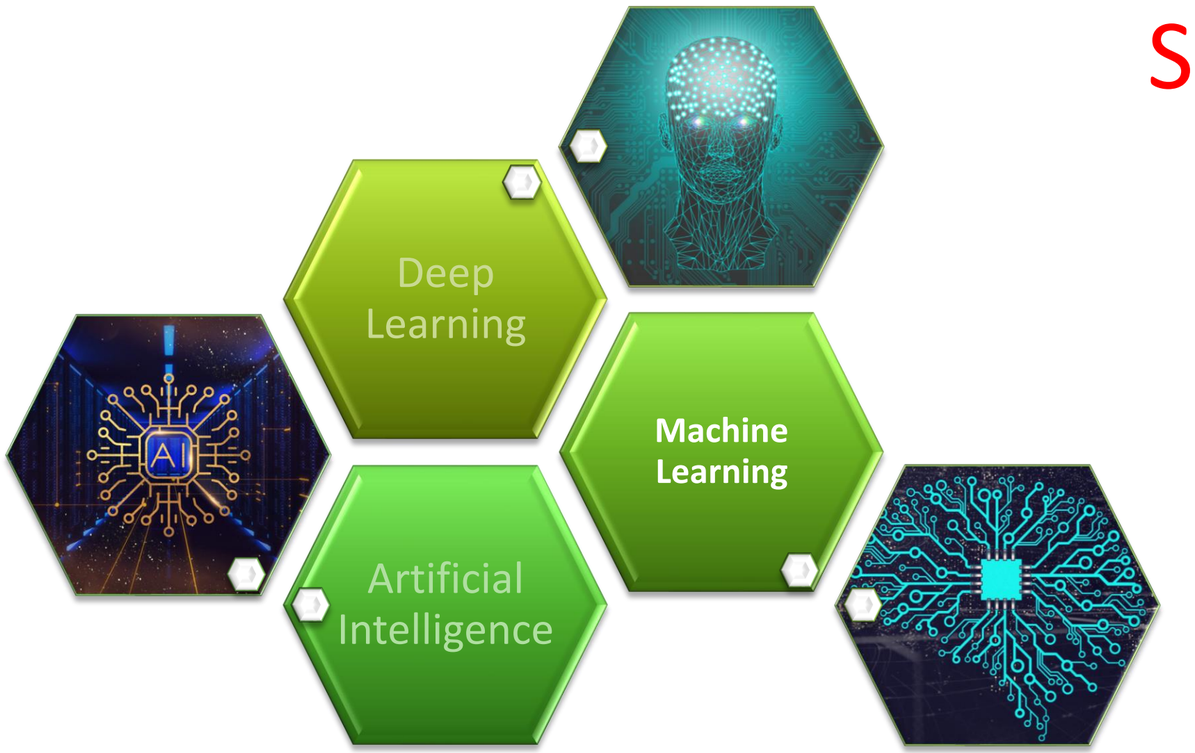


Artificial Intelligence Summer Internship/USRF

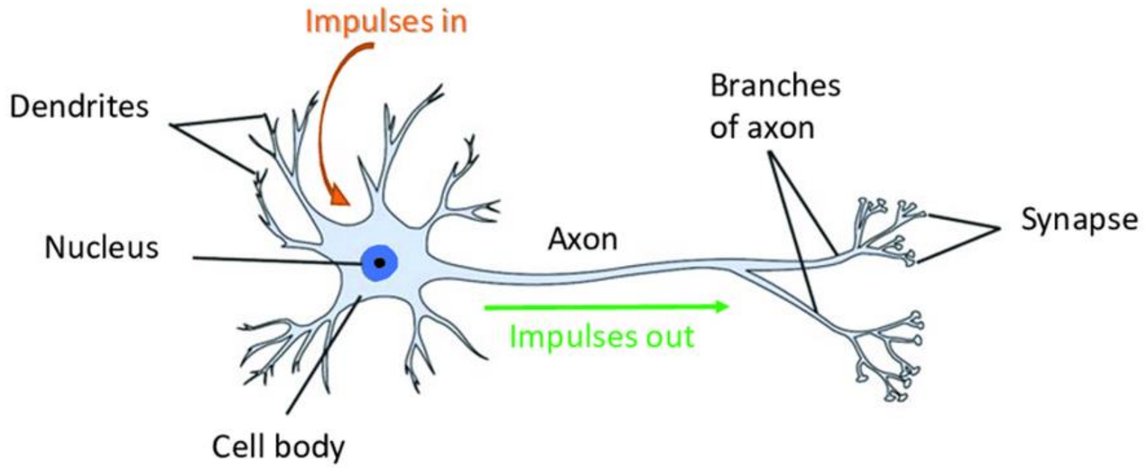
- Day 3: **Loss optimization, Gradient Decent, Learning rate, Over-fitting, Dropout**

Neuron in ANN

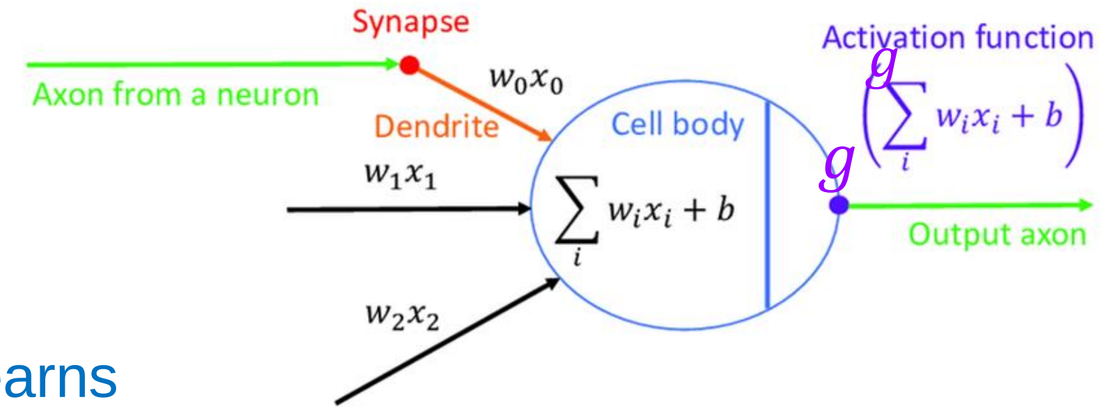
- ANN has Nodes or Units
- Each unit
  - computes a weighted sum of its inputs
  - then, applies an activation function g to derive the output activation

$$y = g\left(\sum_{i=0}^{n} w_i x_i\right), \qquad x_0 = 1$$

- Activation Functions g is mostly a Non-linear function
- Changing weights changes what Neuron Learns



