The function $f(x)$ can be arbitrarily complex. The Universal Approximation Theorem tells us that Neural Networks has a kind of ***universality*** i.e. no matter what $f(x)$ is, there is a network that can approximately approach the result and do the job! This result holds for any number of inputs and outputs.
- In simple words, *the universal approximation theorem says that neural networks can approximate any function*. Now, this is powerful. Because what this means is that any task that can be thought of as a function computation, can be performed/computed by the neural networks. And almost any task that neural network does today is a function computation – be it language translation, caption generation, speech to text, etc. If you ask, how many layers will be required for any function computation? The answer is only 1.

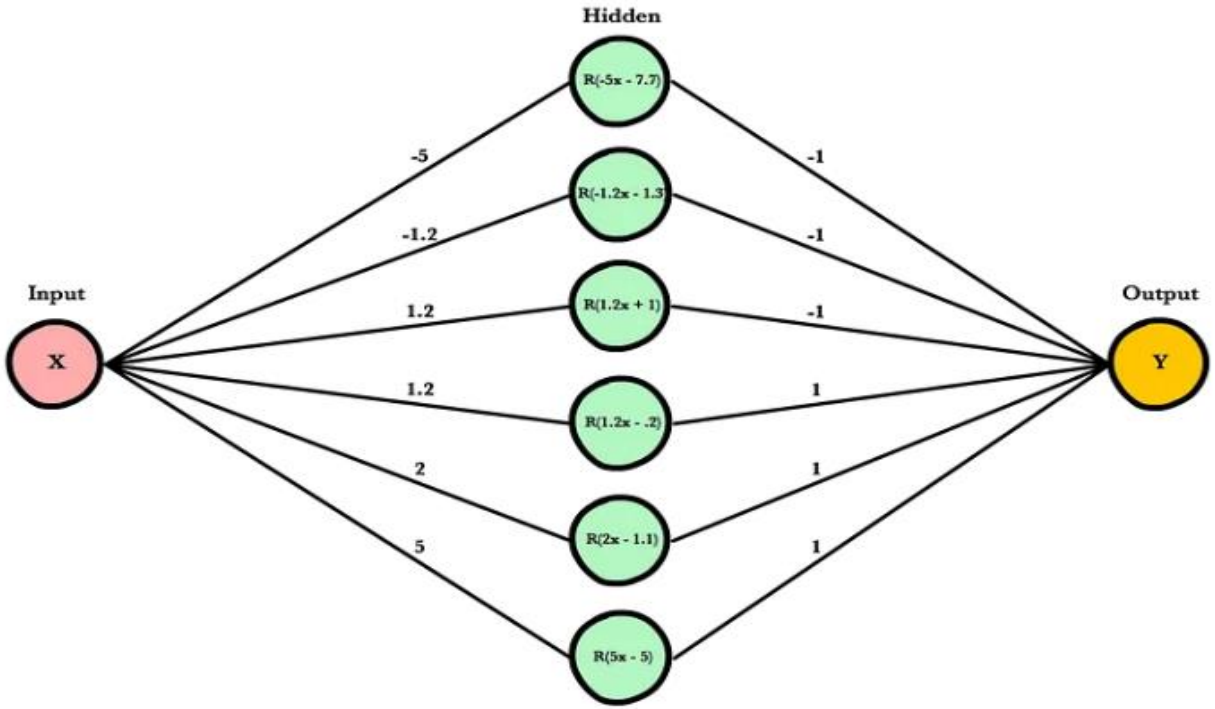
Wikipedia:

In the mathematical theory of artificial neural networks, the universal approximation theorem states that a feed-forward network with a single hidden layer containing a finite number of neurons can approximate continuous functions, under mild assumptions on the activation function.



Visual Understanding of Universal Approximation Theorem

- We have following neural network with one hidden layer:



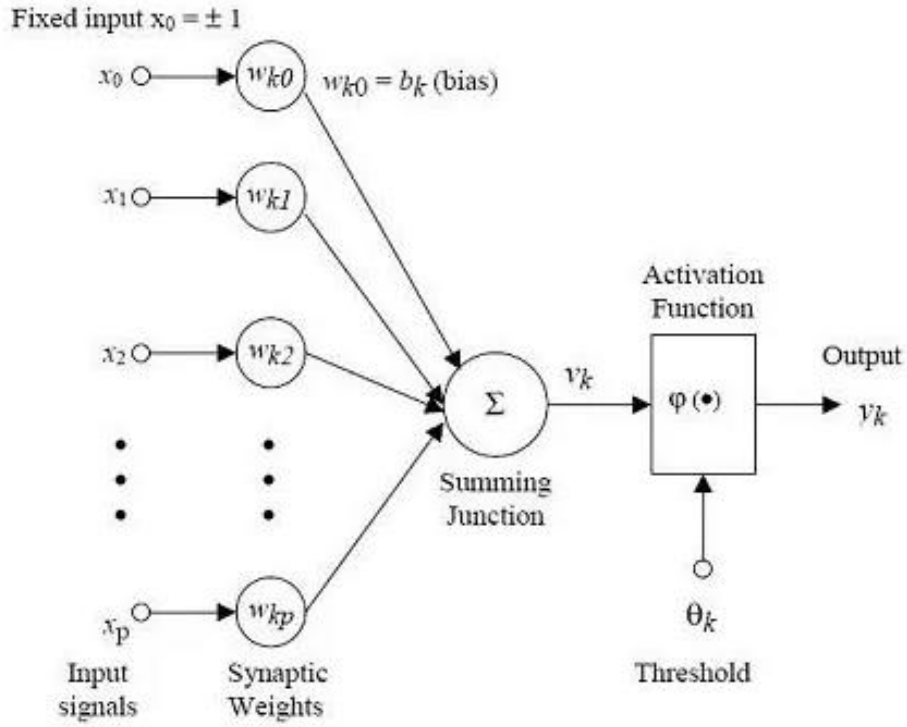
Note: A feedforward network with a single layer is sufficient to represent any function, but the layer may be infeasibly large and may fail to learn and generalize correctly.

