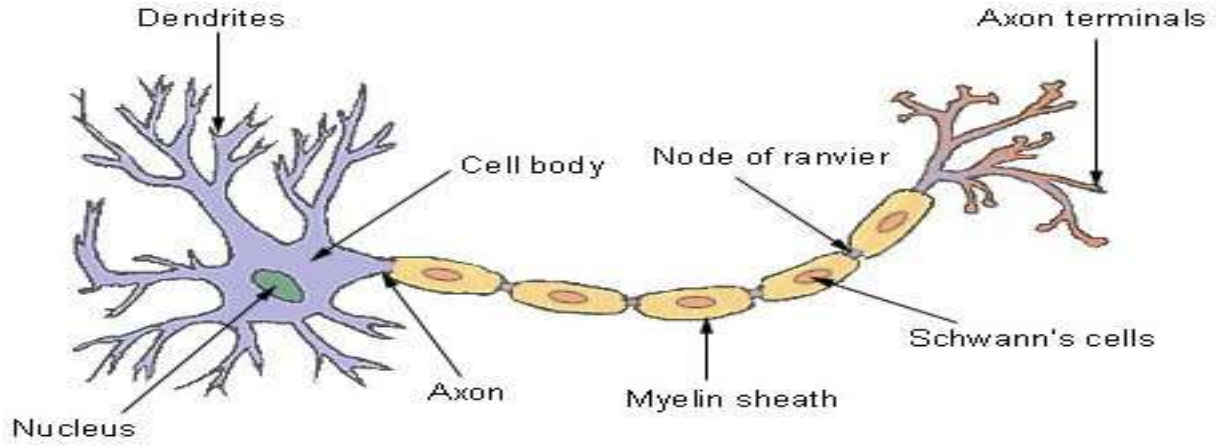
Artificial Neural Network

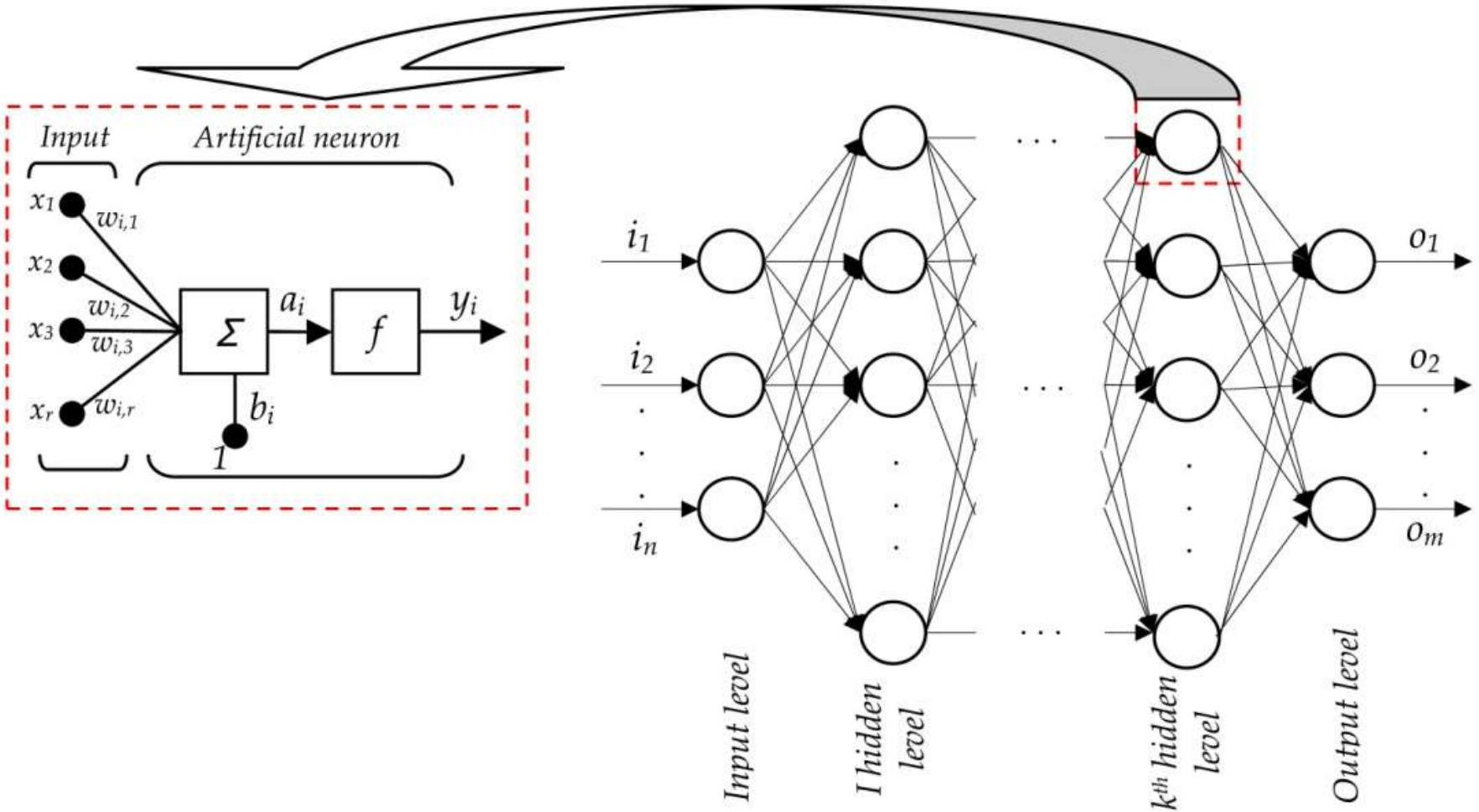
----Varsha Mali

What is a Neural Network?



- ANN models the relationship between set of input signals and output signals using a model derived from biological brain

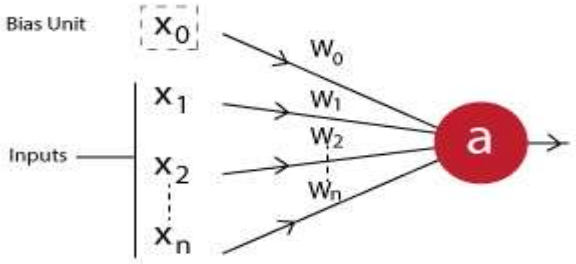
The general structure of a neural network looks like:



- **Input Layer:** The training observations are fed through these neurons
- **Hidden Layers:** These are the intermediate layers between input and output which help the Neural Network learn the complicated relationships involved in data.
- **Output Layer:** The final output is extracted from previous two layers. For Example: In case of a classification problem with 5 classes, the output later will have 5 neurons.

How a Single Neuron works?

Diagram 1: Single NN Working



The different components are:

x_1, x_2,…, x_N: Inputs to the neuron

x_0: Bias unit. It works similar to an intercept term and typically has +1 value.

w_0,w_1, w_2,…,w_N: Weights on each input. Note that even bias unit has a weight.

a: Output of the neuron which is calculate $a = f(\sum_{i=0}^{N} w_i x_i)$

Here **f** is known an **activation function**.

$$f(x) = \begin{cases} 0, & for\ x < 0 \\ 1, & for\ x \geq 0 \end{cases}$$

EXAMPLE 1: AND

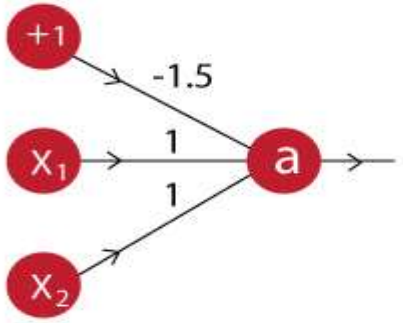
- The AND function can be implemented as:

- The output of this neuron is:

  a = f(-1.5 + x1 + x2)

- The truth table for this implementation is:

Diagram 2: AND



X1	X2	X1 AND X2	(-1.5+X1+X2)	a
0	0	0	-1.5	0
0	1	0	-0.5	0
1	0	0	-0.5	0
1	1	1	0.5	1

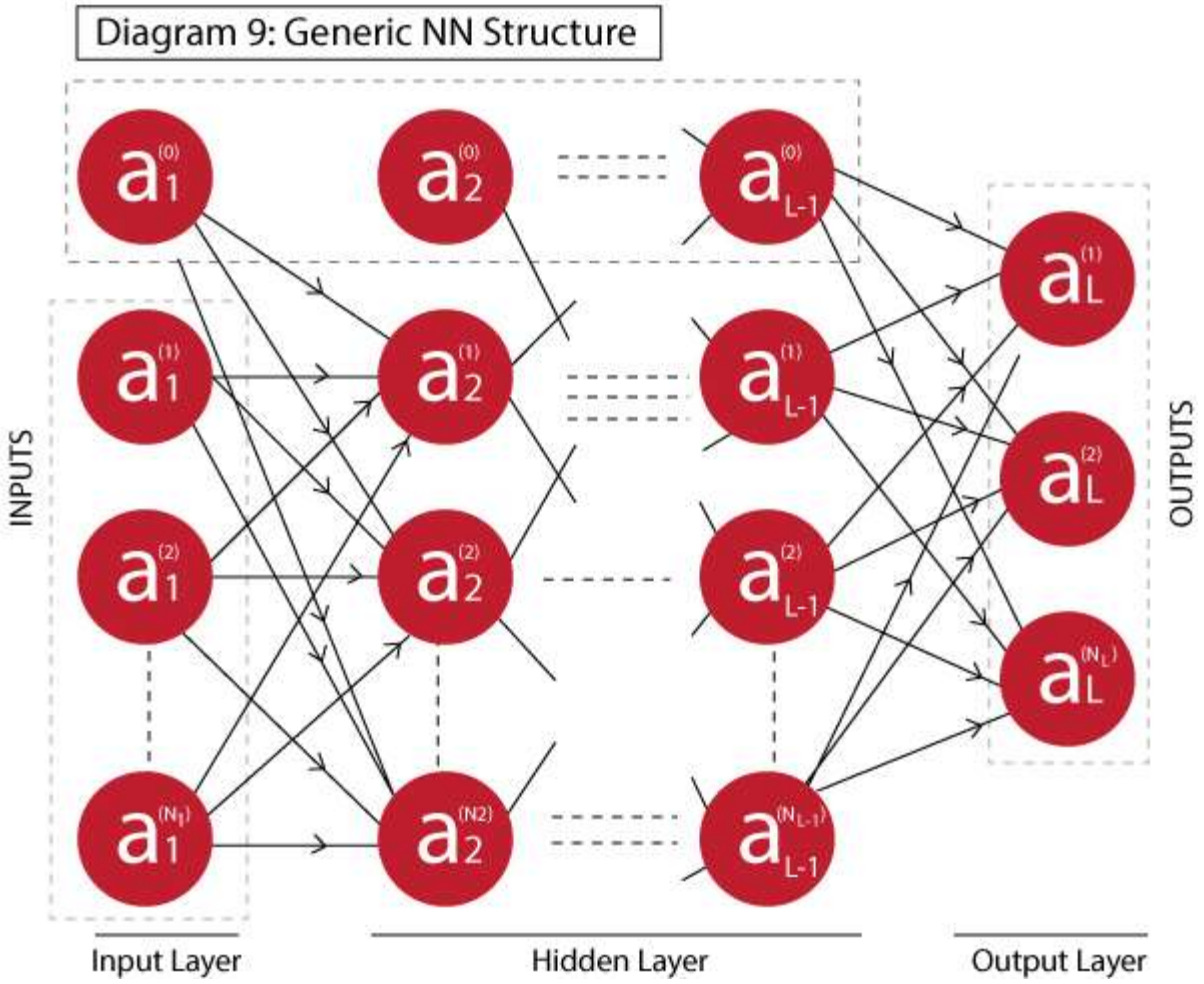
Why multi-layer networks are useful?

- The idea behind using multiple layers is that complex relations can be broken into simpler

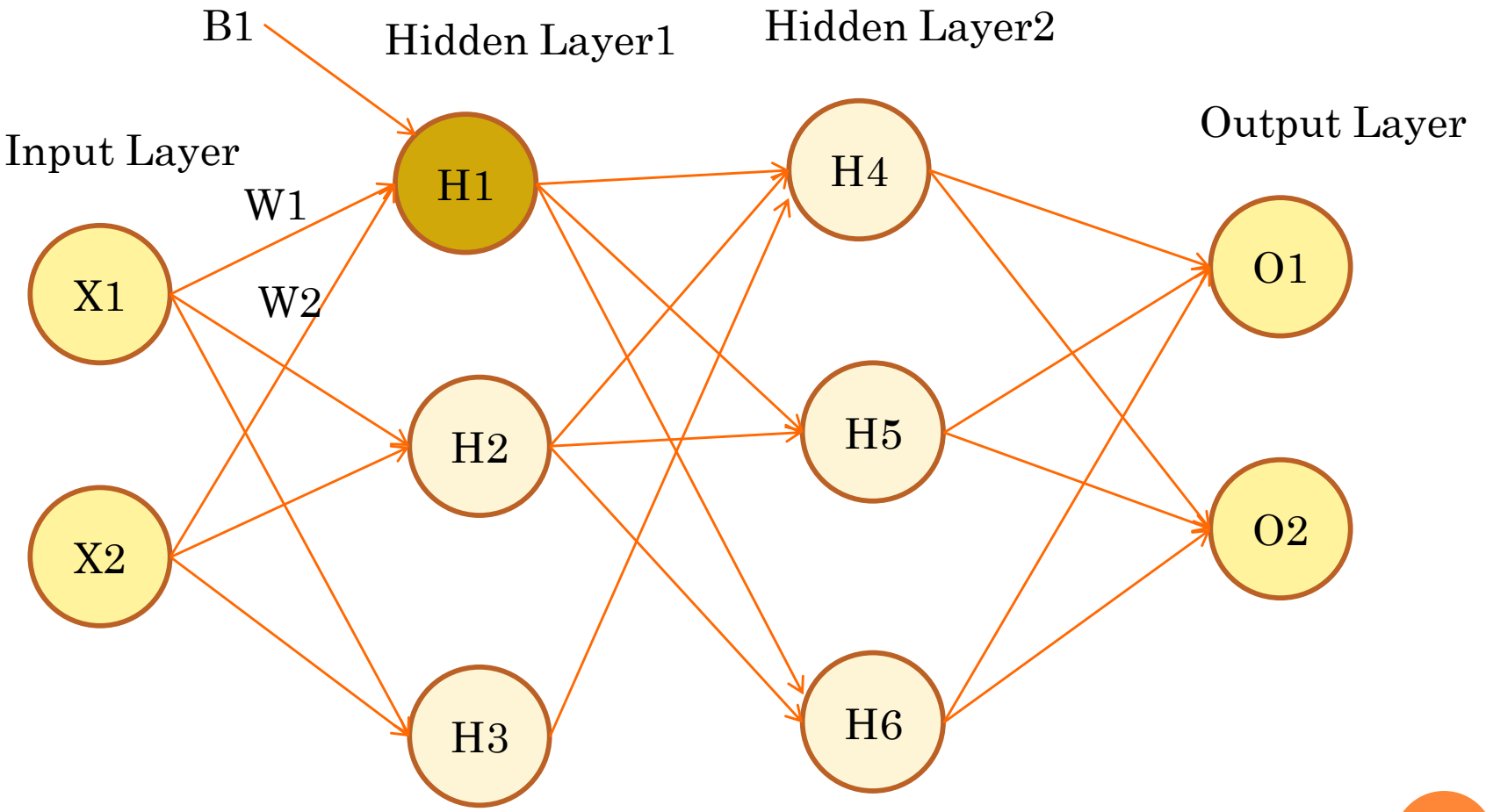
X1	X2	X1 XNOR X2
0	0	1
0	1	0
1	0	0
1	1	1

- The idea This sort of a relationship cannot be modeled using a single neuron.
- Thus we will use a multi-layer network.