(a) Biological Neuron



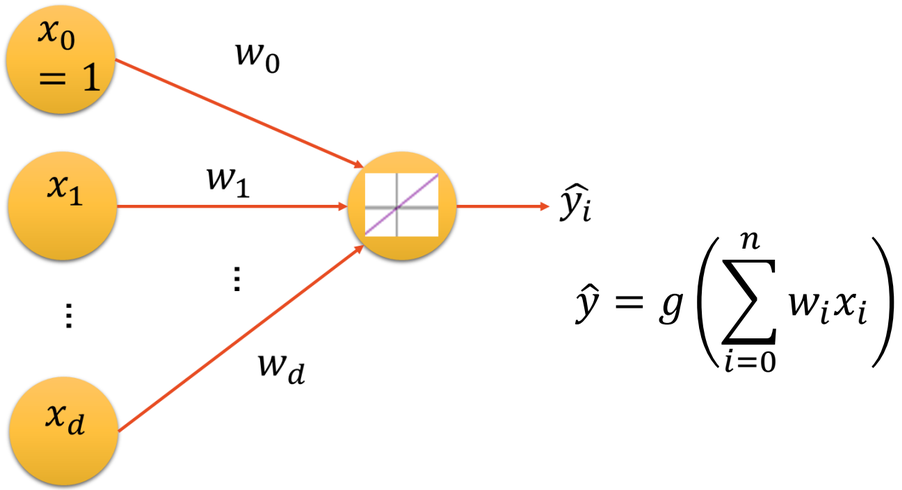
(b) Artificial Neuron

Weight Update Rule

$$W \leftarrow W - \eta \nabla_W \mathrm{E}$$

New Previous

Rate of Convergence Factor/ Learning Rate



- Weights should be such that they MINIMIZE Error in prediction

$$\mathrm{E} = \frac{1}{2}\sum_{i=1}^{N}(\hat{y}_i - y_i)^2$$

Mean Squared Error or Quadratic Loss Function

N : number of observations in data

$\hat{y}_i$: predicted output

y_i : true output

Weight Update Rule

$$W \leftarrow W - \eta \nabla_W \mathrm{E}$$

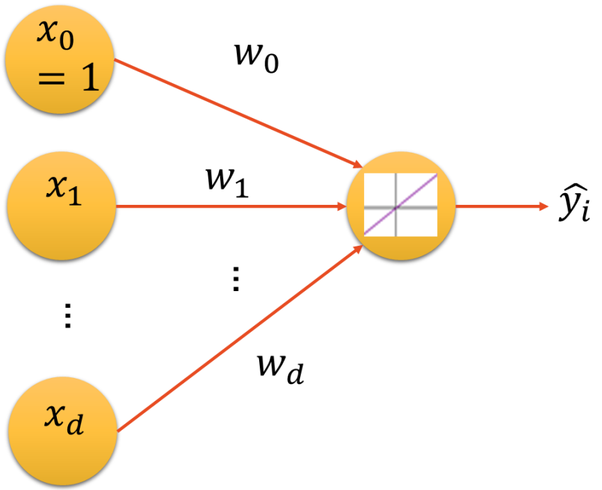
New Previous

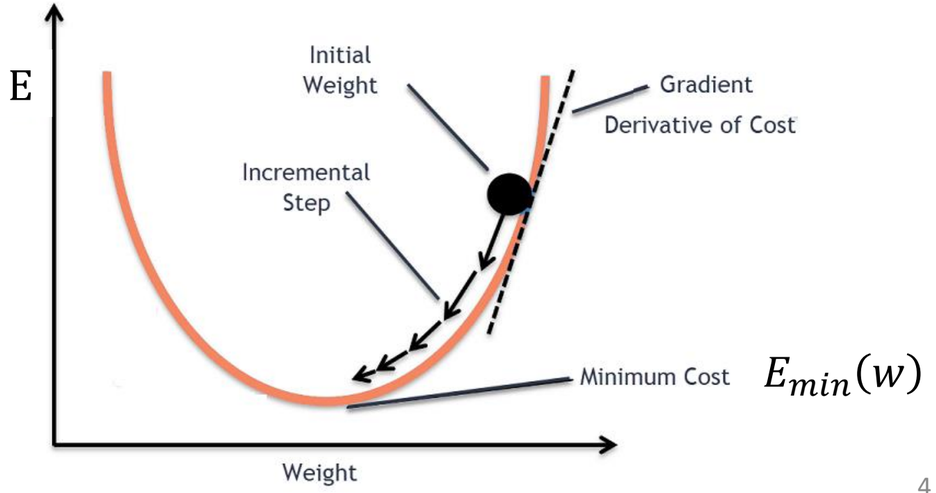
Rate of Convergence Factor/ Learning Rate

- Weights should be such that they MINIMIZE Error in prediction

$$\mathrm{E} = \frac{1}{2}\sum_{i=1}^{N}(\hat{y}_i - y_i)^2$$

- Gradient of Error w.r.t. weight $(\nabla_W \mathrm{E})$ decreases towards minima of the function

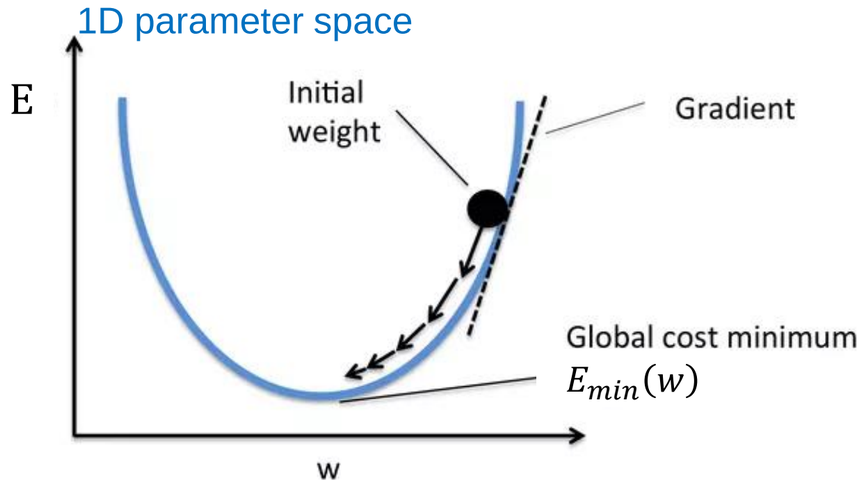


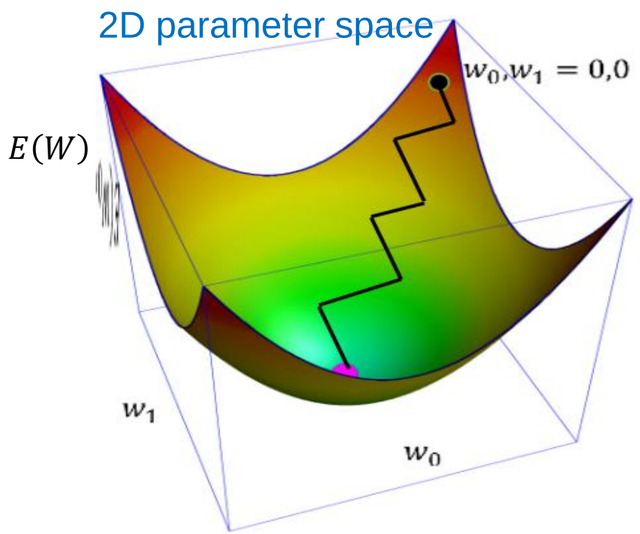


Gradient Descent

1. Randomly Initialize Weights, W
2. Evaluate Gradient of Error w.r.t. weight $\nabla_W \mathrm{E}$
3. Update the weights:
   $W \leftarrow W - \eta \nabla_W \mathrm{E}$
4. Repeat 2 and 3 until convergence

$W \leftarrow W - \eta \nabla_W \mathrm{E}$ always updates weights in the direction of decreasing gradient



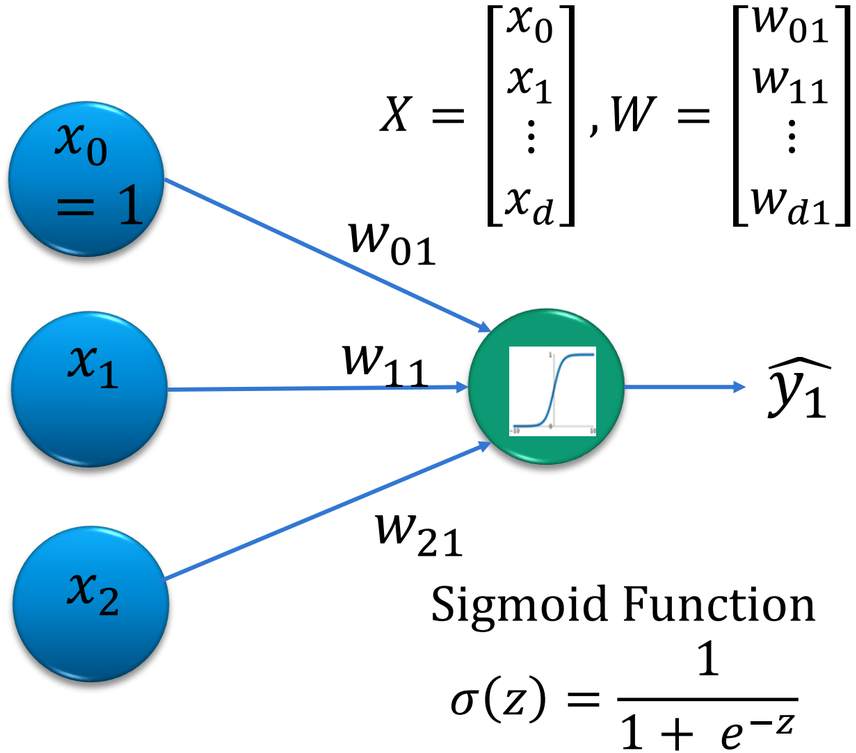


Single layer, Single output NN

- Heaviside Step activation is non-differentiable
- Consider Single layer, Single output NN, with sigmoid activation function

- Pre-activation, $\theta_j = \sum_{i=0}^{d} w_{ij} x_i$
- Output activations, $\hat{y}_j = \sigma(\theta_j),\ j = 1$ for single o/p
- Error in prediction (Mean Square Error Loss)

$\mathrm{E} = \frac{1}{2}(\hat{y}_j - y_j)^2$ for one observation $(X \quad , y_j)$