Visual Understanding of Universal Approximation Theorem

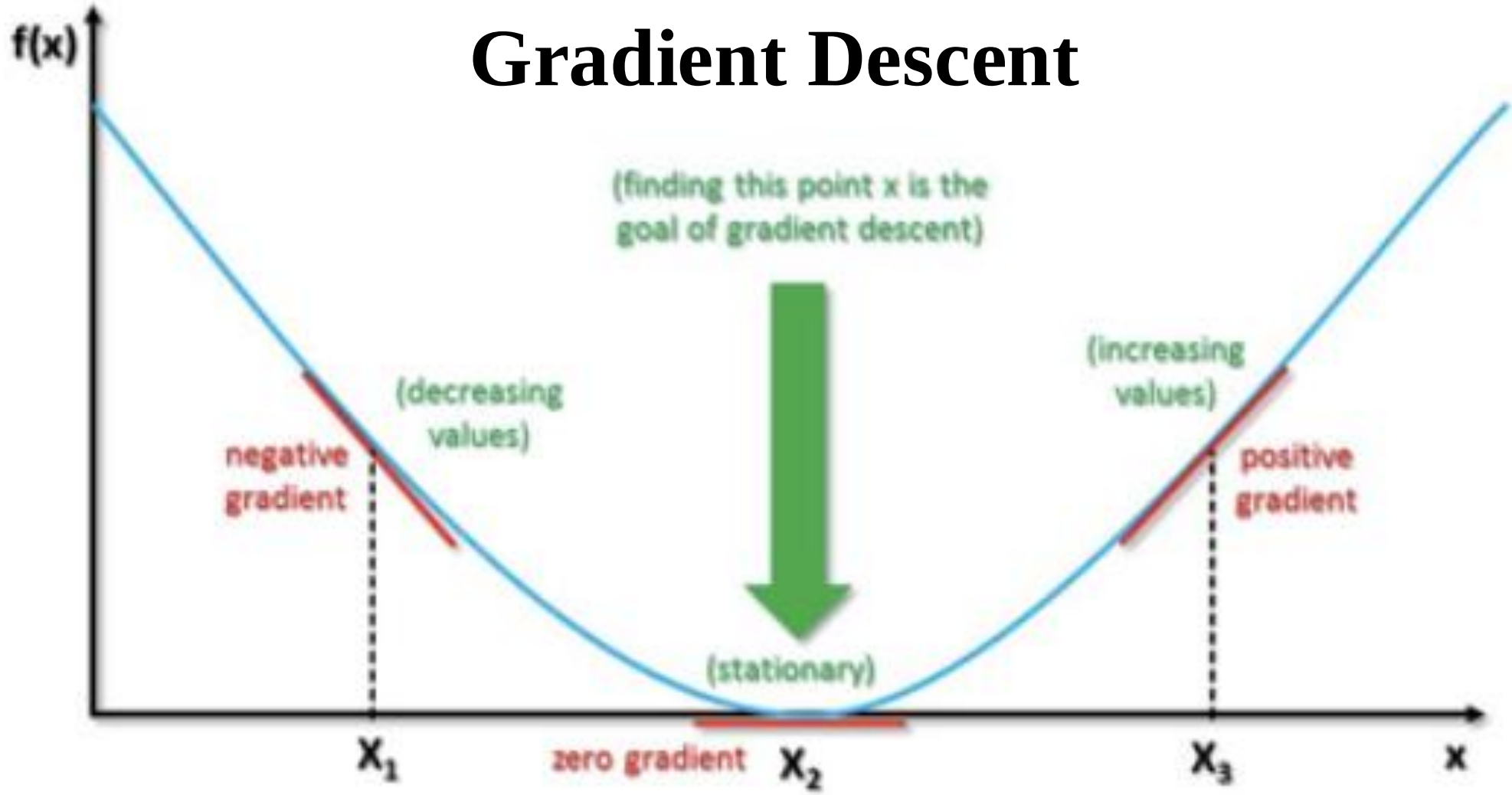
- A simple neural network including only a single hidden layer can approximate any continuous function. By adjusting weights and biases a neural network can take any input for *x* and approximate it to fit.

- By including *n* number of neurons in a single hidden layer, a neural network can approximate an input of *x* for any function, *f(x)*. This function must be continuous, if the function is not continuous a single hidden layer neural network does not easily or accurately approximate *f(x)*. (We can easily fix this by adding another layer to mimic the output of a discontinuous function.)