General Structure of a Neural Network

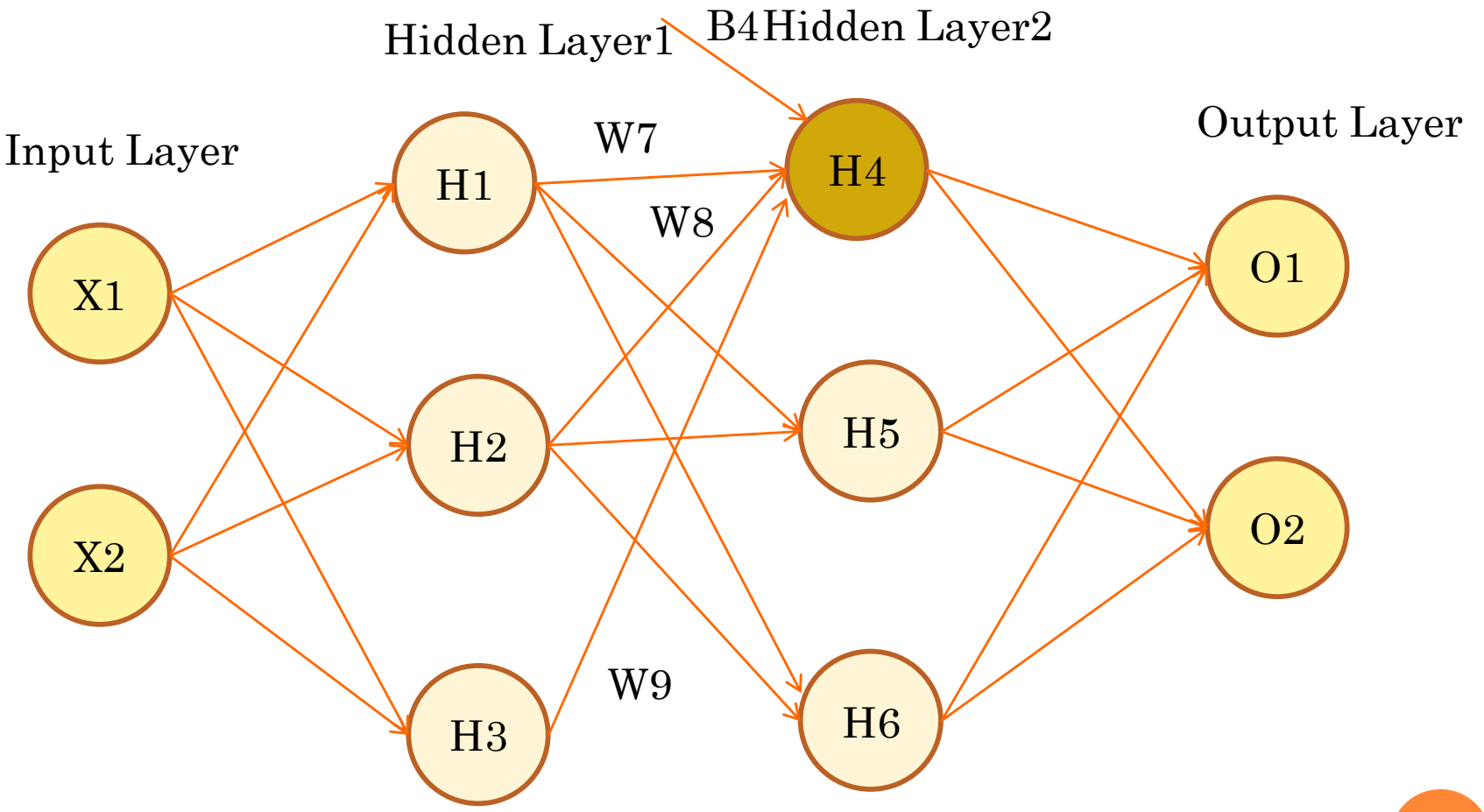


Forward Propagation of layer 1



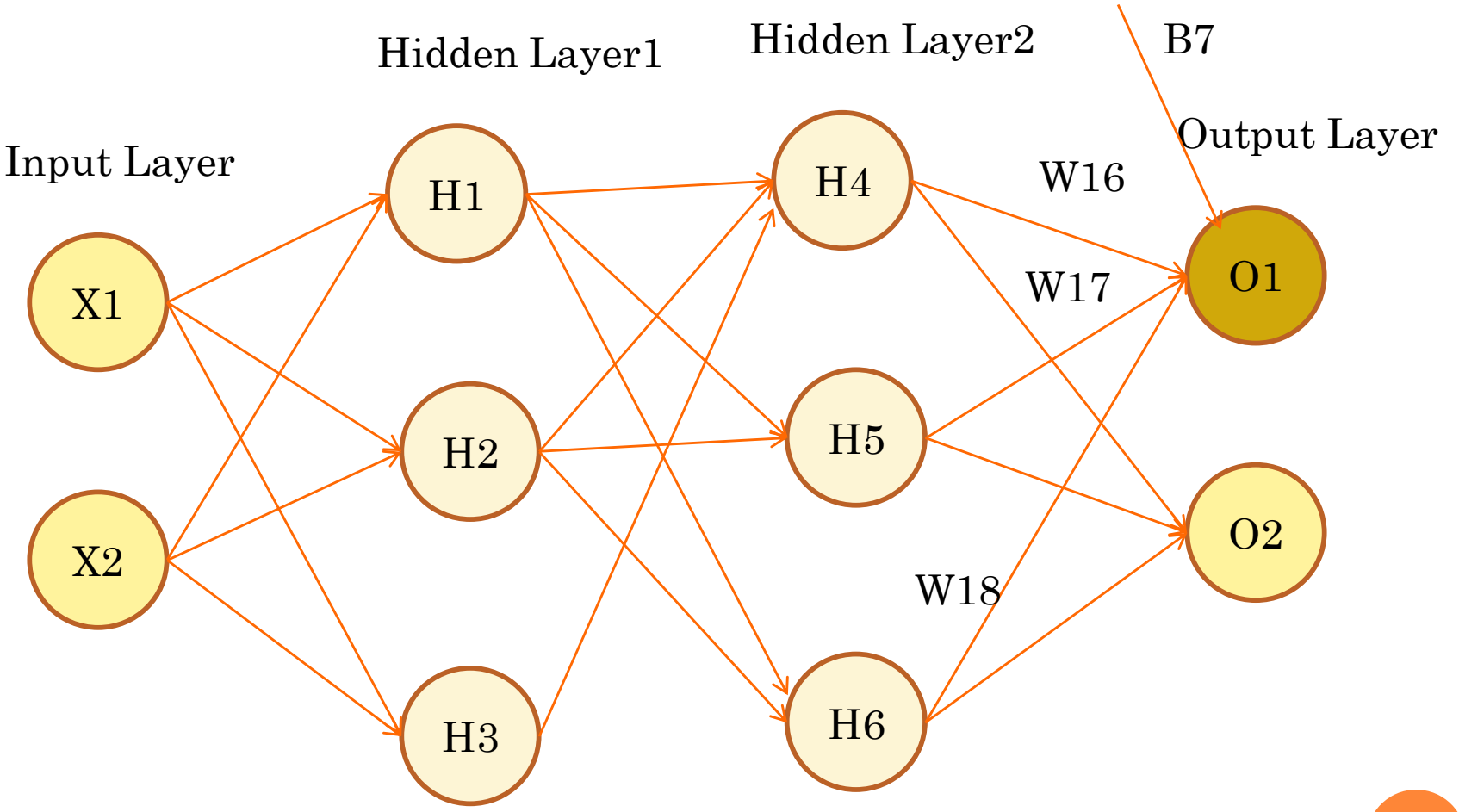
$$H1 = f(B1 + W1 * X1 + W2 * X2)$$

Forward Propagation of layer 2