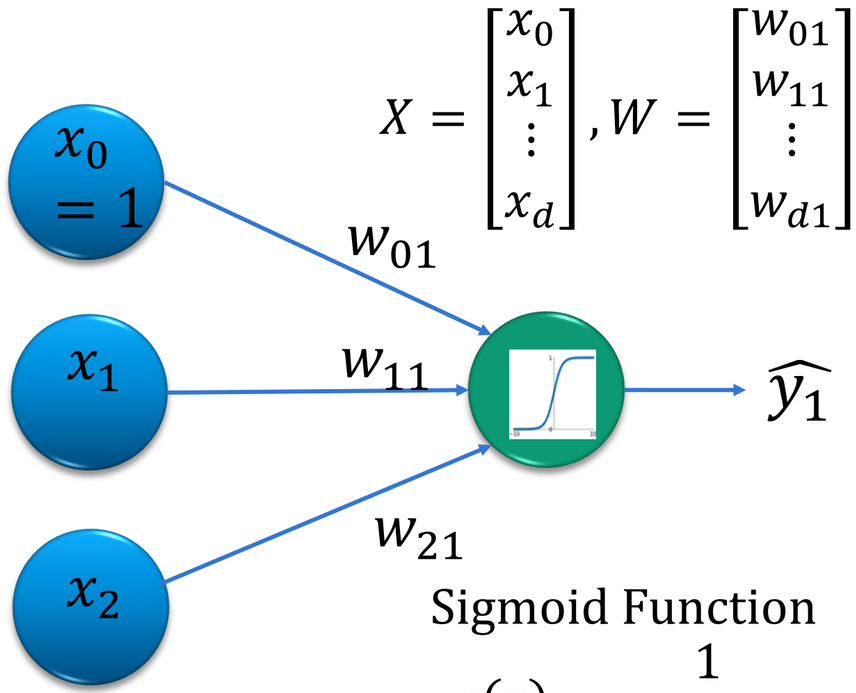
w_{ij}^k = Weight of connection b/w i^{th} node in $(k-1)^{th}$ layer to j^{th} node in k^{th} layer

Single layer, Single output NN

- Pre-activation, $\theta_j = \sum_{i=0}^{d} w_{ij} x_i$
- Output activations, $\hat{y}_j = \sigma(\theta_j),\ j = 1$
- Error in prediction

$$\mathrm{E} = \frac{1}{2}\left(\hat{y}_j - y_j\right)^2$$

- Gradient of Error/Loss function
- $\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial \hat{y}_j} \frac{\partial \hat{y}_j}{\partial \theta_j} \frac{\partial \theta_j}{\partial w_{ij}}$ (Chain Rule)
- $\frac{\partial E}{\partial w_{ij}} =$

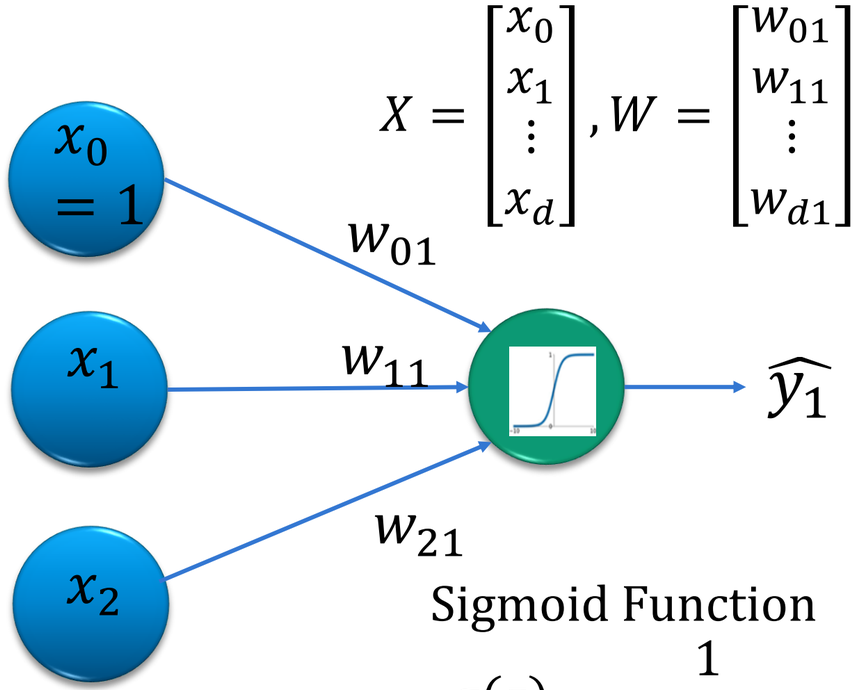


Single layer, Single output NN

- Pre-activation, $\theta_j = \sum_{i=0}^{d} w_{ij} x_i$
- Output activations, $\hat{y}_j = \sigma(\theta_j),\ j = 1$
- Error in prediction

$$\mathrm{E} = \frac{1}{2}\left(\hat{y}_j - y_j\right)^2$$

- Gradient of Error/Loss function
- $\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial \hat{y}_j}\frac{\partial \hat{y}_j}{\partial \theta_j}\frac{\partial \theta_j}{\partial w_{ij}}$ (Chain Rule)
- $\frac{\partial E}{\partial w_{ij}} = \left(\hat{y}_j - y_j\right)\frac{\partial \hat{y}_j}{\partial \theta_j}\frac{\partial \theta_j}{\partial w_{ij}}$



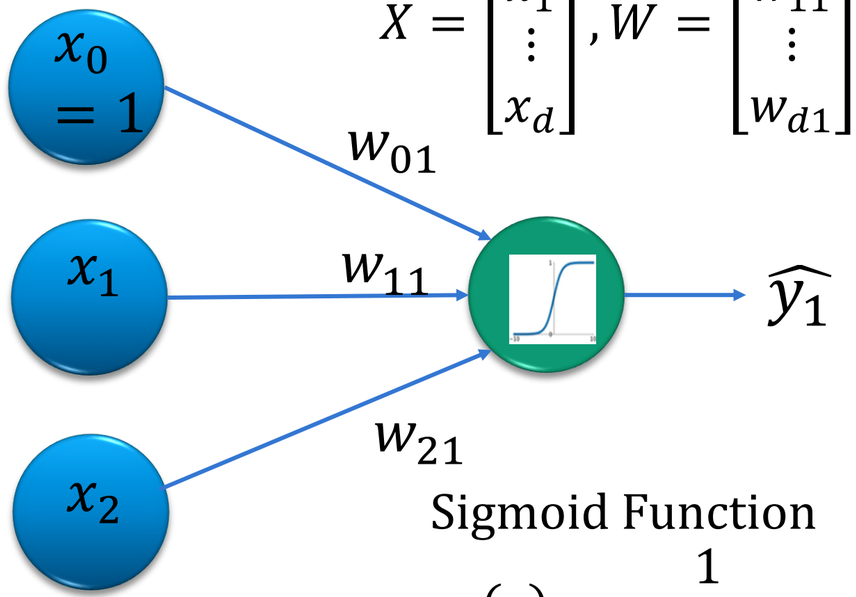
Single layer, Single output NN

- Pre-activation, $\theta_j = \sum_{i=0}^{d} w_{ij} x_i$
- Output activations, $\hat{y}_j = \sigma(\theta_j),\ j = 1$
- Error in prediction

$$\mathrm{E} = \frac{1}{2}(\hat{y}_j - y_j)^2$$

- Gradient of Error/Loss function
- $\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial \hat{y}_j} \frac{\partial \hat{y}_j}{\partial \theta_j} \frac{\partial \theta_j}{\partial w_{ij}}$ (Chain Rule)
- $\frac{\partial E}{\partial w_{ij}} = (\hat{y}_j - y_j)\hat{y}_j(1 - \hat{y}_j)\frac{\partial \theta_j}{\partial w_{ij}}$

$$X = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_d \end{bmatrix}, W = \begin{bmatrix} w_{01} \\ w_{11} \\ \vdots \\ w_{d1} \end{bmatrix}$$



Sigmoid Function

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

$$\sigma'(z) = \sigma(z)(1 - \sigma(z))$$

$$\frac{\partial \hat{y}_j}{\partial \theta_j} = \sigma(\theta_j)(1 - \sigma(\theta_j))$$

