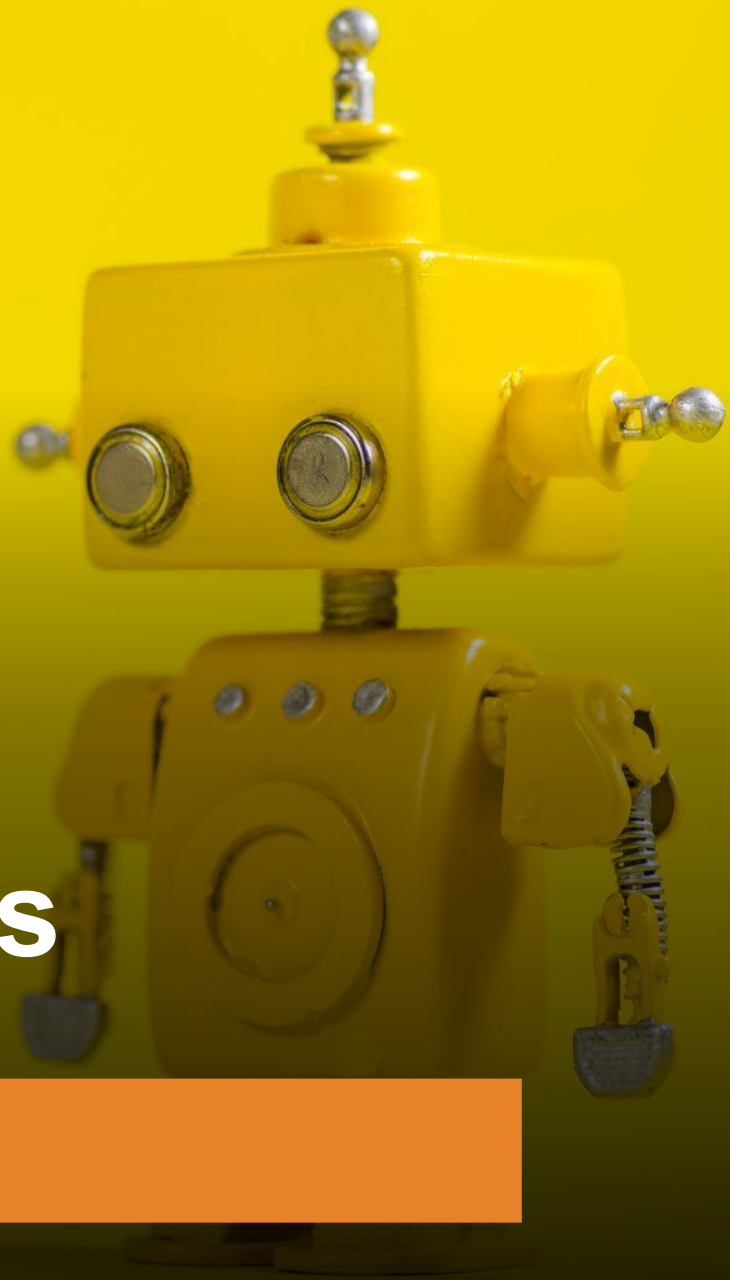
Introduction to Artificial Neural Networks