- In other words, a single hidden layer neural network can approximate any continuous function of x to any degree of precision



Gradient Descent

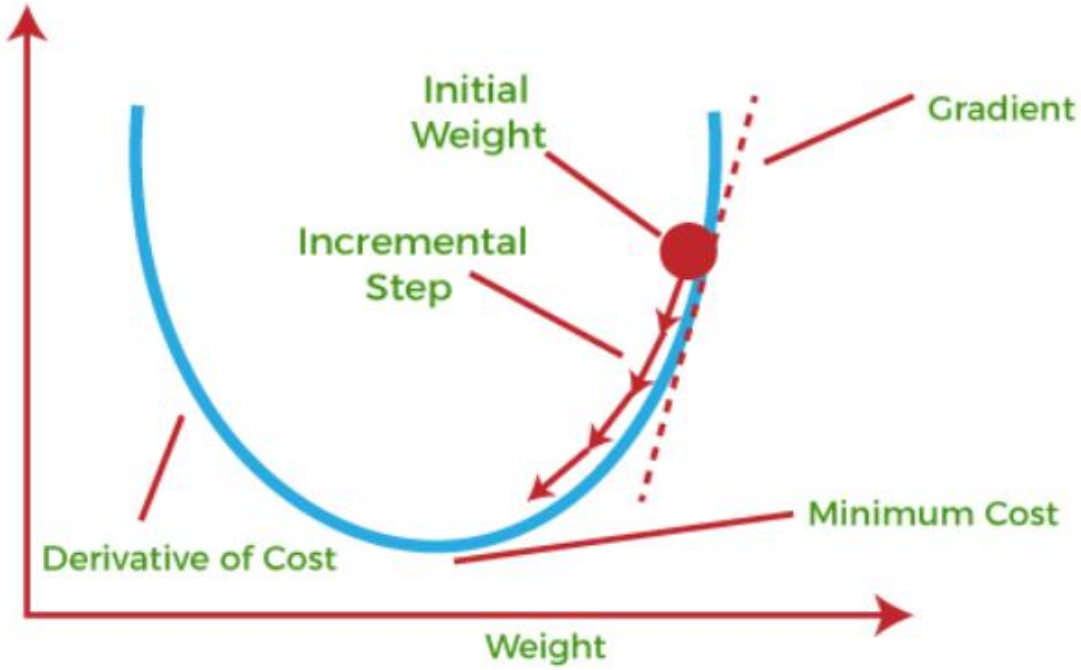


Gradient Descent

Gradient Descent is defined as one of the most commonly used iterative optimization algorithms to train the deep learning models. It helps in finding the local minimum of a function.

The best way to define the local minimum or local maximum of a function using gradient descent is as follows:

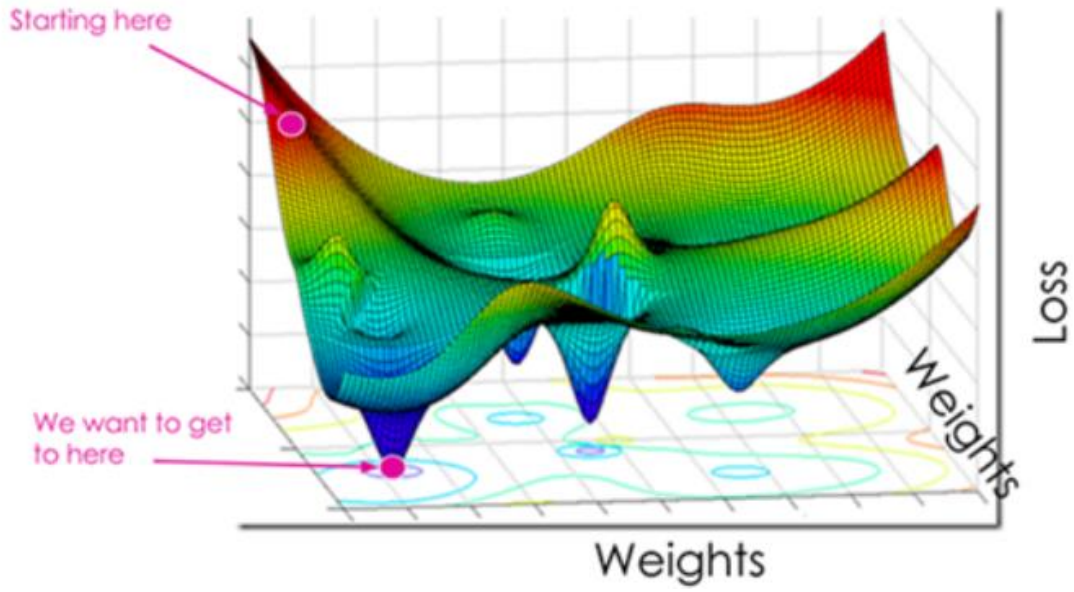
- If we move towards a negative gradient or away from the gradient of the function at the current point, it will give the local minimum of that function.
- Whenever we move towards a positive gradient or towards the gradient of the function at the current point, we will get the local maximum of that function.