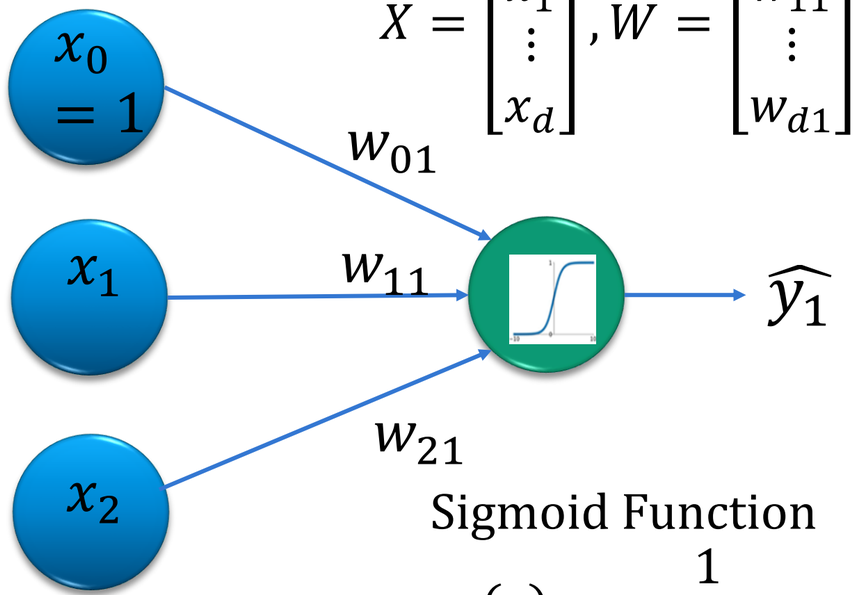
Single layer, Single output NN

- Pre-activation, $\theta_j = \sum_{i=0}^{d} w_{ij} x_i$
- Output activations, $\hat{y}_j = \sigma(\theta_j),\ j = 1$
- Error in prediction

$$\mathrm{E} = \frac{1}{2}(\hat{y}_j - y_j)^2$$

- Gradient of Error/Loss function
- $\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial \hat{y}_j} \frac{\partial \hat{y}_j}{\partial \theta_j} \frac{\partial \theta_j}{\partial w_{ij}}$ (Chain Rule)
- $\frac{\partial E}{\partial w_{ij}} = (\hat{y}_j - y_j)\hat{y}_j(1 - \hat{y}_j)x_i$

$$X = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_d \end{bmatrix}, W = \begin{bmatrix} w_{01} \\ w_{11} \\ \vdots \\ w_{d1} \end{bmatrix}$$



Sigmoid Function

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

$$\theta_j = w_{0j}x_0 + \cdots + w_{ij}x_i + \cdots w_{dj}x_d$$

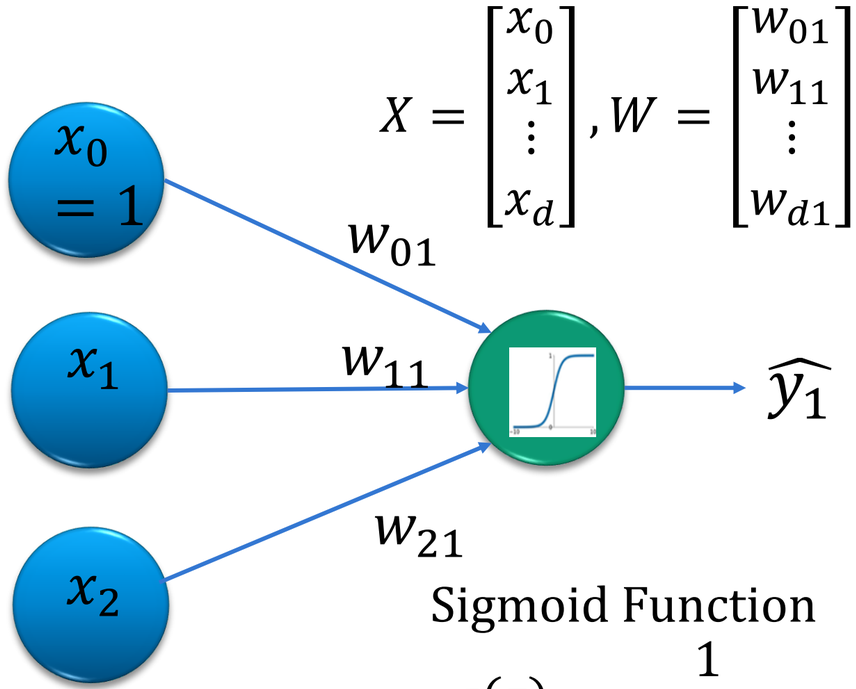
$$\frac{\partial \theta_j}{\partial w_{ij}} = x_i$$

Single layer, Single output NN

- Pre-activation, $\theta_j = \sum_{i=0}^{d} w_{ij} x_i$
- Output activations, $\hat{y}_j = \sigma(\theta_j),\ j = 1$
- Error in prediction

$$\mathrm{E} = \frac{1}{2}(\hat{y}_j - y_j)^2$$

- Gradient of Error/Loss function
- $\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial \hat{y}_j} \frac{\partial \hat{y}_j}{\partial \theta_j} \frac{\partial \theta_j}{\partial w_{ij}}$ (Chain Rule)
- $\frac{\partial E}{\partial w_{ij}} = (\hat{y}_j - y_j)\hat{y}_j(1 - \hat{y}_j)x_i$
- Weight Updation: $w_{ij} \leftarrow w_{ij} - \eta\,(\hat{y}_j - y_j)\hat{y}_j(1 - \hat{y}_j)\,x_i$



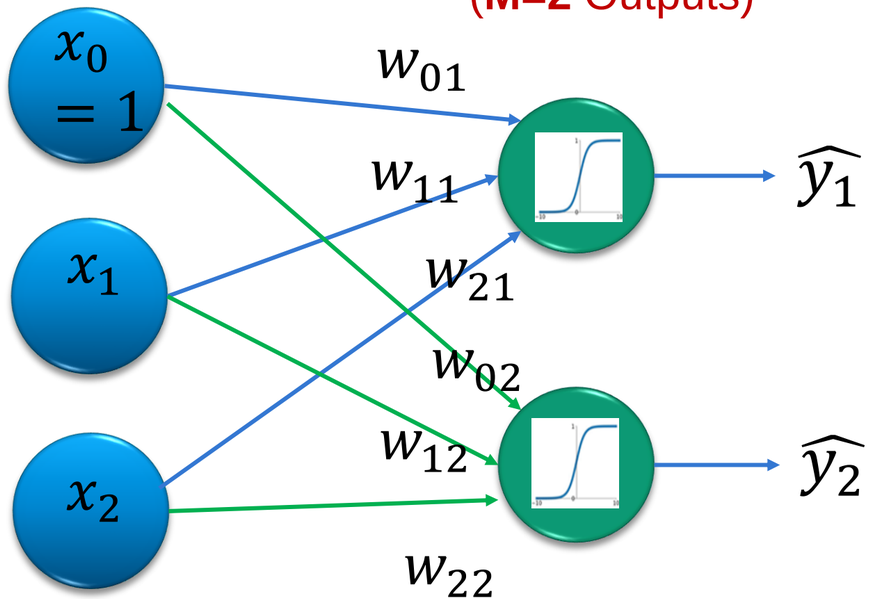
Weight updated Simultaneously

Single layer, Multiple output NN

- Pre-activation, $\theta_j = \sum_{i=0}^{d} w_{ij} x_i$
- Output activations, $\hat{y}_j = \sigma(\theta_j),\ j = 1, \ldots, M$
- Error in prediction (sum over all classes)

$$\mathrm{E} = \frac{1}{2}\sum_{j=1}^{M}(\hat{y}_j - y_j)^2$$

Single Layer
(**M=2** Outputs)



w_{ij} = Weight of connection b/w
i^{th} node in $previous$ layer to
j^{th} node in $next$ layer

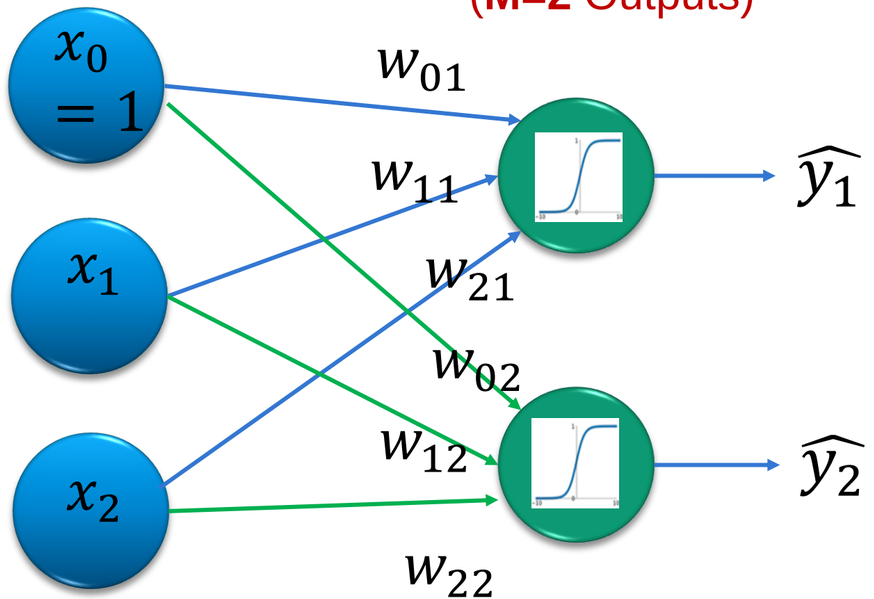
Single layer, Multiple output NN

- Pre-activation, $\theta_j = \sum_{i=0}^{d} w_{ij} x_i$
- Output activations, $\hat{y}_j = \sigma(\theta_j),\ j = 1, \ldots, M$
- Error in prediction (sum over all classes)

$$\mathrm{E} = \frac{1}{2}\sum_{j=1}^{M}(\hat{y}_j - y_j)^2$$

- Gradient of Error/Loss function
- $\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial \hat{y}_j}\frac{\partial \hat{y}_j}{\partial \theta_j}\frac{\partial \theta_j}{\partial w_{ij}}$ (Chain Rule)
- $\frac{\partial E}{\partial w_{ij}} = (\hat{y}_j - y_j)\hat{y}_j(1 - \hat{y}_j)x_i$
- Weight Updation: $w_{ij} \leftarrow w_{ij} - \eta\,(\hat{y}_j - y_j)\hat{y}_j(1 - \hat{y}_j)\,x_i$