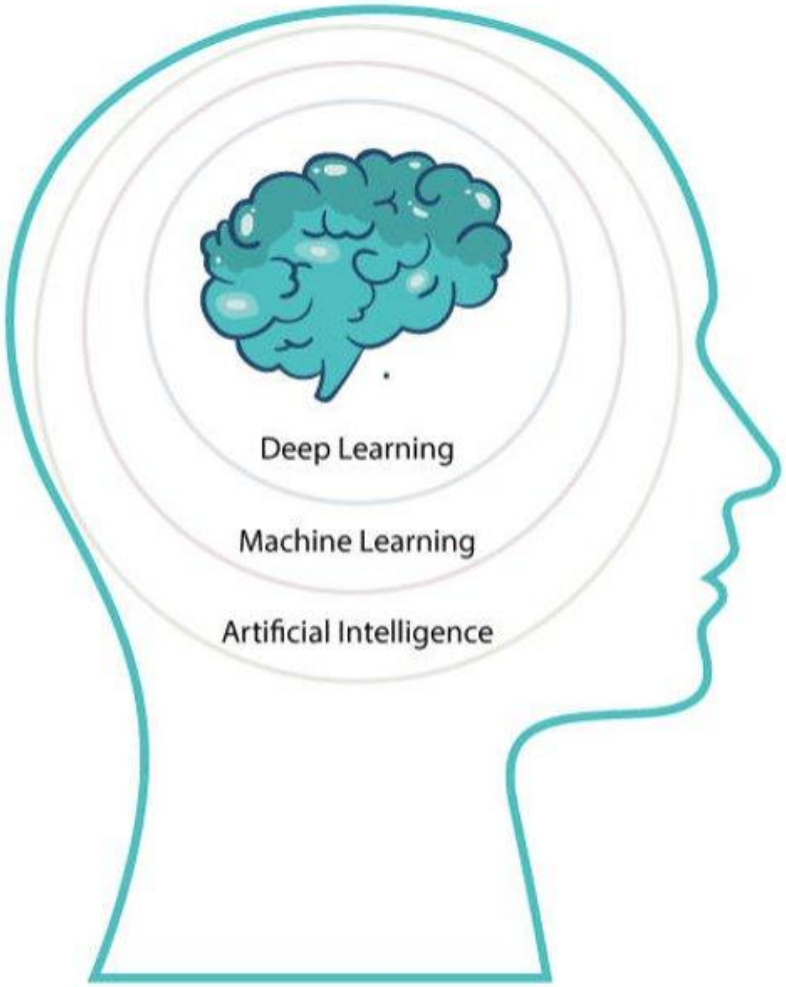


Nagur Shareef Shaik

What will you learn in these Sessions…?

- Introduction to Deep Learning
- Fundamentals of Artificial Neural Networks
- Feed Forward Neural Networks
  - Multi Layered Perceptron
  - Convolutional Neural Network
- Feedback Neural Networks
  - Recurrent Neural Networks (LSTM & GRU)
- Transfer Learning
- Transformers for Vision & Sequence Modelling

Introduction to Deep Learning



Deep Learning

- Deep Learning is a type of Machine Learning that is inspired by the structure of the brain for learning **representations** of data.
- Exceptional effective at **learning patterns**.
- Deep learning algorithms attempt to learn (**multiple levels of**) representation by using a **hierarchy of multiple layers.**
- If you provide the system **tons of information**, it begins to understand it and respond in useful ways.
- Involves networks which are capable of learning from data and functions similar to the human brain.

Why Deep Learning...?

- **Processes massive amount of data**
  - Deep Learning can process an enormous amount of both Structured and Unstructured data.
- **Performs Complex Operations**
  - Deep Learning algorithms are capable enough to perform complex operations when compared to the Machine Learning algorithms.
- **Achieves Best Performance**
  - As the amount of data increases, the performance of Machine Learning algorithms decreases.
  - On the other hand, Deep Learning maintains the performance of the model.
- **Feature Extraction**
  - Machine Learning algorithms extract patterns from labeled sample data, while Deep Learning algorithms take large volumes of data as input, analyze them to extract the features on its own.