Aim of Gradient Descent

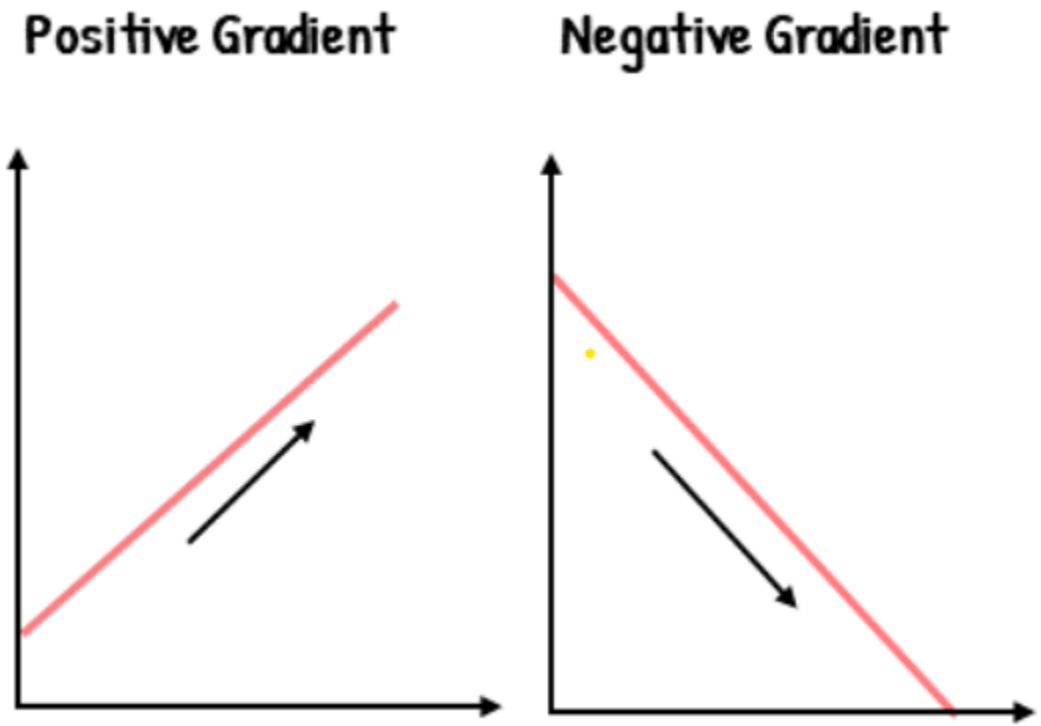
Gradient descent is an optimization algorithm that is used to find the weights that minimize the cost function. Minimizing the cost function means getting to the minimum point of the cost function. So, gradient descent aims to find a weight corresponding to the cost function's minimum point.

To find this weight, we must navigate down the cost function until we find its minimum point.



The Direction: The direction for navigating the cost function is found using the gradient.

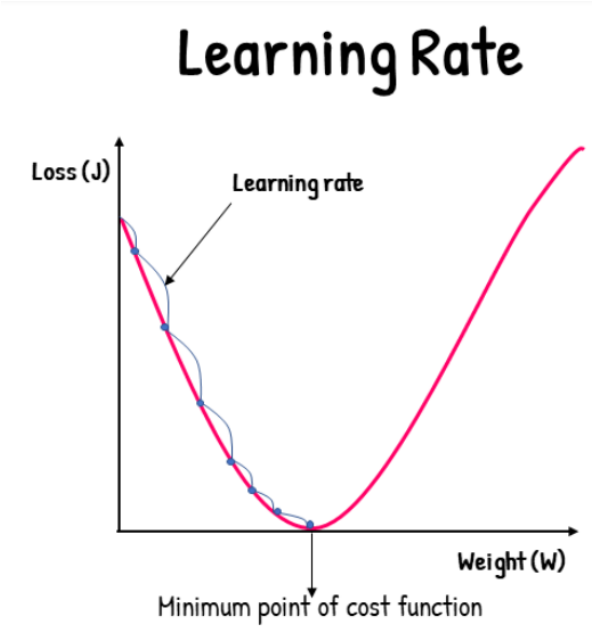
The Gradient: To know in which direction to navigate, gradient descent uses backpropagation. More specifically, it uses the gradients calculated through backpropagation. These gradients are used for determining the direction to navigate to find the minimum point. Specifically, we aim to find the negative gradient. This is because a negative gradient indicates a decreasing slope. A decreasing slope means that moving downward will lead us to the minimum point. For example:



The Step Size: The step size for navigating the cost function is determined using the learning rate.

Learning Rate : The learning rate is a tuning parameter that determines the step size at each iteration of gradient descent. It determines the speed at which we move down the slope. The step size plays an important part in ensuring a balance between optimization time and accuracy. The step size is measured by a parameter alpha (α). A small α means a small step size, and a large α means a large step size. If the step sizes are too large, we could miss the minimum point completely. This can yield inaccurate results. If the step size is too small, the optimization process could take too much time. This will lead to a waste of computational power.

The step size is evaluated and updated according to the behavior of the cost function. The higher the gradient of the cost function, the steeper the slope and the faster a model can learn (high learning rate). A high learning rate results in a higher step value, and a lower learning rate results in a lower step value. If the gradient of the cost function is zero, the model stops learning.