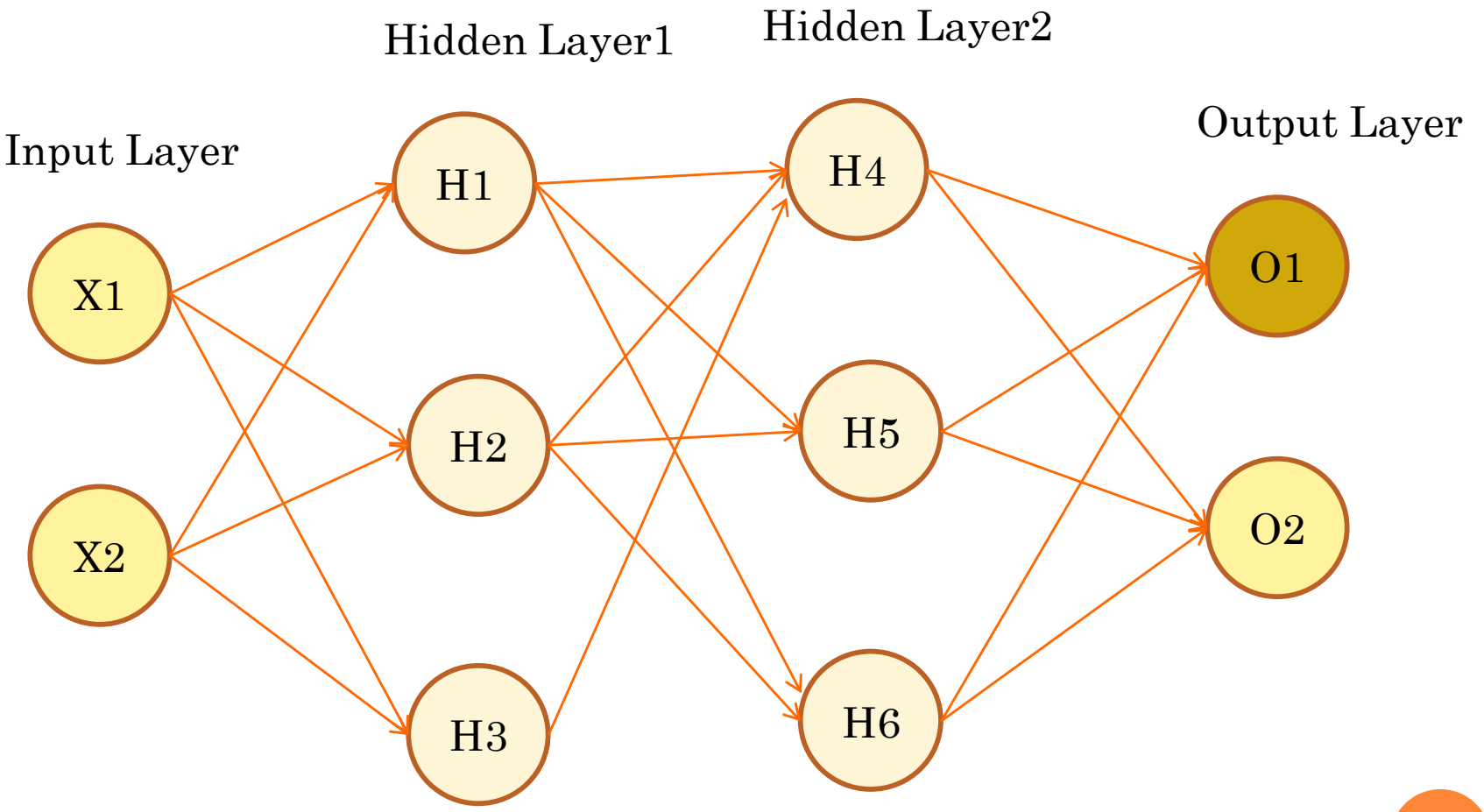
$$H4 = f(B4 + W7*H1 + W8*H2 + W9*H3)$$

Forward Propagation of Output Layer



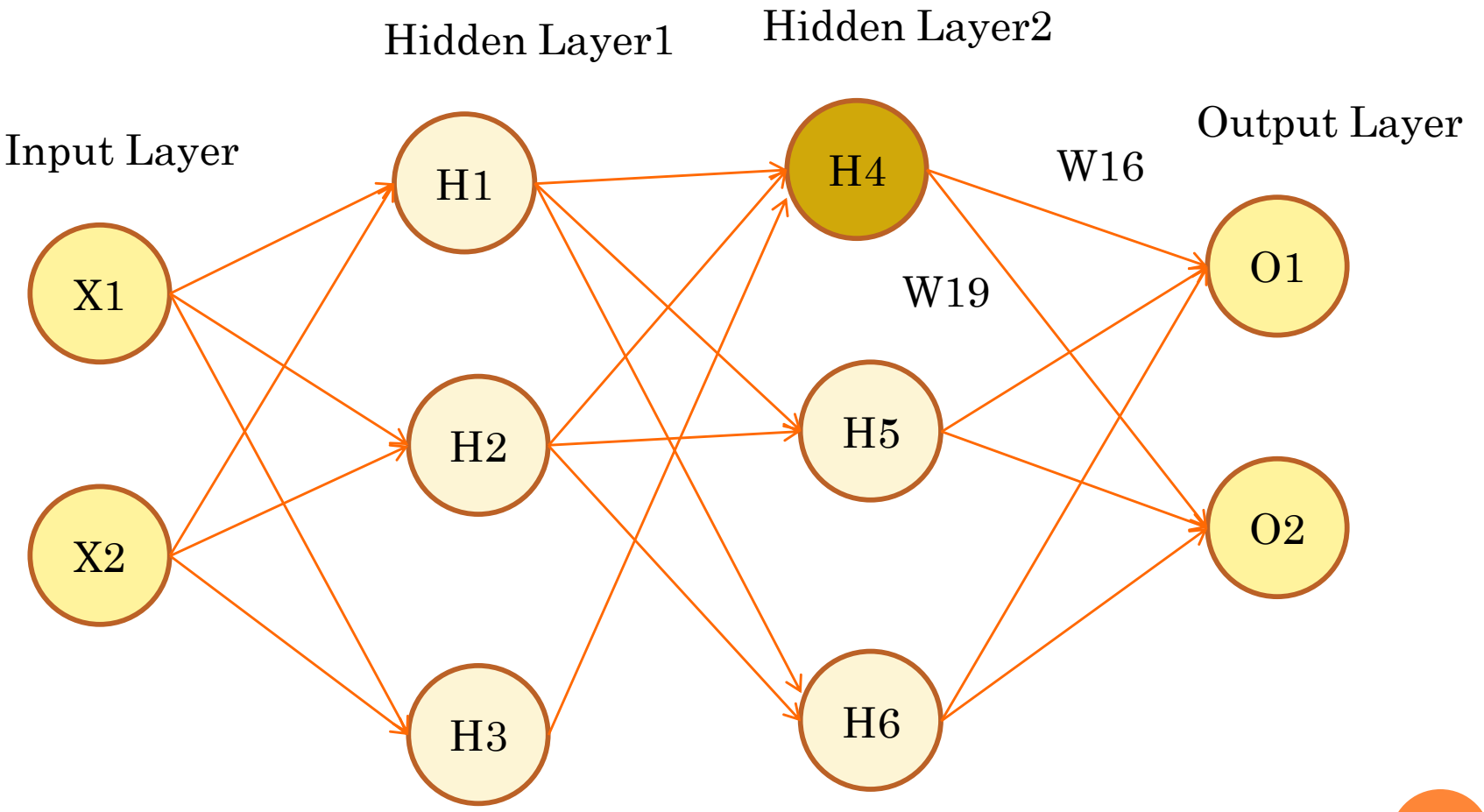
$$O1 = f(B7 + W1*H4 + W17*H5 + W18*H6)$$