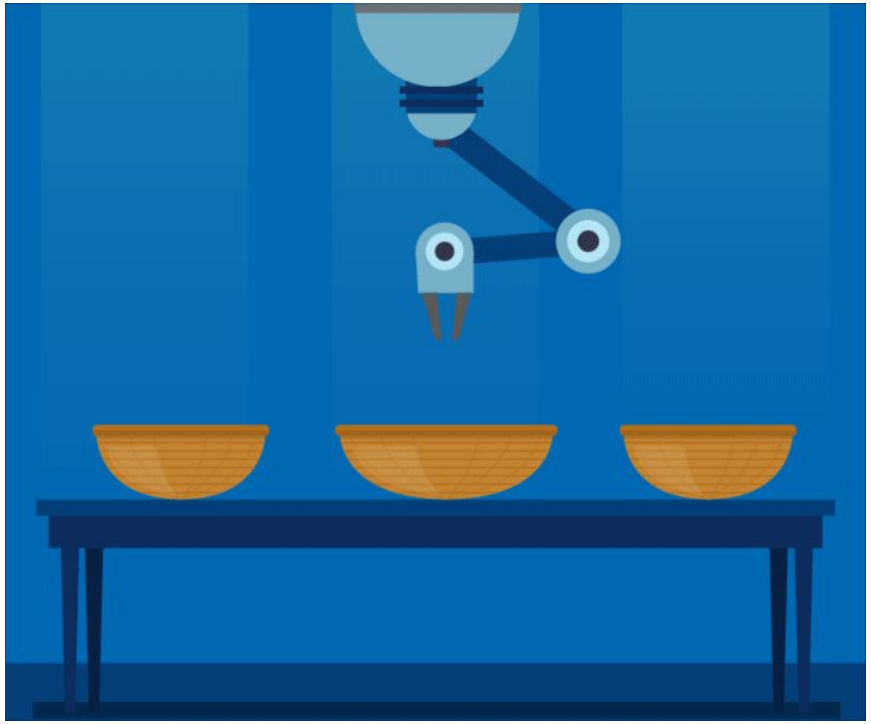
ML vs DL

Say, for instance, we develop a machine that differentiates between *cherries and tomatoes*.