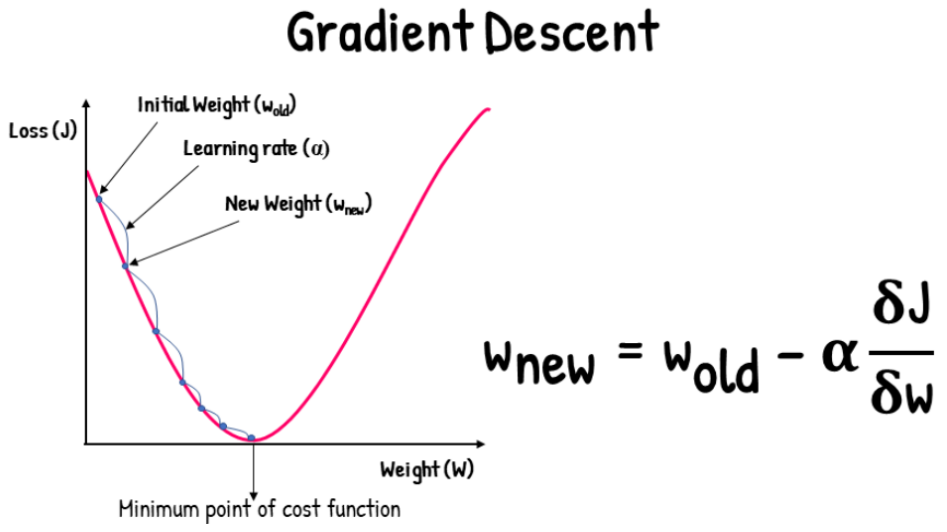


Descending the Cost Function

Navigating the cost function consists of adjusting the weights. The weights are adjusted using the following formula:

$$w_{new} = w_{old} - \alpha \underbrace{\frac{dJ}{dw}}_{\text{Gradient}}$$

This is the formula for gradient descent. As we can see, to obtain the new weight, we use the gradient, the learning rate, and an initial weight.
Adjusting the weights consists of multiple iterations. We take a new step down for each iteration and calculate a new weight. Using the initial weight and the gradient and learning rate, we can determine the subsequent weights.



From the graph of the cost function, we can see that:

1.To start descending the cost function, we first initialize a random weight.
2.Then, we take a step down and obtain a new weight using the gradient and learning rate. With the gradient, we can know which direction to navigate. We can know the step size for navigating the cost function using the learning rate.
3.We are then able to obtain a new weight using the gradient descent formula.
4.We repeat this process until we reach the minimum point of the cost function.
5.Once we've reached the minimum point, we find the weights that correspond to the minimum of the cost function

	Backpropagation	Gradient Descent
Definition	An algorithm for calculating the gradients of the cost function	Optimization algorithm used to find the weights that minimize the cost function
Requirements	Differentiation via the chain rule	•Gradient via Backpropagation •Learning rate
Process	Propagating the error backwards and calculating the gradient of the error function with respect to the weights	Descending down the cost function until the minimum point and find the corresponding weights

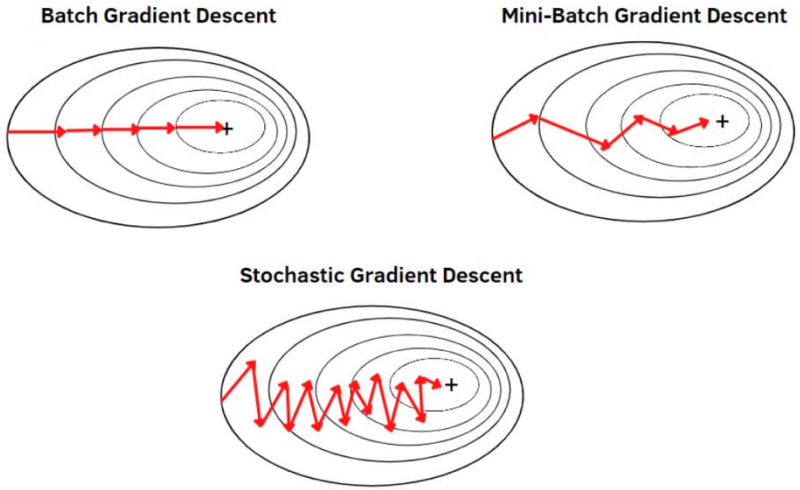
Types of Gradient Descent Algorithms

The cost of a neural network prediction is determined by evaluating the expected result and data samples in the training set. The delivery of data to neural networks for weight modification using backpropagation and gradient descent takes various forms.

This section presents three common methods of computing gradient descent based on the training data and data samples within training datasets.

- **Batch Gradient Descent (BGD)**
- **Stocastic Gradient Descent (SGD)**
- **Mini-Batch Gradient Descent**