Single Layer
(**M=2** Outputs)



Weight updated Simultaneously

Types of Gradient Descent

- Batch Gradient Descent (BGD)
- Stochastic Gradient Descent (SGD)
- Mini Batch Gradient Descent

Types of Gradient Descent

- Batch Gradient Descent (BGD): $W \leftarrow W - \eta \sum_{i=1}^{N} (\hat{y}_i - y_i)\, \hat{y}_j \left(1 - \hat{y}_j\right) X_i$

It uses the *entire dataset* at every step, making it slow for large datasets. However, it is computationally efficient, since it produces a *stable* error gradient and a stable convergence

- Stochastic Gradient Descent (SGD)

- Mini Batch Gradient Descent

Types of Gradient Descent

- Batch Gradient Descent (BGD): $W \leftarrow W - \eta \sum_{i=1}^{N} (\hat{y}_i - y_i)\, \hat{y}_j \left(1 - \hat{y}_j\right) X_i$

It uses the *entire dataset* at every step, making it slow for large datasets. However, it is computationally efficient, since it produces a *stable* error gradient and a stable convergence

- Stochastic Gradient Descent (SGD): $W \leftarrow W - \eta (\hat{y}_i - y_i) \hat{y}_j \left(1 - \hat{y}_j\right) X_i$

It is on the other extreme of the idea, using a *single example* (batch of 1) for each learning step. Much faster, may return *noisy gradients* which can cause the error rate to jump around

- Mini Batch Gradient Descent

Types of Gradient Descent

- Batch Gradient Descent (BGD): $W \leftarrow W - \eta \sum_{i=1}^{N} (\hat{y}_i - y_i)\, \hat{y}_j \left(1 - \hat{y}_j\right) X_i$

It uses the *entire dataset* at every step, making it slow for large datasets. However, it is computationally efficient, since it produces a *stable* error gradient and a stable convergence

- Stochastic Gradient Descent (SGD): $W \leftarrow W - \eta (\hat{y}_i - y_i) \hat{y}_j \left(1 - \hat{y}_j\right) X_i$

It is on the other extreme of the idea, using a *single example* (batch of 1) per each learning step. Much faster, may return *noisy gradients* which can cause the error rate to jump around

- Mini Batch Gradient Descent: $W \leftarrow W - \eta \sum_{X_i \in Minibatch} (\hat{y}_i - y_i)\, \hat{y}_j \left(1 - \hat{y}_j\right) X_i$

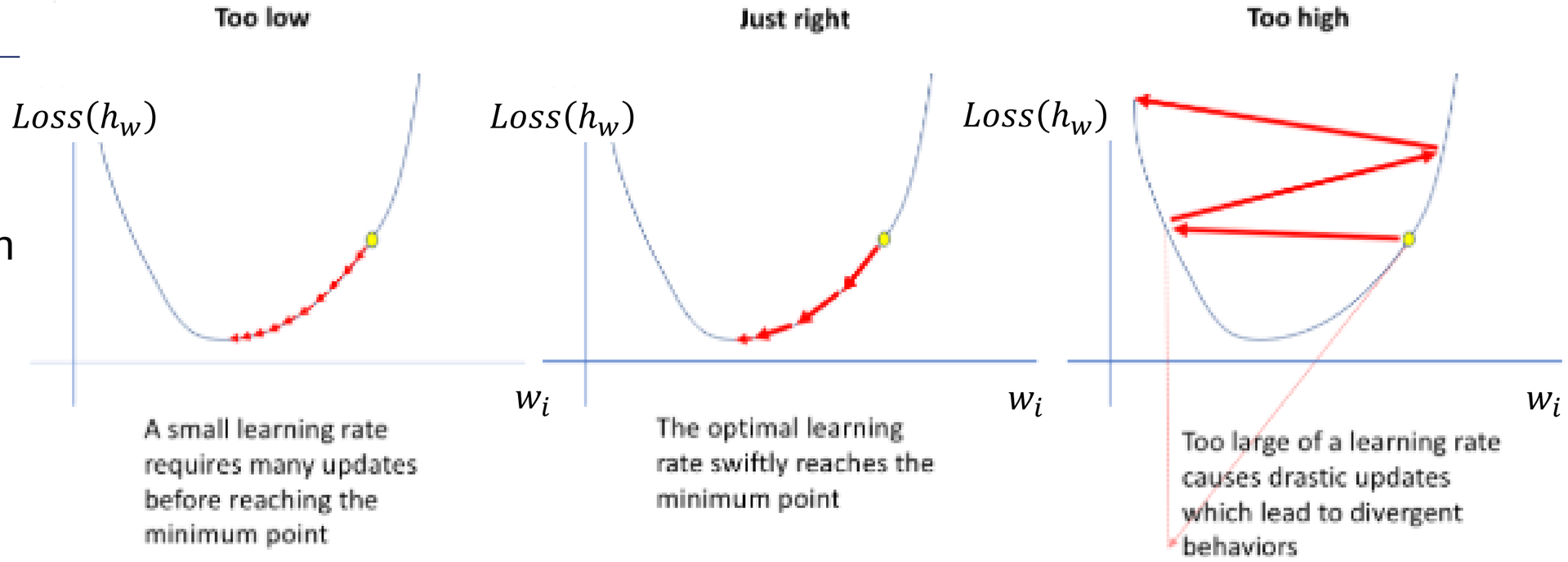
  Computes the gradients on small random sets of instances called *mini batches*. Reduce noise from SGD and still more efficient than BGD

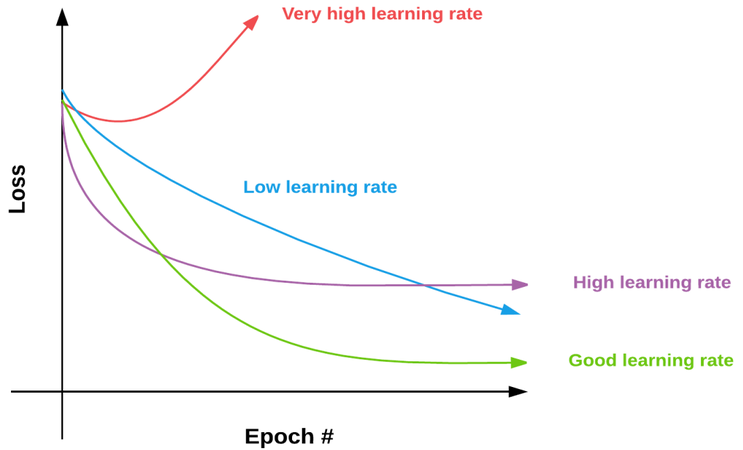
Learning Rate

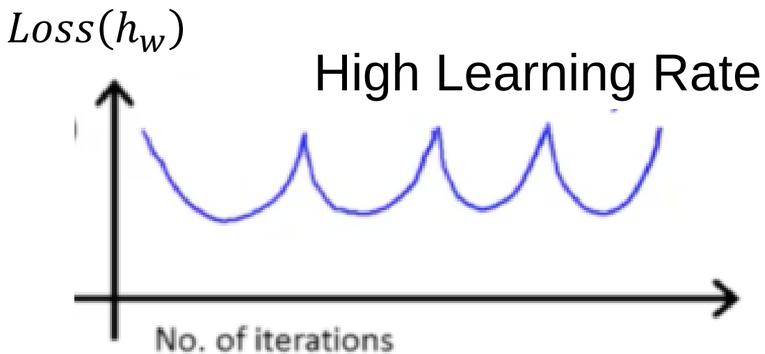
Weight Updation Equation

$W \leftarrow W - \eta \nabla_W \mathrm{E}$

- η= Learning Rate is a Hyperparameter (selected/tuned, not learned during training)