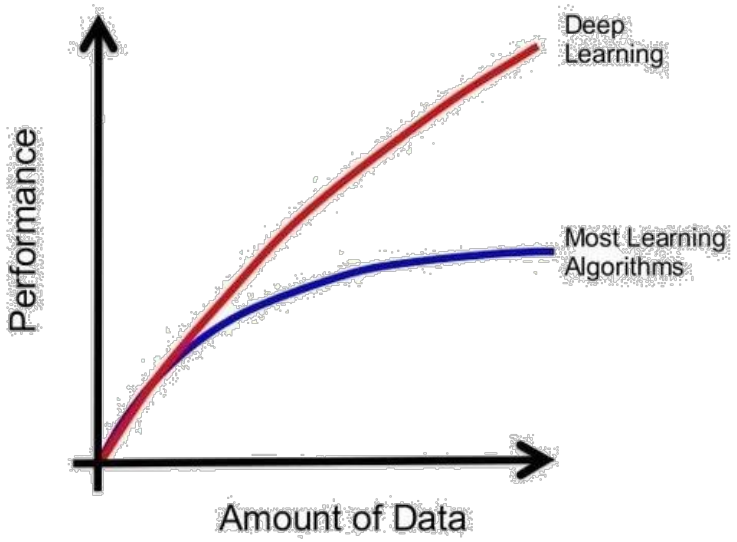
Machine Learning

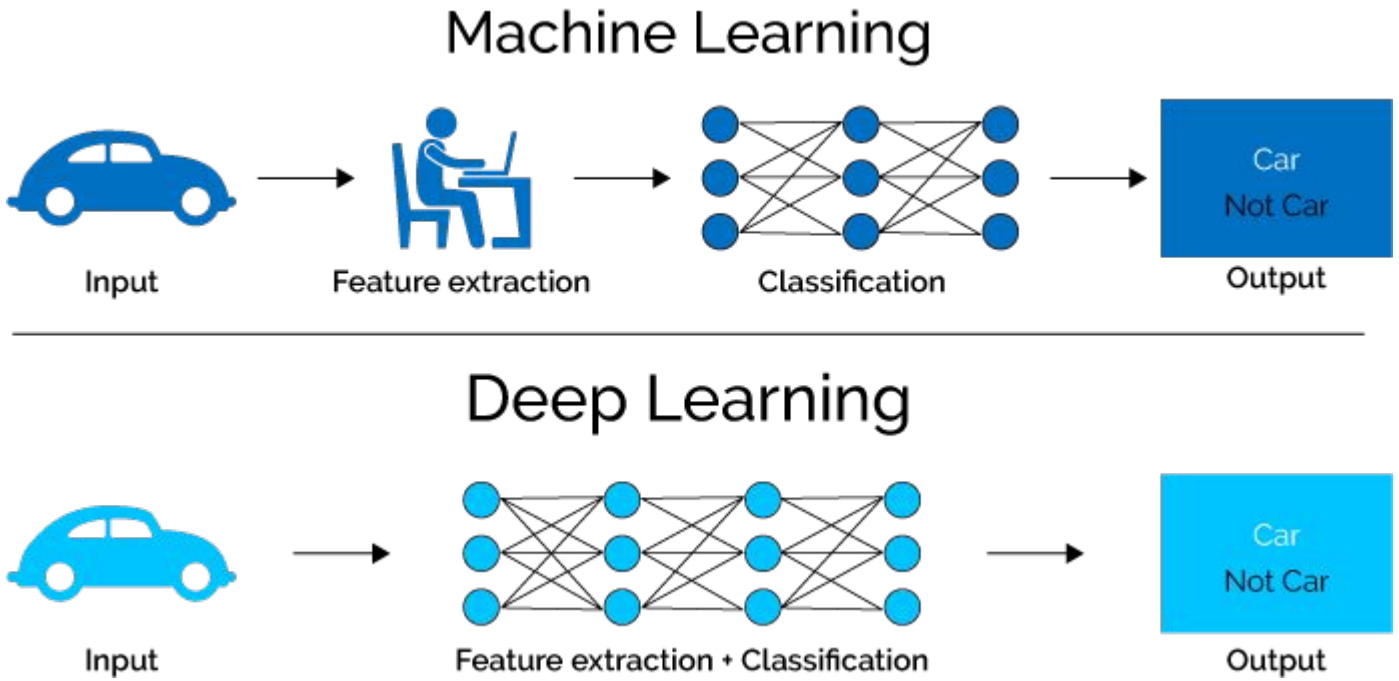
- ✔ If done through Machine Learning, ***we need to specify the features*** based on which the two can be differentiated like size and stem, in this case.

- **Deep Learning**
  - ✔ In Deep Learning, the ***features are picked by the Neural Network*** without any human intervention. But, that kind of independence can be achieved by a higher volume of data in training the machine.



ML vs DL


Deep
Learning
Most Learning
Algorithms
Performance
Amount of Data


Machine Learning
Input
Feature extraction
Classification
Car
Not Car
Output
Deep Learning
Input
Feature extraction + Classification
Car
Not Car
Output

Introduction to ANNs

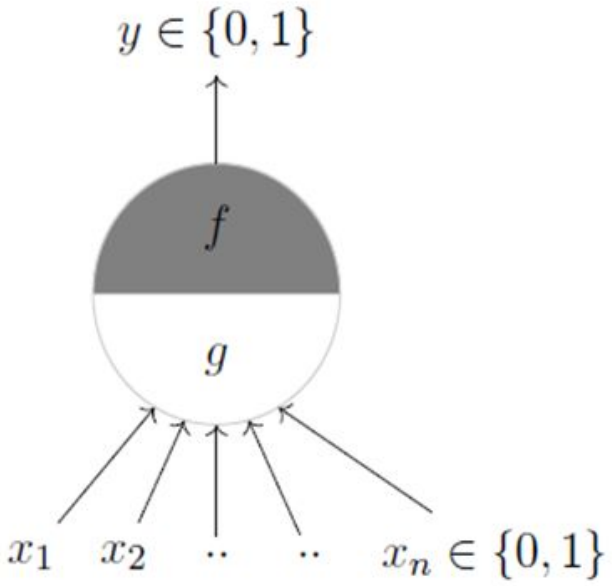
Evolution of Artificial Neural Networks

MP Neuron to Transformers (1943 to 2021)