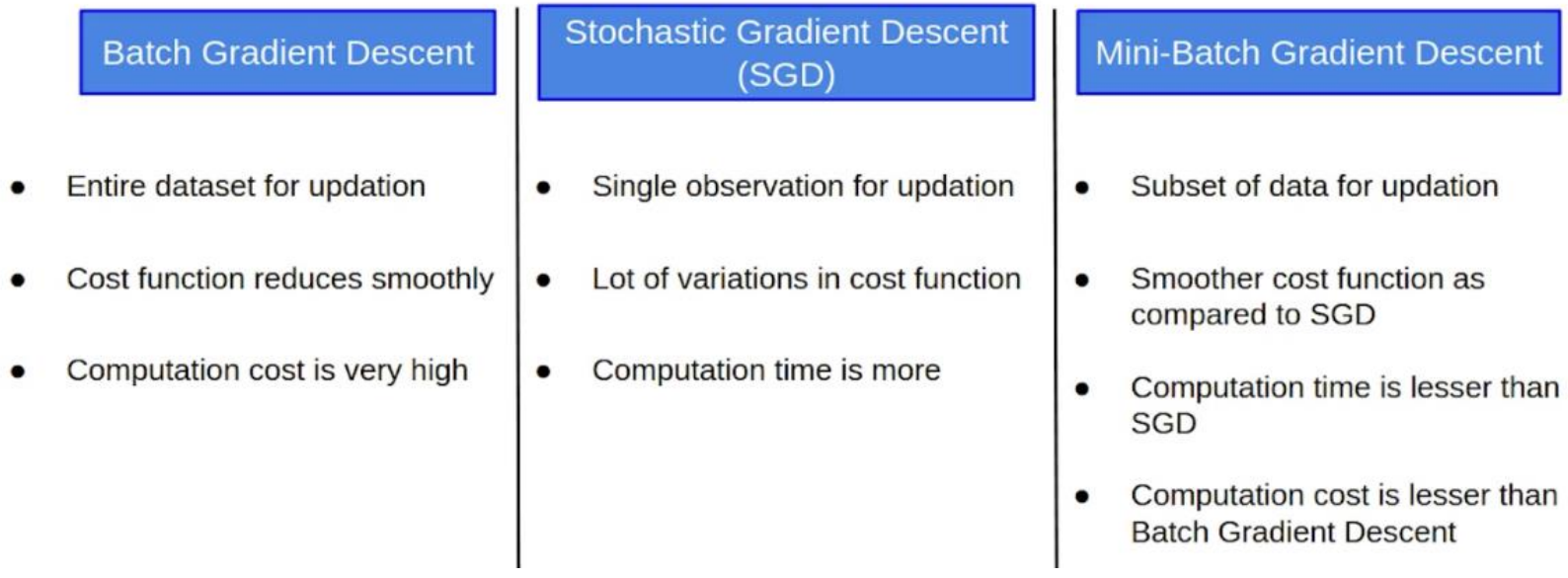
Variant	Advantages	Disadvantages
Batch gradient descent	Guaranteed convergence to global optimum	Computationally expensive for large datasets, slow convergence
Stochastic gradient descent	Faster convergence, more efficient for large datasets	High variance, may not converge to global optimum
Mini-batch gradient descent	Balanced convergence speed and computational cost, efficient for large datasets	Choice of mini-batch size can be a challenge



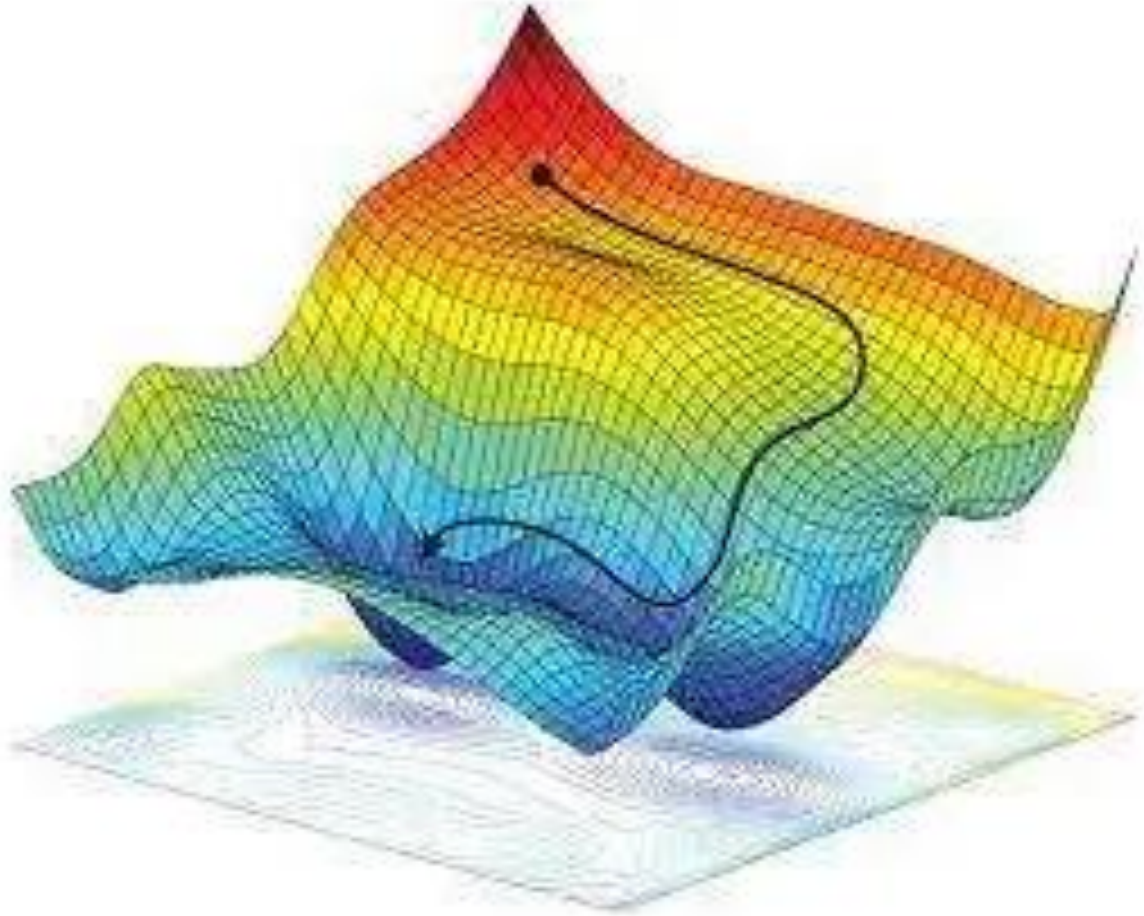
Batch gradient descent, also known as vanilla gradient descent, calculates the error for each example within the training dataset. Still, the model is not changed until every training sample has been assessed. The entire procedure is referred to as a cycle and a training epoch.