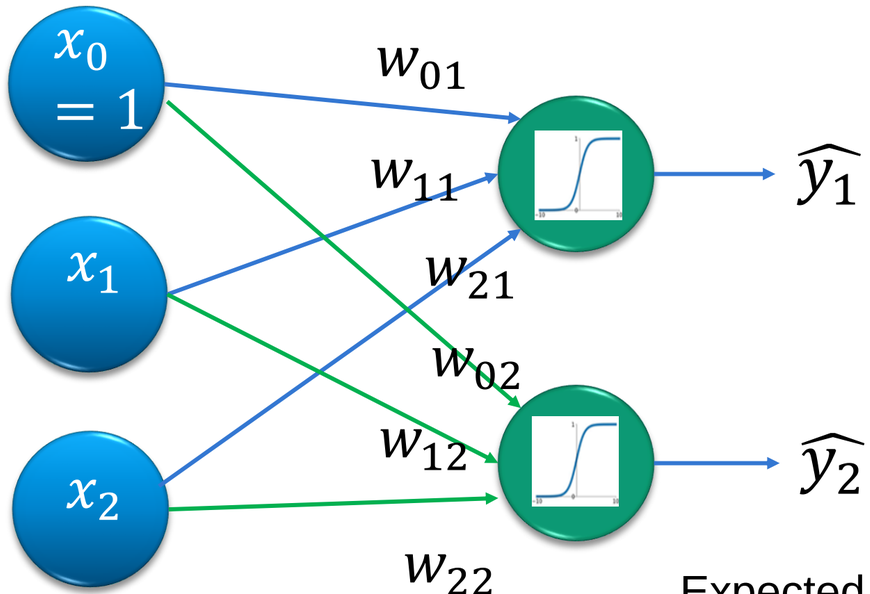


https://srdas.github.io/DLBook/GradientDescentTechniques.html#issues-with-gradient-descent