- MP Neuron
- Perceptron
- Multi Layered Perceptron
- Boltzmann Machine
- Restricted Boltzmann Machine
- Deep Belief Networks
- Deep Neural Network
- Convolutional Neural Network
- Recurrent Neural Network
- Long Short Term Memory
- Gated Recurrent Unit
- Auto-encoder
- Generative Adversarial Networks
- Attention Mechanism
- Transformer
- Visual Transformer

Nagur Shareef Shaik

McCulloch and Pitts (MP) Neuron



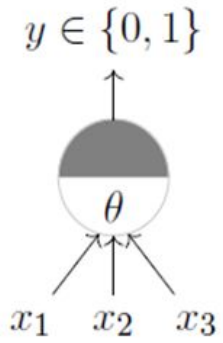
- McCulloch (neuroscientist) and Pitts (logician) proposed a highly simplified computational model of the neuron (1943)
- g aggregates the inputs and the function f takes a decision based on this aggregation
- The inputs can be excitatory or inhibitory
- $y = 0$ if any x_i is inhibitory, else

$$g(x_1, x_2, ..., x_n) = g(\mathbf{x}) = \sum_{i=1}^{n} x_i$$

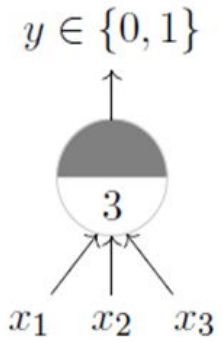
$$y = f(g(\mathbf{x})) = 1 \quad if \quad g(\mathbf{x}) \geq \theta$$
$$= 0 \quad if \quad g(\mathbf{x}) < \theta$$

- θ is called the thresholding parameter
- This is called Thresholding Logic

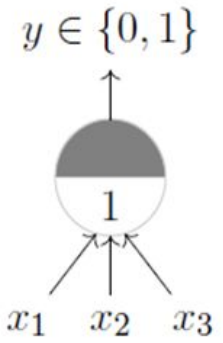
Realizing Boolean Logics using MP Neuron



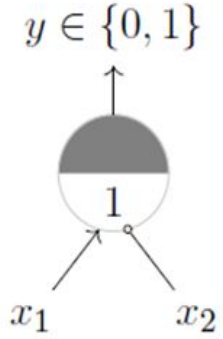
A McCulloch Pitts unit



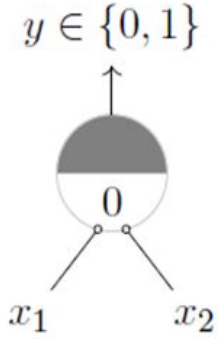
AND function



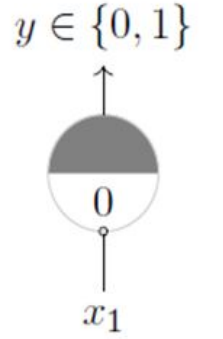
OR function



x_1 AND $!x_2$*



NOR function



NOT function

Implementing MP Neuron in Python

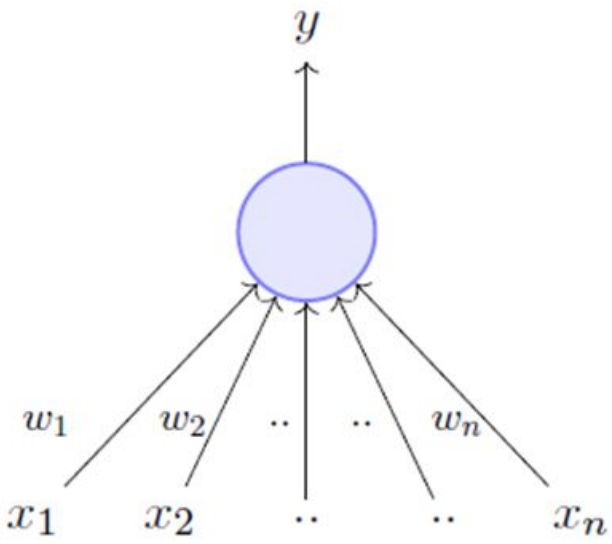
Conclusion

A single McCulloch Pitts Neuron can be used to represent Boolean functions which are linearly separable

Problems with MP Neuron...