BackPropagation of Output Layer



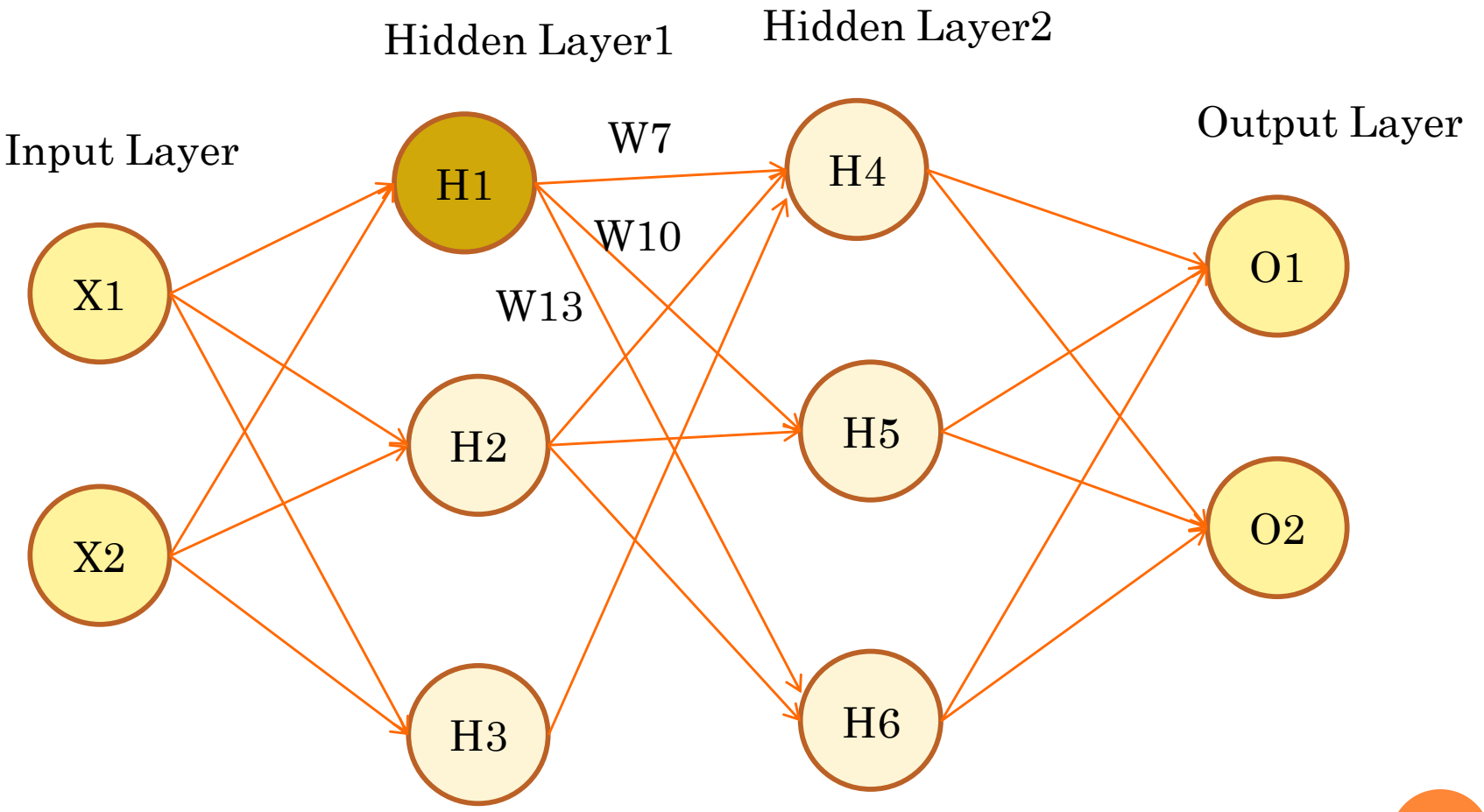
$$E(O1) = f'(O1) * (True1 - O1)$$

BackPropagation of Hidden Layer 2



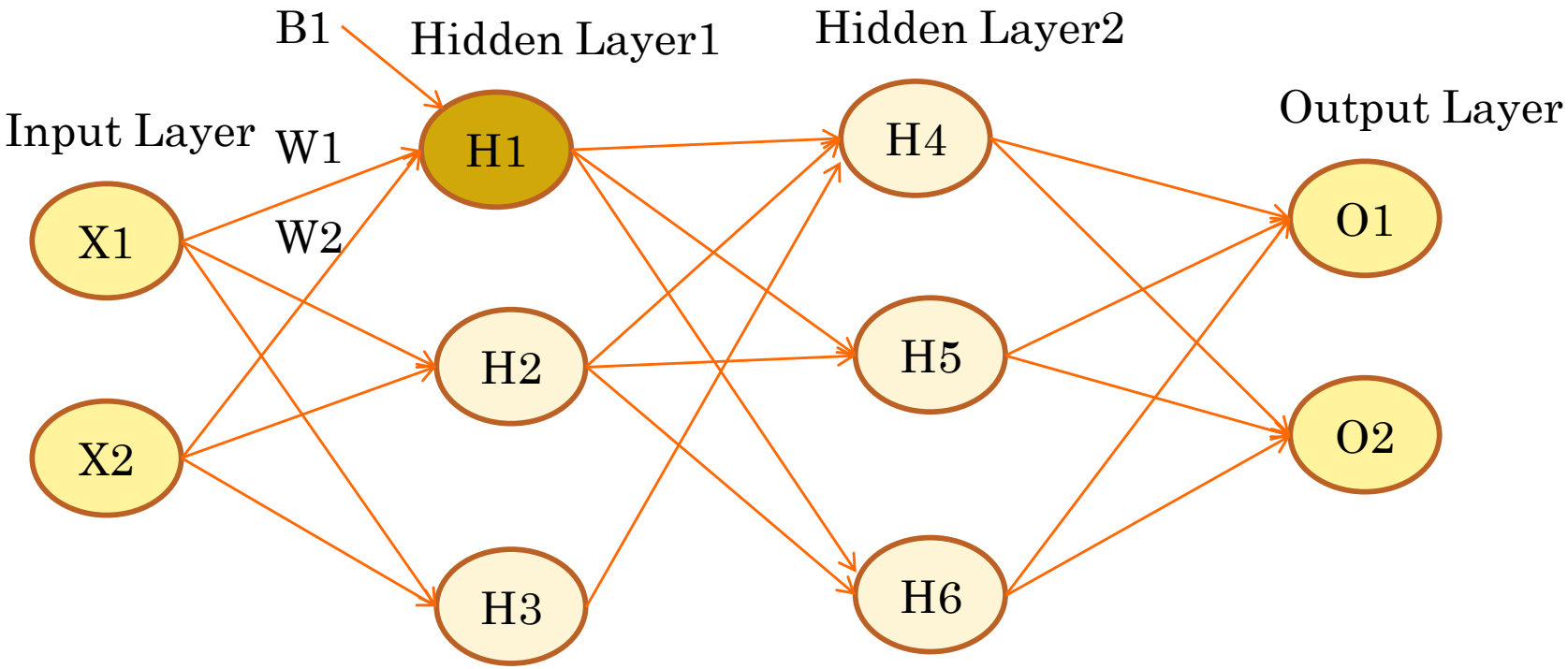
$$E(H4) = f'(H4) + (W16 * E(O1) + W19 * E(O2))$$

BackPropagation of Hidden Layer 1



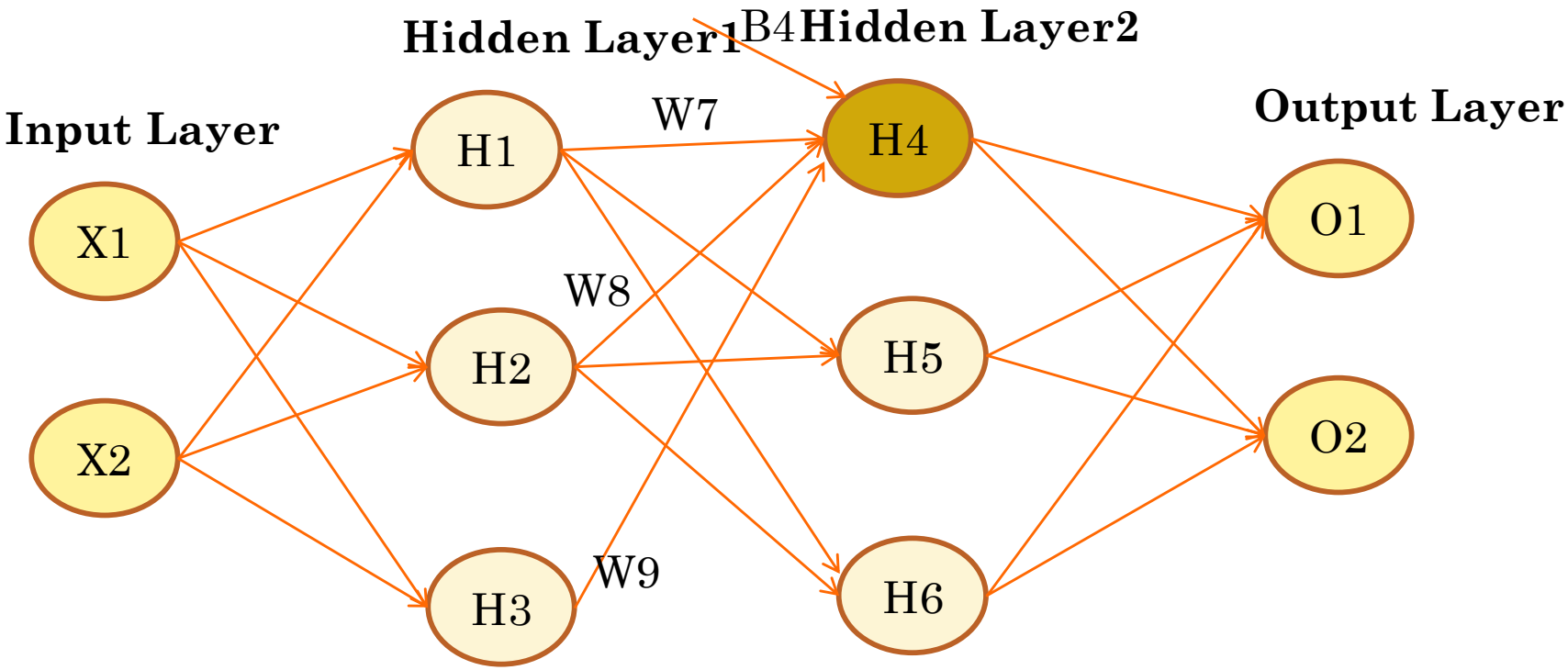
$$E(H1) = f'(H1) + (W7 * E(H4) + W10 * E(H5) + W13 * E(H6))$$

Updating Weights of Hidden layer 1



B1= B1+ a * E(H1) *1
W1= W1 + a * E(H1) *X1
W2= W2 + a * E(H1) *X2

Forward Propagation of layer 2



B4= B4+ a * E(H4) *1
W7= W7 + a * E(H4) *H1
W8= W8 + a * E(H4) *H2
W9= W9 + a * E(H4) *H3

Back-Propagation

- Back-propagation (BP) algorithms works by determining the loss (or error) at the output and then propagating it back into the network.
- The back-propagation algorithm can be used to train feed forward neural networks or multilayer perceptrons.
- It is a method to minimize the cost function by changing weights and biases in the network.
- To learn and make better predictions, a number of epochs (training cycles) are executed where the error determined by the cost function is backward propagated by gradient descent until a sufficiently small error is achieved.

Error Optimization

- let's sum up the entire process behind optimization of a neural network. The various steps involved in each iteration are:
  1. Select a **network architecture**, i.e. number of hidden layers, number of neurons in each layer and activation function
  2. **Initialize weights** randomly
  3. Use **forward propagation** to determine the output node
  4. Find the **error** of the model using the known labels

     $e_L^{(i)} = y^{(i)} - a_L^{(i)} \mid i = 1,2,....,N_L$

  Here $y^{(i)}$ is the actual outcome from training data

5. **Back-propogate** the error into the network and determine the error for each node

The error for layer L-1 should be determined first using the following:

where i = 0,1,2, ….., NL-1 (number of nodes in L-1th layer)

$$e_{L-1}^{(i)} = \left(\sum_{k=1}^{N_L} W_{ik}^{(L-1)} . e_L^{(i)} \right) * f'(x)^{(i)}$$

6. **Update** the **weights** to minimize gradient

Use the following update rule for weights:

$$W_{ik}^{(l)} = W_{ik}^{(l)} + a^{(i)}.e_{l+1}^{(k)}$$

where,

- $l = 1,2,\ldots..,(L-1)$ | index of layers (excluding the last layer)
- $i = 0,1,\ldots..,N_l$ | index of node in l^{th} layer
- $k = 1,2,\ldots..,N_{l+1}$ | index of node in $l+1^{th}$ layer
- $W_{ik}^{(l)}$ refers to the weight from the l^{th} layer to $l+1^{th}$ layer from i^{th} node to k^{th} node

Popular types of activation functions and when to use them

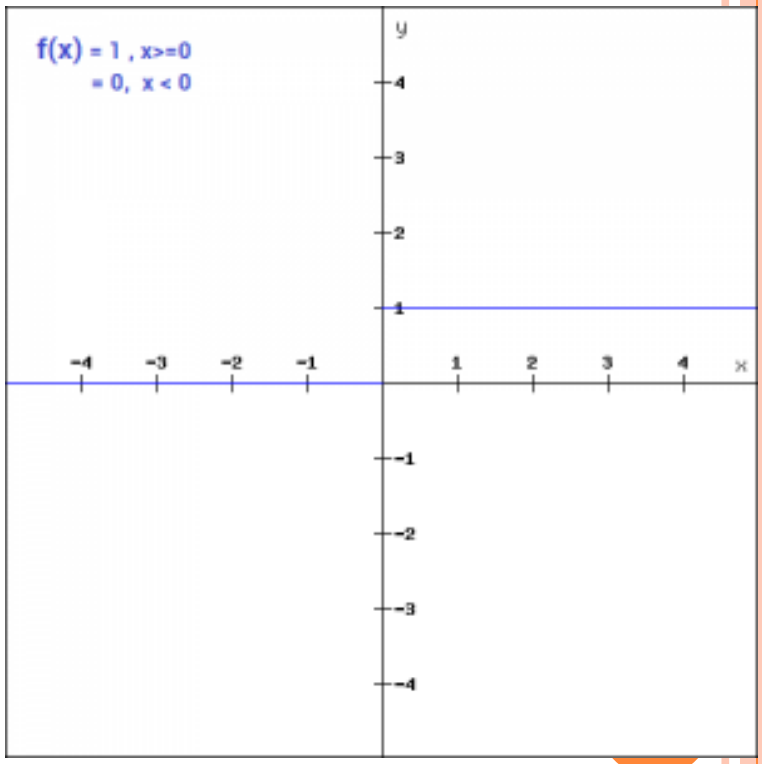
- **Binary Step Function**
  - f(x) = 1, x>=0