Training neural networks-

Backpropagation algorithm

Single Layer Multiple Output NN

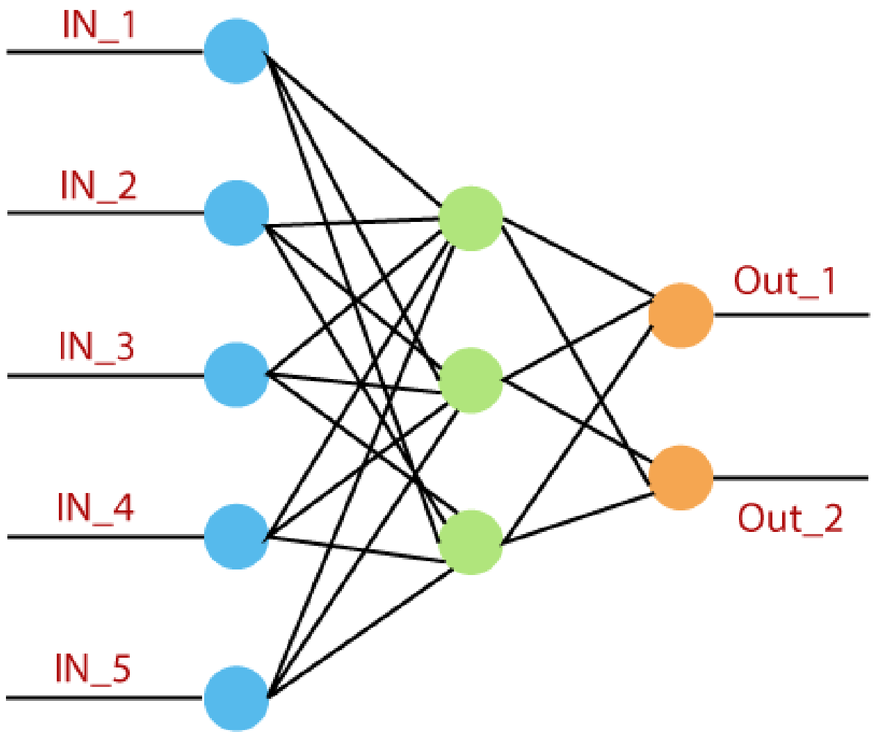


Expected Output is Known

Input Layer

Output Layer

Multi Layer Multiple Output NN

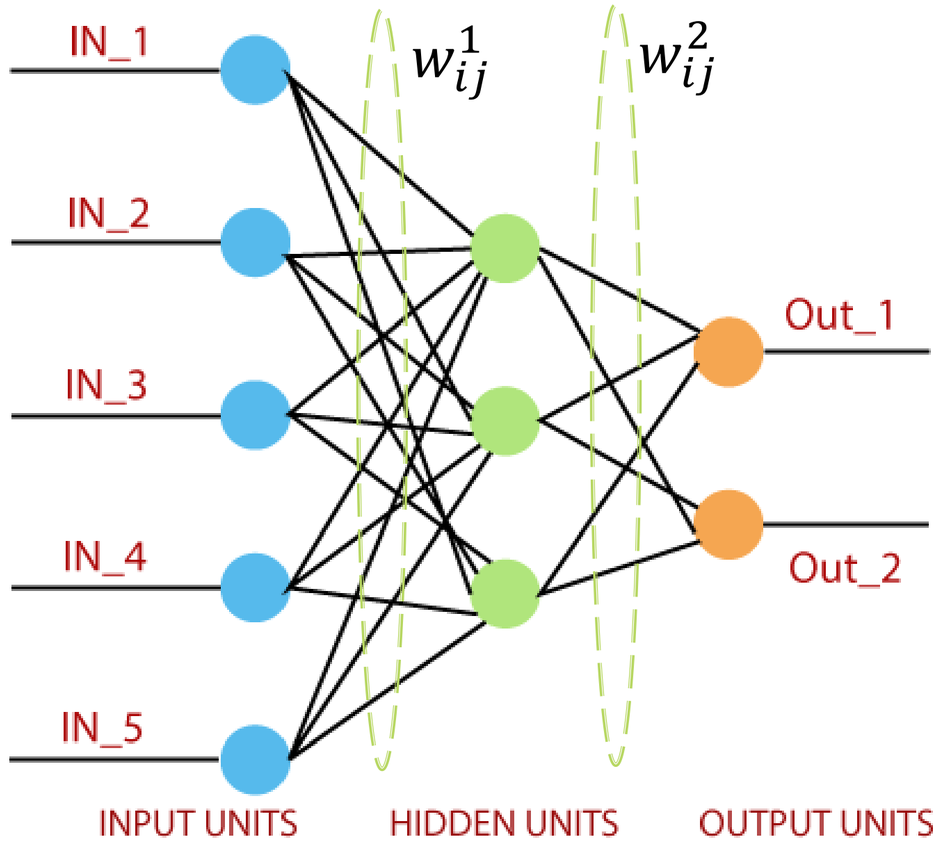


Expected Output Not Known

Expected Output is Known

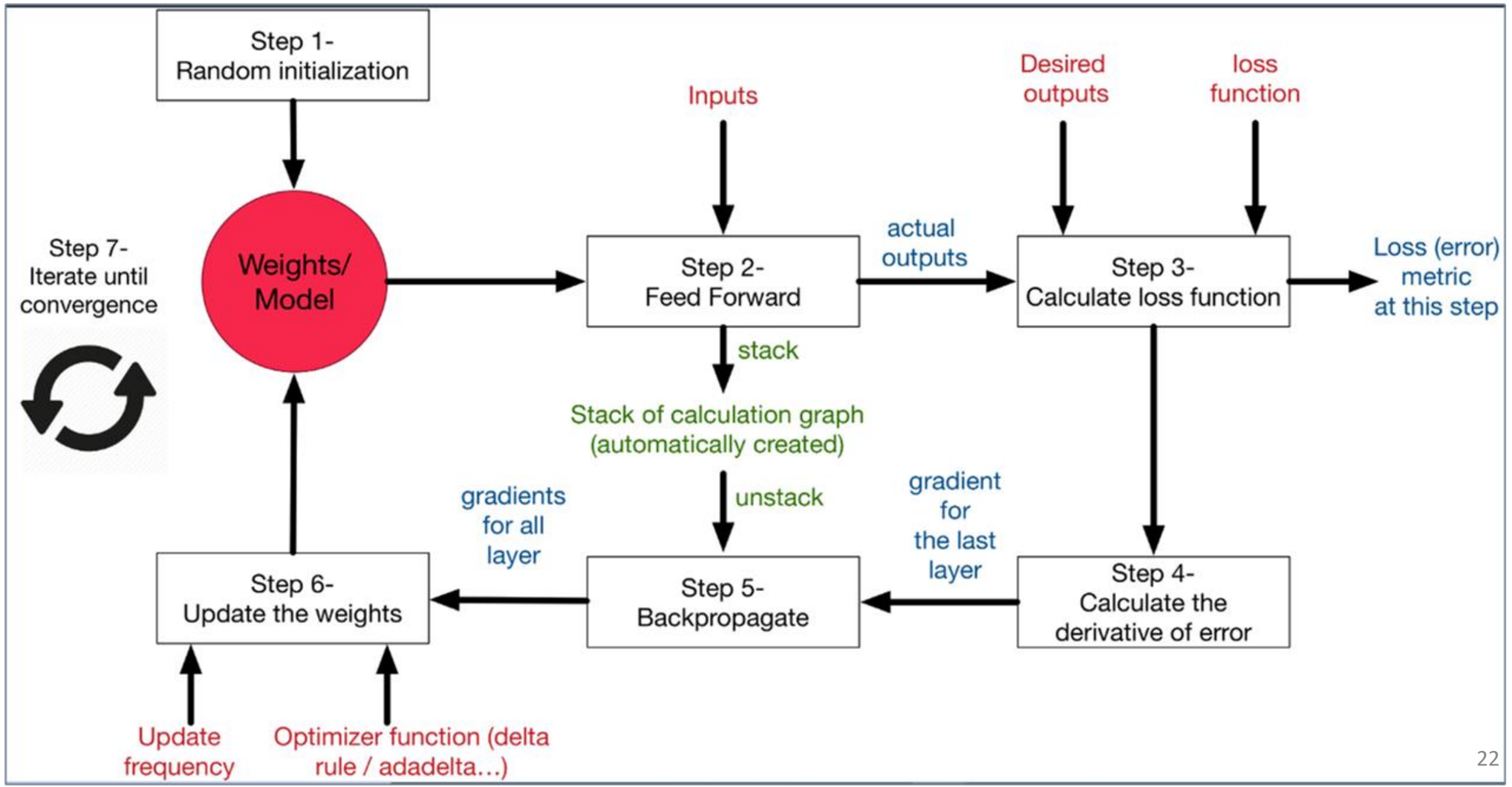
Multi-layer Perceptron

- Output of hidden layers is not known
- Error cannot be calculated for output of hidden layers
- Training of weights for hidden layers is done by assessing the effect of weights of hidden layer on the Error of output layer (Backpropagation)



Training of Multilayer Neural Networks

- Forward Propagation:
  - Weights are fixed
  - Loss is calculated for each input
- Backward Propagation:
  - Loss is considered as a function of weights
  - Weights are varied to reduce loss



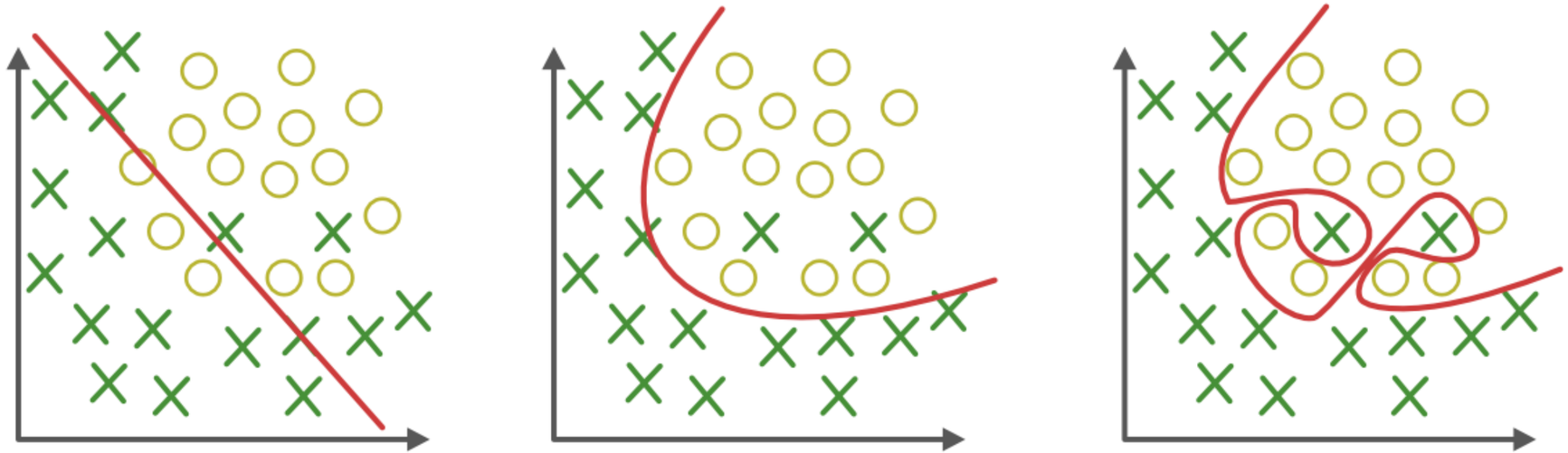
Training neural networks-

Overfitting

Which Model is Better?

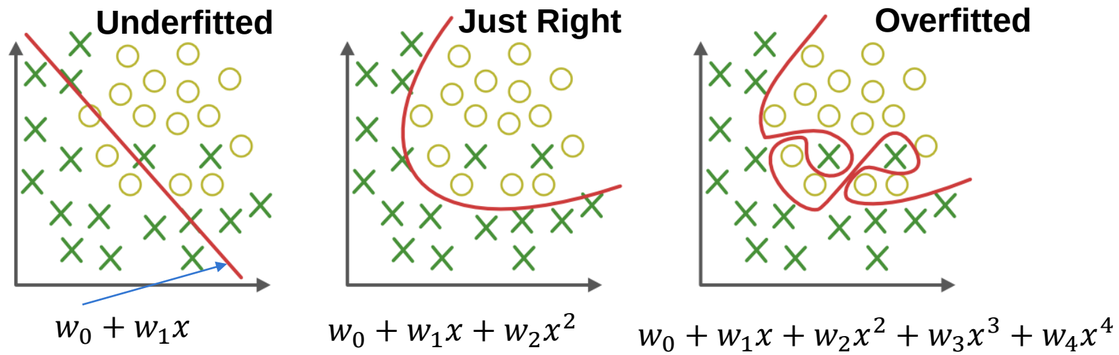
Example of Binary Classification

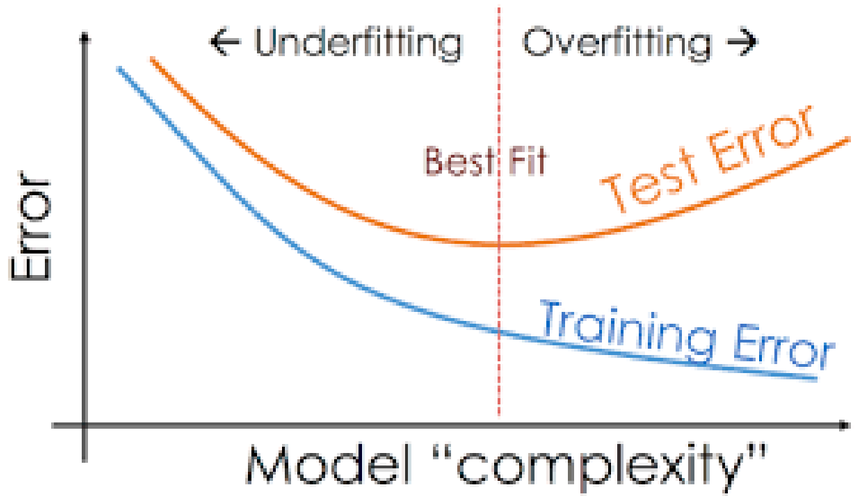
- Learning is an ill posed problem: Multiple possible hypothesis



- The real aim of supervised learning is to do well on test data that is not known during learning
- Generalization refers to How well the model trained on the training data predicts the correct output for new instances