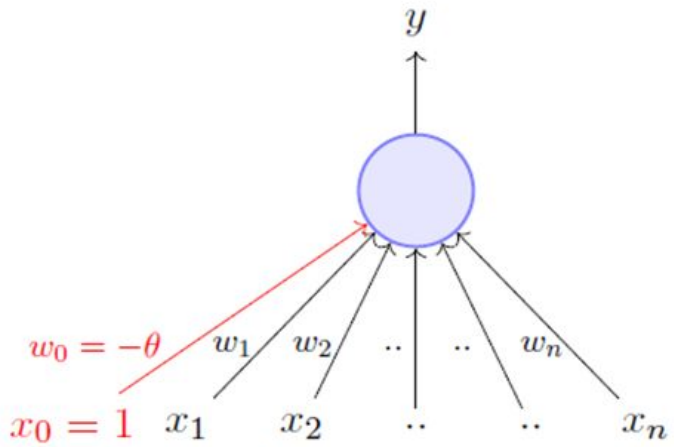
1. What about non Boolean (say, real) inputs ?
2. Do we always need to hand code the threshold ?
3. Are all inputs equal ? What if we want to assign more weight (importance) to some inputs ?
4. What about functions which are not linearly separable ?

Perceptron



- Frank Rosenblatt, an American psychologist, proposed the **classical perceptron** model (1958)
- A more general computational model than McCulloch–Pitts neurons
- **Main differences:** Introduction of numerical weights for inputs and a mechanism for learning these weights
- Inputs are no longer limited to boolean values
- Refined and carefully analyzed by Minsky and Papert (1969) - their model is referred to as the **perceptron** model here

Perceptron



A more accepted convention,

$$y = 1 \quad if \sum_{i=0}^{n} w_i * x_i \geq 0$$

$$= 0 \quad if \sum_{i=0}^{n} w_i * x_i < 0$$

$where, \quad x_0 = 1 \quad and \quad w_0 = -\theta$

$$y = 1 \quad if \sum_{i=1}^{n} w_i * x_i \geq \theta$$

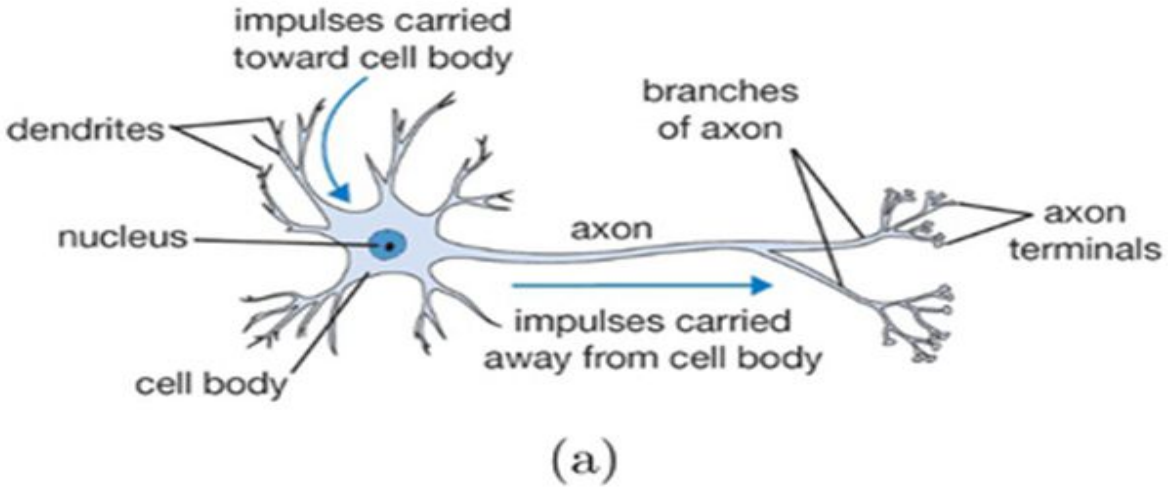
$$= 0 \quad if \sum_{i=1}^{n} w_i * x_i < \theta$$