Pros:

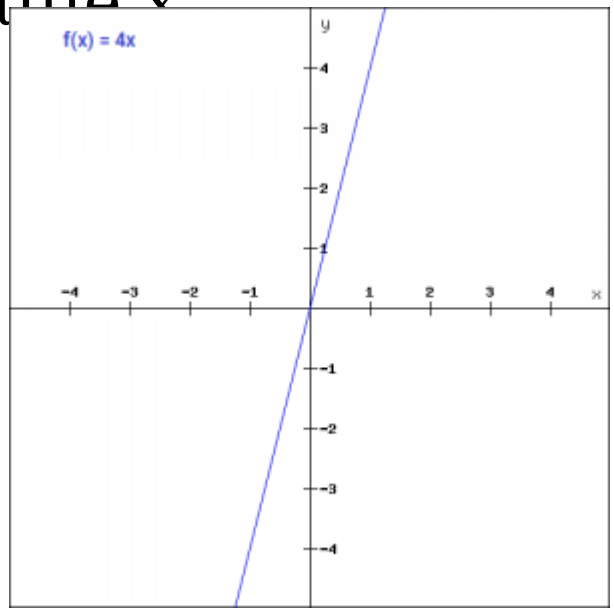
- Simple
- Best for binary classification

Cons:

- Not suitable for multiple classes
- Gradient is zero



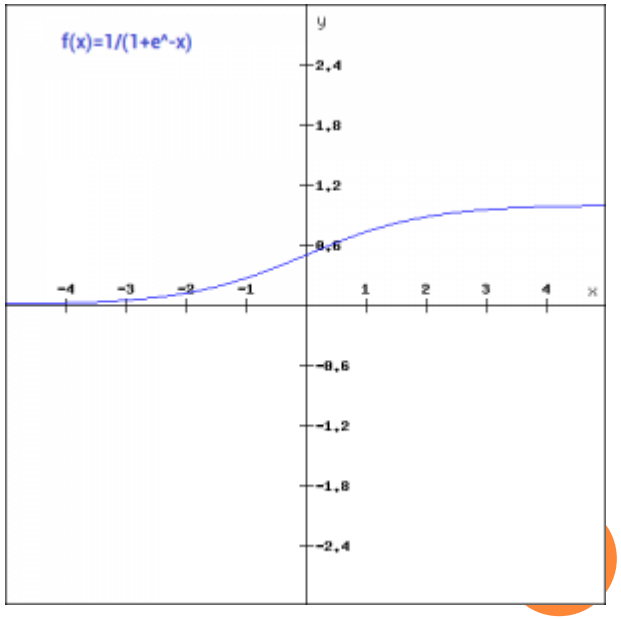
- **Linear Function**
  - f(x)=ax
  - The derivative of a linear function is constant i.e. it does not depend upon the input value x
  - f'(x) = a



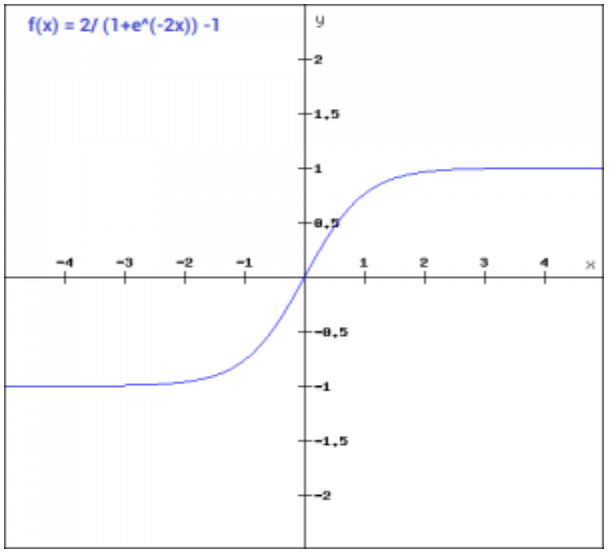
- **Sigmoid**
  - f(x)=1/(1+e^-x)
  - This is a smooth function and is continuously differentiable. The biggest advantage that it has over step and linear function is that it is non-linear.

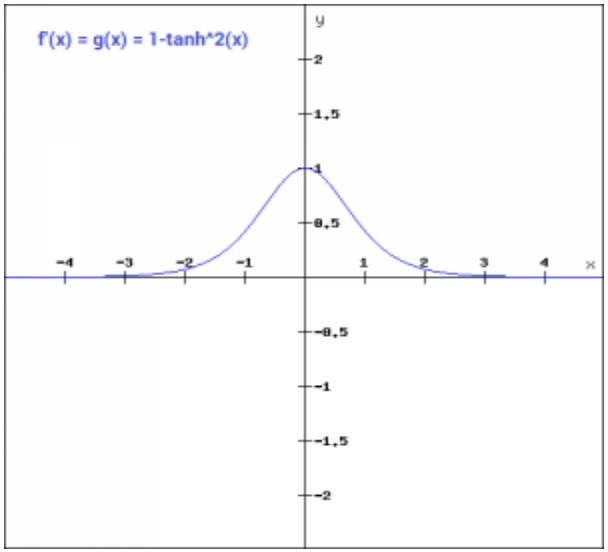
Cons:

- The function is pretty flat beyond the +3 and -3 region.
- Gradients become very small in this region
- values only range from 0 to 1.



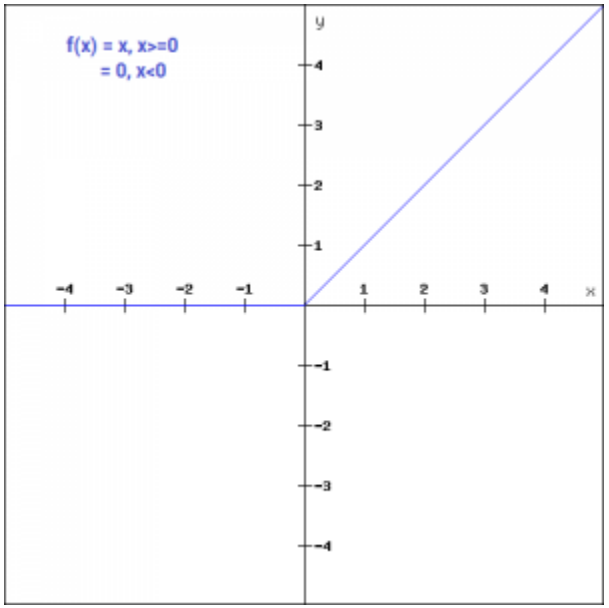
- **Tanh**
  - Similar to the sigmoid function. It is actually just a scaled version of the sigmoid function
  - tanh(x)=2sigmoid(2x)-1





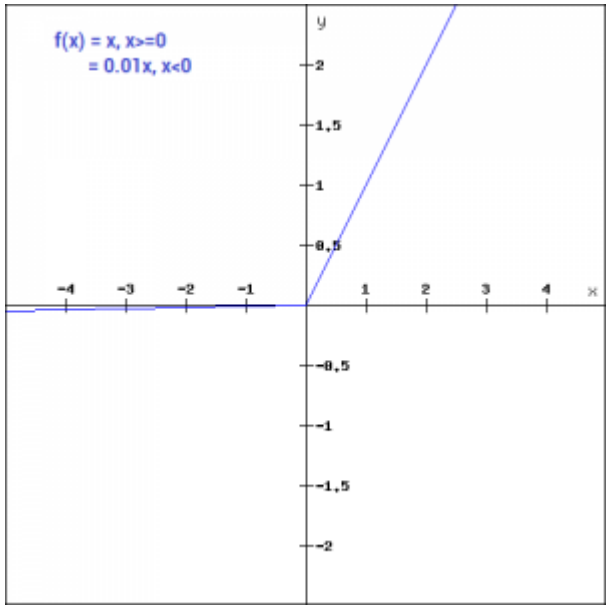
  - Similar to the sigmoid function it has the vanishing gradient problem.

- **ReLU**
  - f(x)=max(0,x)
  - Most widely used activation
  - Non linear
  - It does not activate all the neurons at the same time.
  - Only a few neurons are activated making the network sparse making it efficient and easy for computation.
- **Cons**
  - Gradients moving towards zero.
  - Dead neurons which never get activated



- **Leaky ReLU**
  - f(x)= ax, x<0 Here a is a small value like 0.01

    = x, x>=0



- Removal of the zero gradient.
- No longer encounter dead neurons in that region.

- **Softmax**

$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \quad \text{for } j = 1, \ldots, K.$$

- E.g. Outputs as-[1.2 , 0.9 , 0.75], When we apply the softmax function we would get [0.42 , 0.31, 0.27].
- So now we can use these as probabilities for the value to be in each class.
- Ideally used in the output layer of the classifier where we are actually trying to attain the probabilities to define the class of each input.