Stochastic gradient descent (SGD) changes the parameters for each training sample one at a time for each training example in the dataset. Depending on the issue, this can make SGD faster than batch gradient descent. One benefit is that the regular updates give us a fairly accurate idea of the rate of improvement.

Mini-batch gradient descent combines the ideas of batch gradient descent with SGD, it is the preferred technique. It divides the training dataset into manageable groups and updates each separately. This strikes a balance between batch gradient descent's effectiveness and stochastic gradient descent's durability.



Optimizers

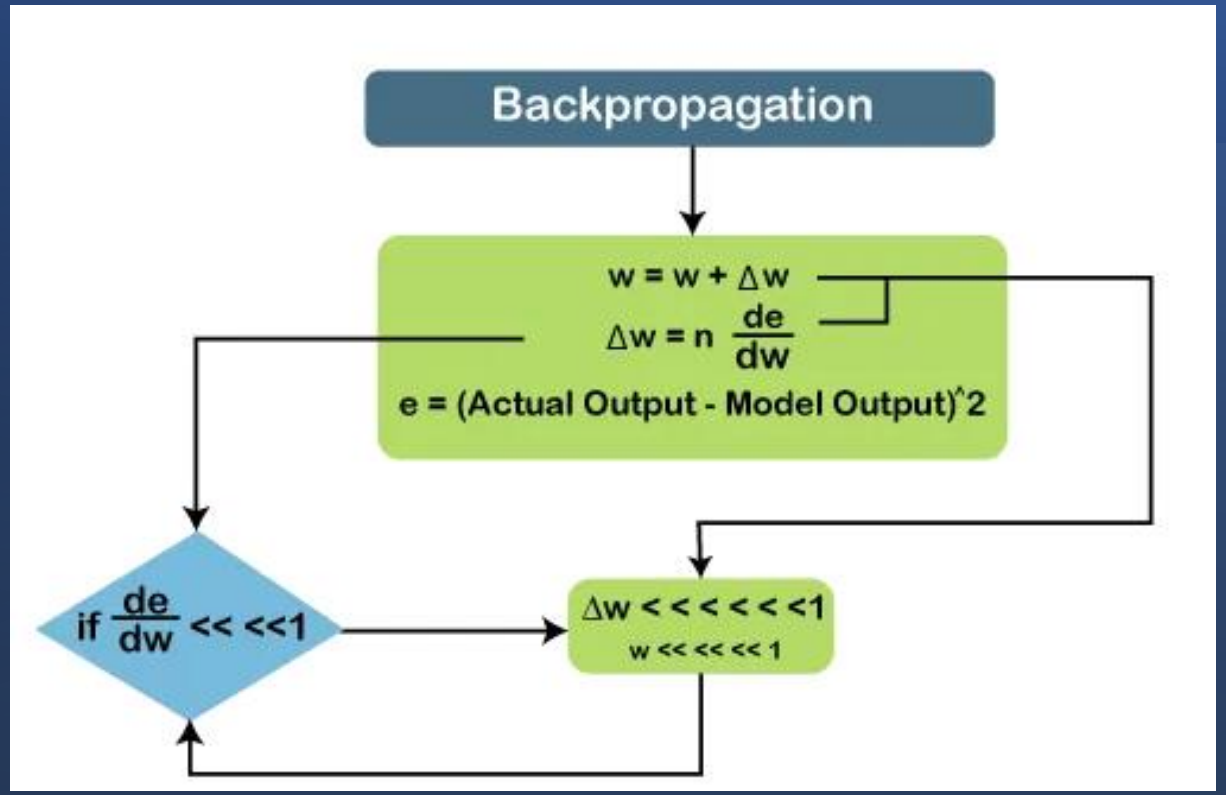


Optimizers

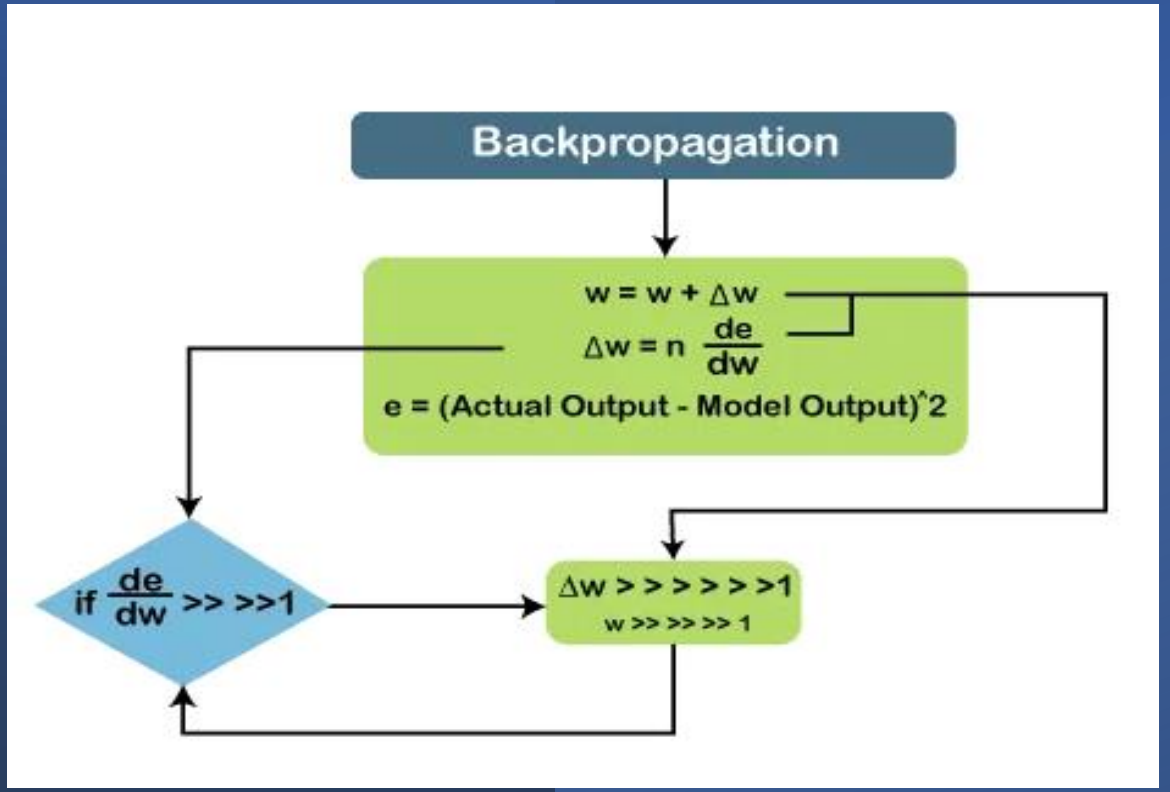
- It is very important to tweak the weights of the model during the training process, to make our predictions as correct and optimized as possible. But how exactly do you do that? How do you change the parameters of your model, by how much, and when?
- Best answer to all above question is *optimizers*. They tie together the loss function and model parameters by updating the model in response to the output of the loss function. In simpler terms, optimizers shape and mold your model into its most accurate possible form by futzing with the weights. The loss function is the guide to the terrain, telling the optimizer when it’s moving in the right or wrong direction.