Trading off goodness of fit against complexity of the model

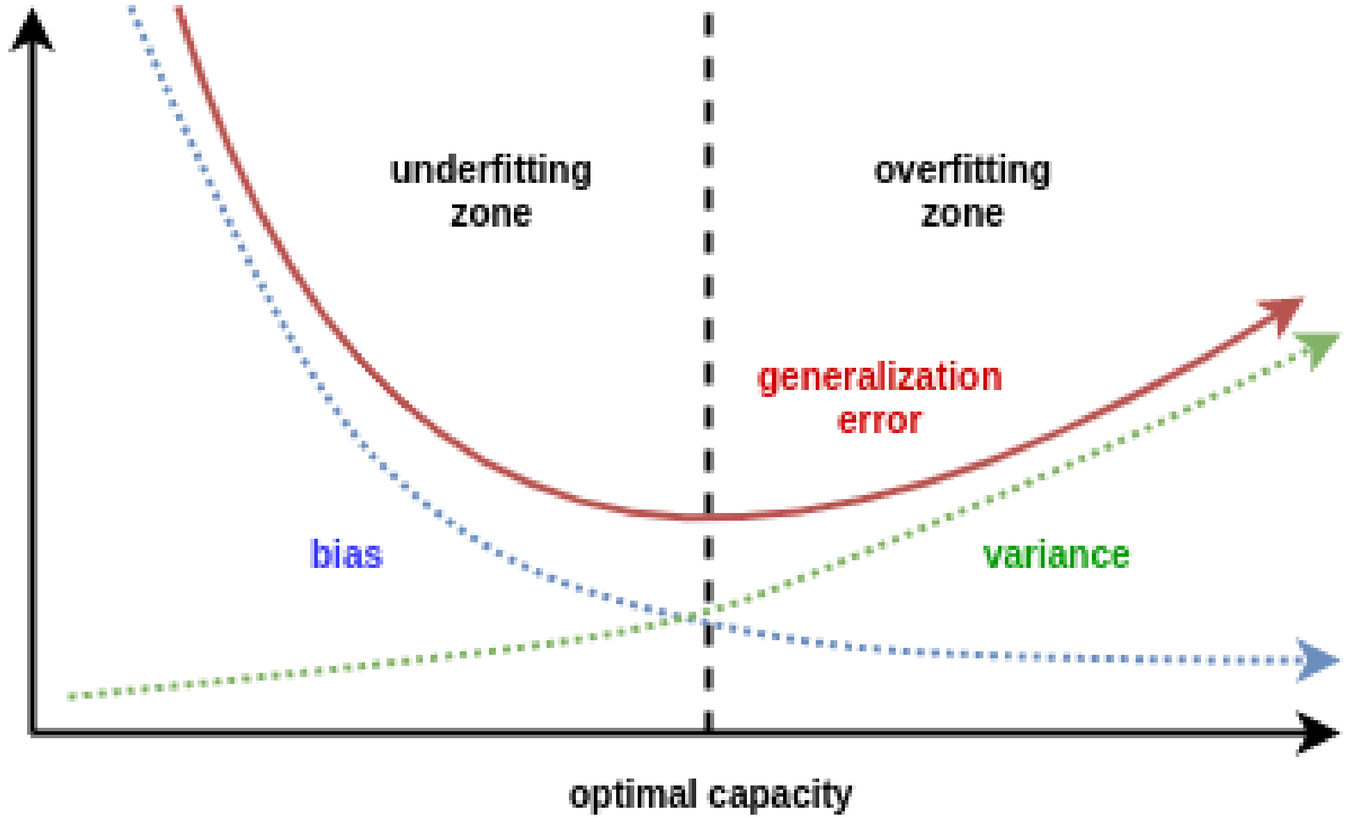




Simple model has less parameters to be learned (Low complexity, low capacity)	**Complex model** has more parameters to be learned (High complexity, High capacity)
Model may **Underfit**, it may not capture underlying trend of the data	Model may **Overfit**, it may start learning from noise and inaccurate data entries
Higher error for training data, may give high error for validation data also	Lower error for training data, may give higher error for validation data
High Bias, Low Variance	Low Bias, High Variance

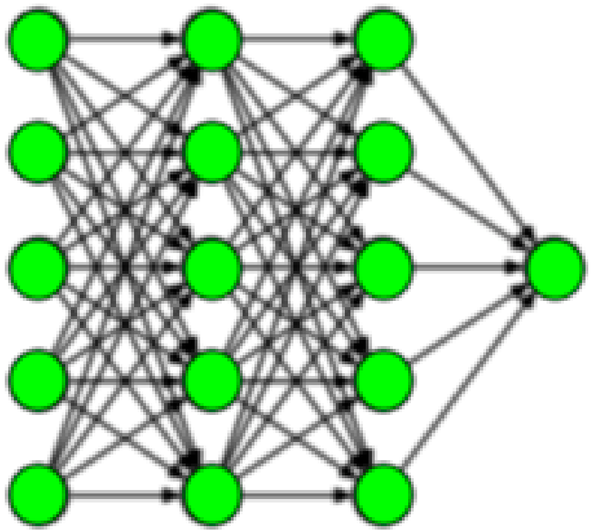
Bias vs Variance

- **Bias** of a Model: Underlying assumptions to make learning possible. Simpler model=>More assumption=> High Bias
- **Variance** of a Model: Variability of model for given data points, Model with high variance pays a lot of attention to training data, may end up memorizing data rather than learning from it

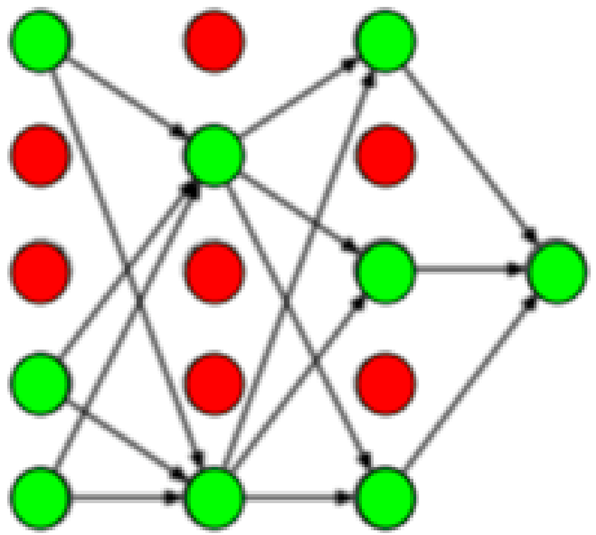


Dropout: Solution to Overfitting

- During training, some number of layer outputs are randomly ignored or "dropped out"
- During weight updation, the layer configuration appears "new"
- Provides Regularization by avoiding co-adaption between network layers to correct mistakes from prior layers



Standard Neural Network

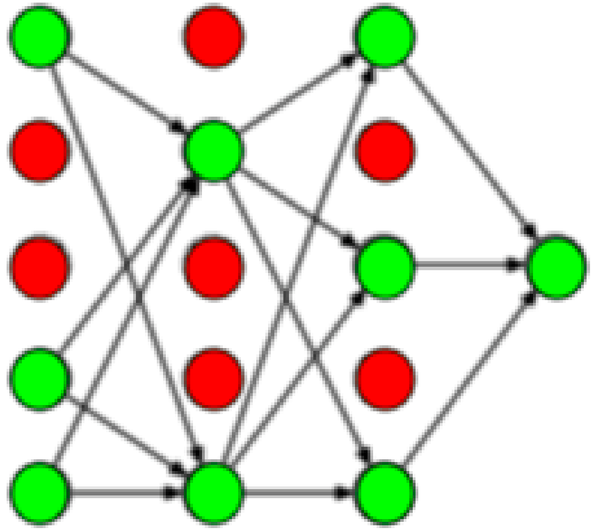


Neural Network with Dropout

- Improves generalization of the model
- Useful in Wider Networks to avoid overfitting

Dropout: Solution to Overfitting

- Implemented layer-wise in a neural network
- Implemented on any or all hidden layers in the network, as well as the input layer, but not on the output layer
- A hyperparameter specifies the probability at which outputs of a layer are dropped out, typically
  - 0.5 for hidden layers
  - 0.2 or less for input layer
- Can be used with most types of layers, such as dense fully connected layers, convolutional layers, and recurrent layers
- During testing, there is no drop-out of units



Neural Network with Dropout

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I. and Salakhutdinov, R..“Dropout: A Simple Way to Prevent Neural Networks from Overfitting”, 2014
https://www.cs.toronto.edu/~hinton/absps/JMLRdropout.pdf

Summary

- Training of Deep neural networks means determining the optimal set of parameters
- Gradient Descent and Backpropagation are used for optimizing weights in directed neural networks
- Learning rate is a hyperparameter, which should be neither too high nor too low
- Generalization means the ability of the model to perform well on new data
- Overfitting reduces the generalization ability of the model and degrades its performance
- Dropout in deep neural networks can help avoid overfitting

References

1. Machine Learning: A Probabilistic Perspective" by Kevin Murphy, published by MIT Press, 2012.
2. "Pattern Recognition and Machine Learning" by Christopher M. Bishop, published by Springer, 2006.
3. "Python Machine Learning" by Sebastian Raschka and Vahid Mirjalili, published by Packt Publishing, 2015.
4. "Machine Learning Yearning" by Andrew Ng, published by Goodfellow Publishers, 2018.
5. "Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow" by Aurélien Géron, published by O'Reilly Media, 2019.
6. "Applied Predictive Modeling" by Max Kuhn and Kjell Johnson, published by Springer, 2013.
7. "Reinforcement Learning: An Introduction" by Richard S. Sutton and Andrew G. Barto, published by MIT Press in 2018.

Thank You !

Any Questions