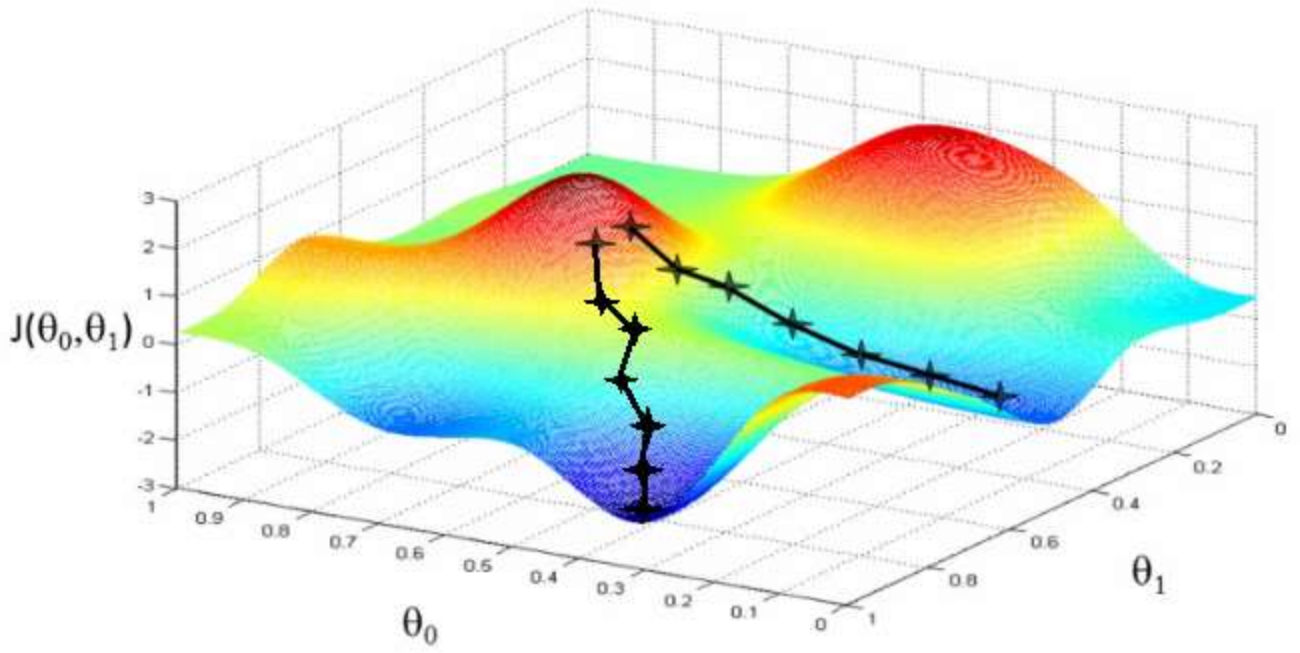
Choosing the right Activation Function

- Sigmoid functions and their combinations generally work better in the case of classifiers
- Sigmoids and tanh functions are sometimes avoided due to the vanishing gradient problem
- ReLU function is a general activation function and is used in most cases these days
- If we encounter a case of dead neurons in our networks the leaky ReLU function is the best choice
- Always keep in mind that ReLU function should only be used in the hidden layers
- As a rule of thumb, you can begin with using ReLU function and then move over to other activation functions in case ReLU doesn't provide with optimum results

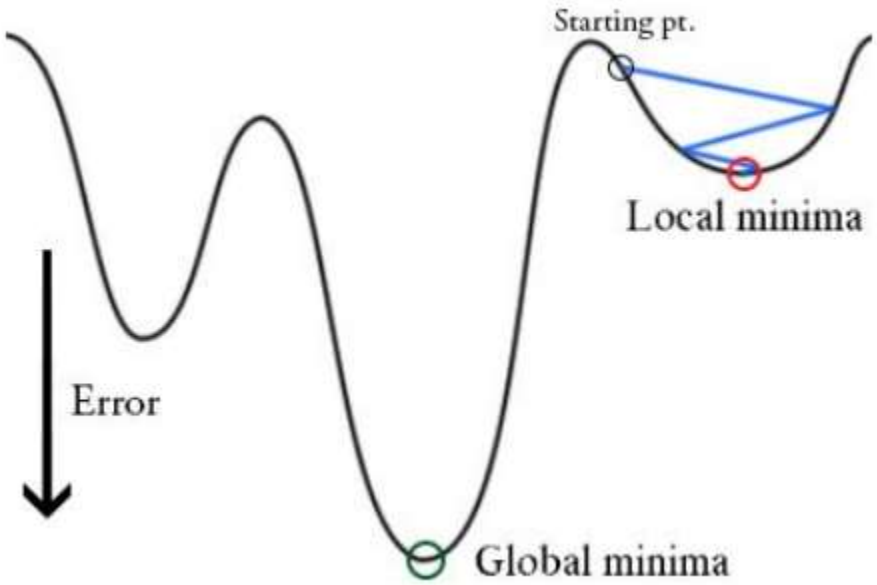
Optimizers

- Gradient Descent Optimizer / **Vanilla Gradient Descent**(Batchwise)
- Stochastic Gradient Descent Optimizer (one observation at a time)
- **Gradient Descent with Momentum**
- Adaptive Gradient Descent (Adagrad)
- **ADAM :** momentum + ADAGRAD.

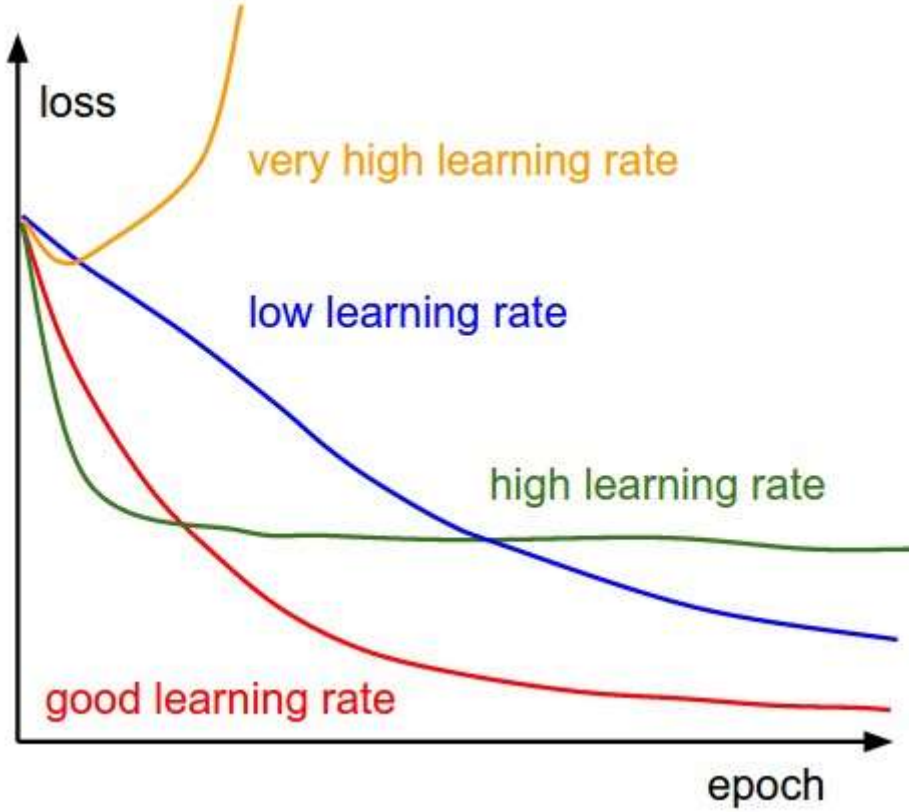
Gradient Descent Optimization



Choosing a starting point?



How to choose learning rate?



Cost Function

- There are various cost functions. Below are some examples:

1. Mean Squared Error Function

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \tilde{y}_i)^2$$

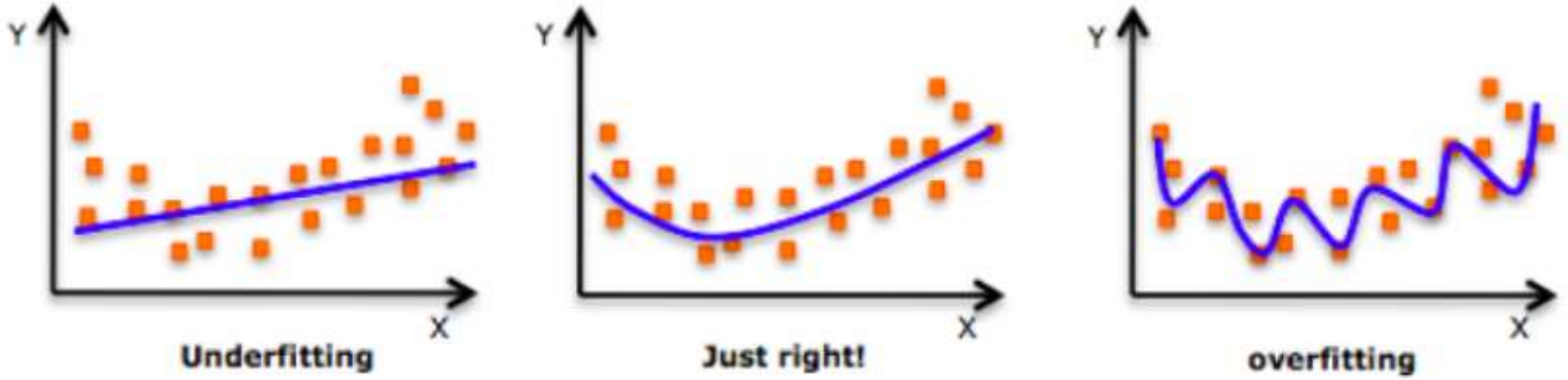
where $\hat{y}_i$ is the predicted output y_i is the actual output