- Choosing a good optimizer for your machine learning project can be overwhelming. Popular deep learning libraries such as PyTorch or TensorFLow offer a broad selection of different optimizers — each with its own strengths and weaknesses. However, picking the wrong optimizer can have a substantial negative impact on the performance of your deep learning model. This makes optimizers a critical design choice in the process of building, testing, and deploying your machine learning model.

Optimizer	State Memory [bytes]	# of Tunable Parameters	Strengths	Weaknesses
SGD	0	1	Often best generalization (after extensive training)	Prone to saddle points or local minima Sensitive to initialization and choice of the learning rate α
SGD with Momentum	$4n$	2	Accelerates in directions of steady descent Overcomes weaknesses of simple SGD	Sensitive to initialization of the learning rate α and momentum β
AdaGrad	$\sim 4n$	1	Works well on data with sparse features Automatically decays learning rate	Generalizes worse, converges to sharp minima Gradients may vanish due to aggressive scaling
RMSprop	$\sim 4n$	3	Works well on data with sparse features Built in Momentum	Generalizes worse, converges to sharp minima
Adam	$\sim 8n$	3	Works well on data with sparse features Good default settings Automatically decays learning rate α	Generalizes worse, converges to sharp minima Requires a lot of memory for the state
AdamW	$\sim 8n$	3	Improves on Adam in terms of generalization Broader basin of optimal hyperparameters	Requires a lot of memory for the state
LARS	$\sim 4n$	3	Works well on large batches (up to 32k) Counteracts vanishing and exploding gradients Built in Momentum	Computing norm of gradient for each layer can be inefficient



Vanishing Gradient



Exploding Gradients

Concepts : Vanishing and Exploring Gradients

What is a Gradient?

The Gradient is nothing but a derivative of loss function with respect to the weights. It is used to updates the weights to minimize the loss function during the back propagation in neural networks.

What is Vanishing Gradients?

Vanishing Gradient occurs when the derivative or slope will get smaller and smaller as we go backward with every layer during backpropagation. When weights update is very small or exponential small, the training time takes too much longer, and in the worst case, this may completely stop the neural network training. A vanishing Gradient problem occurs with the sigmoid and tanh activation function because the derivatives of the sigmoid and tanh activation functions are between 0 to 0.25 and 0–1. Therefore, the updated weight values are small, and the new weight values are very similar to the old weight values. This leads to Vanishing Gradient problem. We can avoid this problem using the ReLU activation function because the gradient is 0 for negatives and zero input, and 1 for positive input.

What is Exploding Gradients?

Exploding gradient occurs when the derivatives or slope will get larger and larger as we go backward with every layer during backpropagation. This situation is the exact opposite of the vanishing gradients. If the gradient becomes large, the model becomes unstable and model unable to learn from the training data. Exploding gradient problem can be identified if there is large changes in the loss from update to update. If the model loss goes to NaN during the training is an indication of exploding gradient problem.

This problem happens because of weights, not because of the activation function. Due to high weight values, the derivatives will also higher so that the new weight varies a lot to the older weight, and the gradient will never converge. So it may result in oscillating around minima and never come to a global minima point.

Solutions Vanishing Exploring Gradients

Vanishing Gradients Solution

Vanishing gradient problem is mainly occurring with sigmoid and tanh functions. We can avoid this problem by using the ReLU activation functions in training the deep neural networks.

Exploding Gradients Solution

Exploding Gradient Problem can be address by :

- By using the Long Short-Term Memory(LSTM) networks
- Using the Gradient Clipping (by setting threshold)
- Using the Weight Regularization
- Re-design the model with fewer layers

Please Refer to Codes Uploaded on Canvas


THANK YOU
Any Questions?