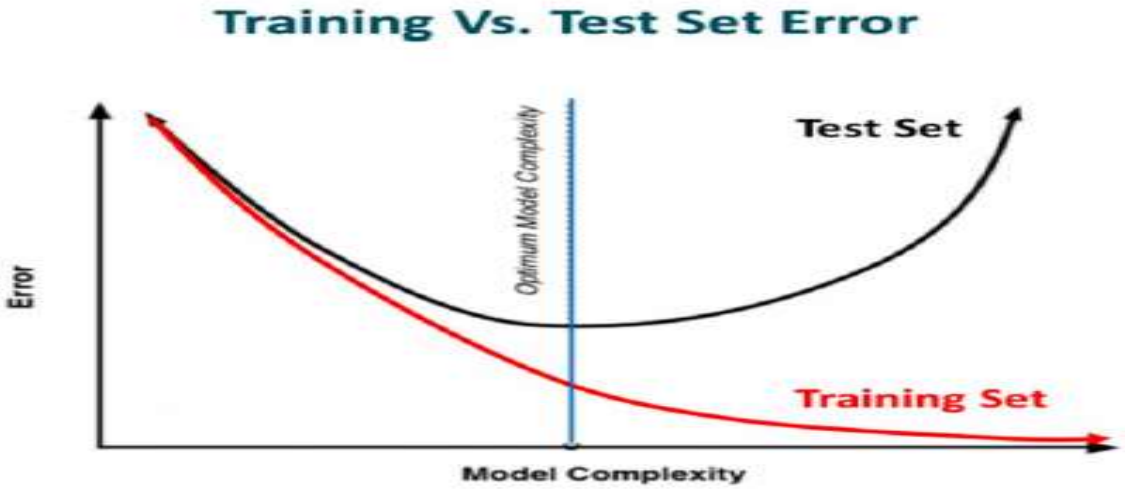
2. Cross-Entropy Function

$$H(y, \hat{y}) = \sum_i y_i \log \frac{1}{\hat{y}_i} = -\sum_i y_i \log \hat{y}_i$$

Regularization Techniques in Deep Learning

- Common Problem:
  - Over fitting





How does Regularization help reduce Overfitting?

- **In deep learning - penalizes the weight matrices of the nodes.**

L2 & L1 REGULARIZATION

- *Cost function = Loss (say, binary cross entropy) + Regularization term*
- In L2, we have:

$$\textbf{Cost Function} = \textbf{Loss} + \lambda * \sum ||w||^2$$

- Lambda- Regularization parameter. (Hyperparameter)
- L2 is also known as weight decay as it forces the weights to decay towards zero (but not exactly zero).
- In L1, we have:

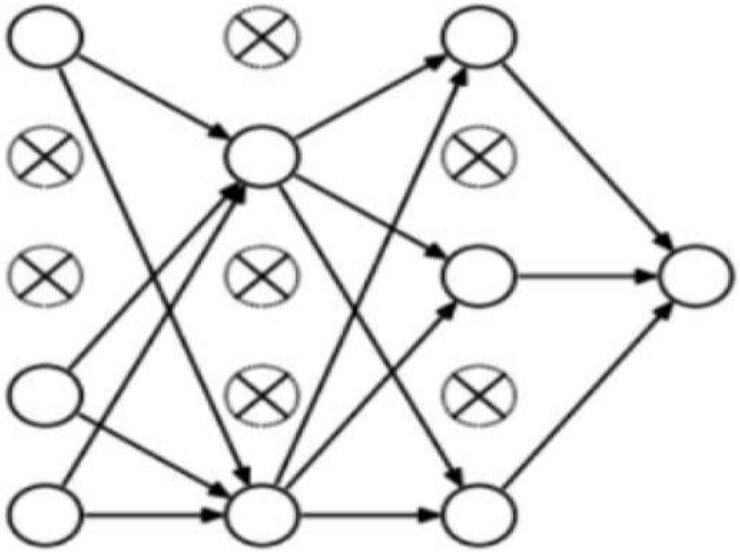
$$\textbf{Cost Function} = \textbf{Loss} + \lambda * \sum ||w||$$

How to choose **Lambda?**

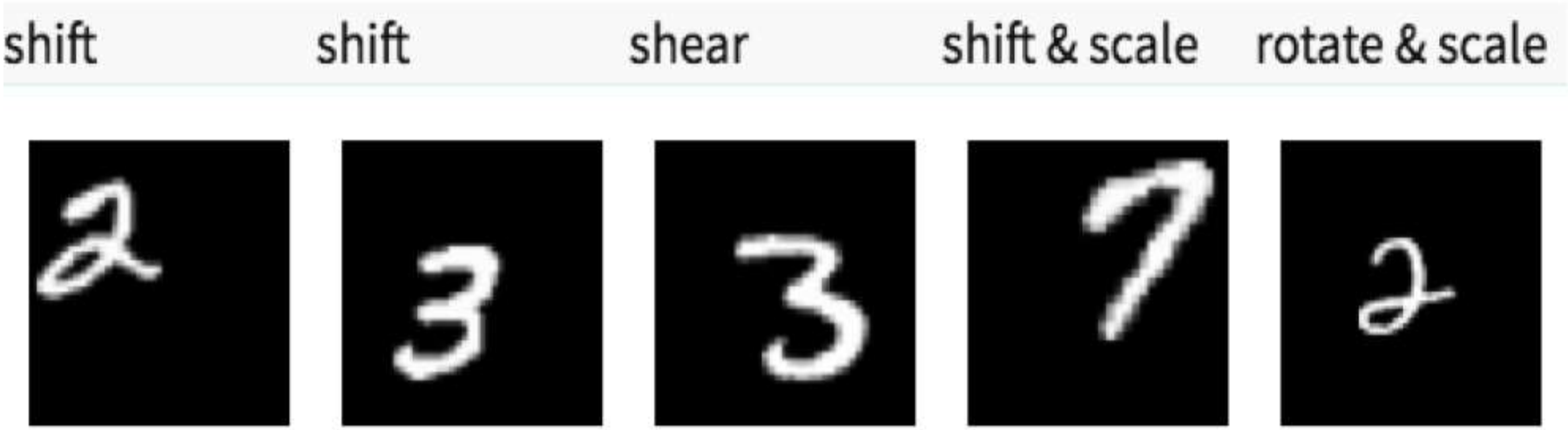
- If your lambda value is too high, your model will be simple, but you run the risk of *underfitting* your data. Your model won't learn enough about the training data to make useful predictions.
- If your lambda value is too low, your model will be more complex, and you run the risk of *overfitting* your data. Your model will learn too much about the particularities of the training data, and won't be able to generalize to new data

Dropout

- At every iteration, it randomly selects some nodes and removes them along with all of their incoming and outgoing connections as shown below.



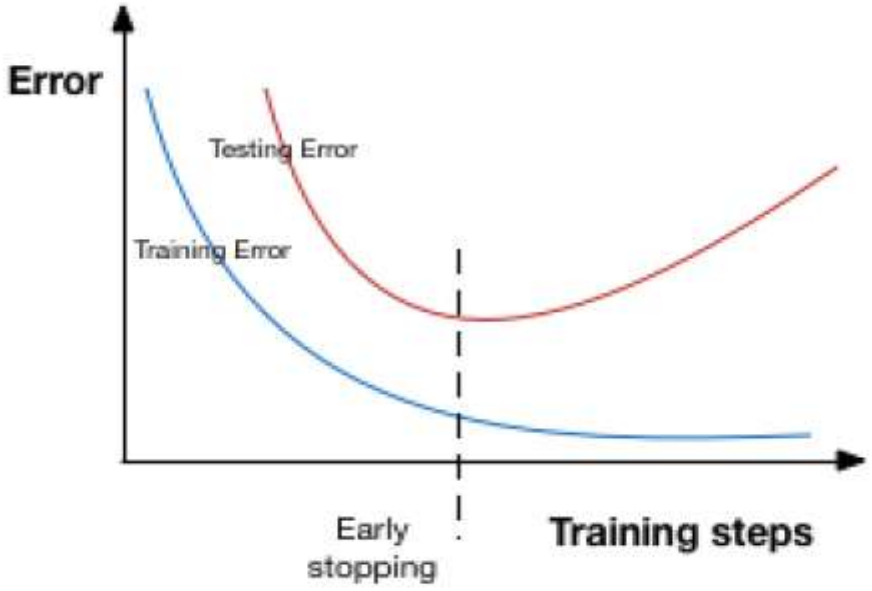
Data Augmentation



- The simplest way to reduce overfitting is to increase the size of the training data.
- If we are dealing with images – rotating the image, flipping, scaling, shifting, etc. can be used for increasing the size of data.

Early stopping

- When we see that the performance on the validation set is getting worse, we immediately stop the training on the model.
- This is known as early stopping.



Thank you

Varsha Mali

 varsha2087@gmail.com

 9870744215

 varsha-mali-200887/