Rewriting the above,

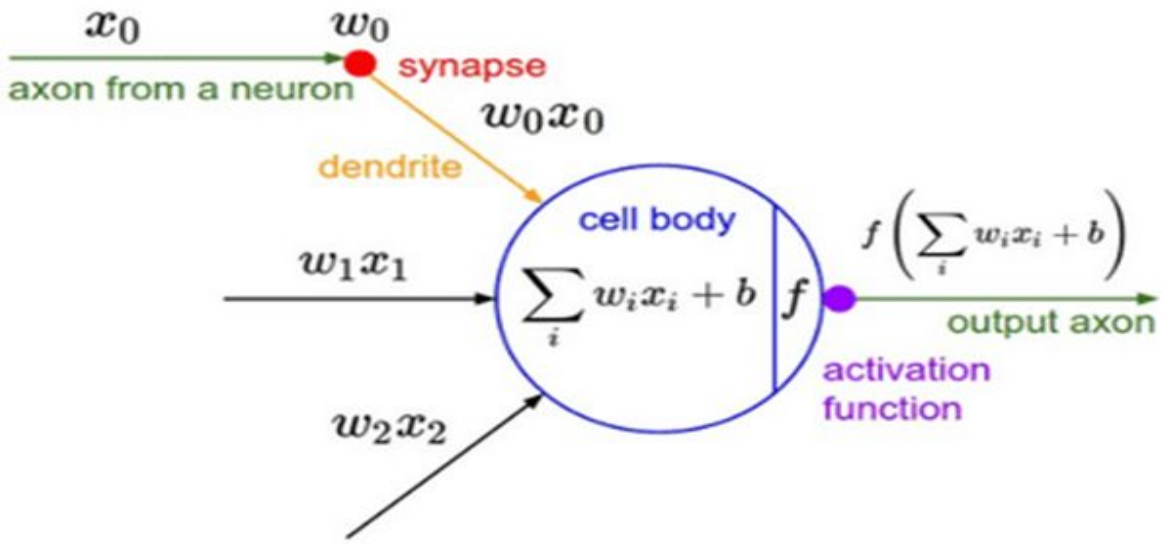
$$y = 1 \quad if \sum_{i=1}^{n} w_i * x_i - \theta \geq 0$$

$$= 0 \quad if \sum_{i=1}^{n} w_i * x_i - \theta < 0$$

Biological vs Artificial Neurons



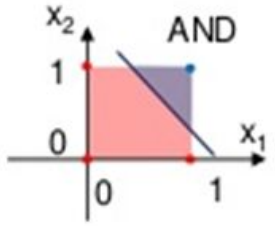
(a)

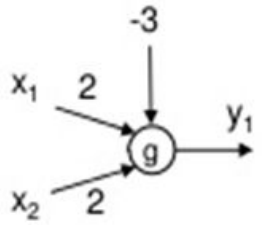


(b)

AND, OR Logic with Perceptron

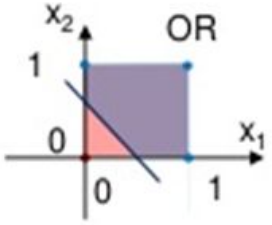
Input vector (x1,x2)	Class AND
(0,0)	0
(0,1)	0
(1,0)	0
(1,1)	1

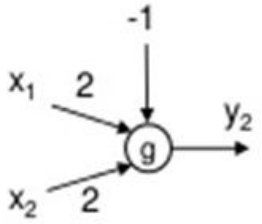




$$y_1 = g(\mathbf{w}^T\mathbf{x} + b) = u((2 \quad 2) \cdot \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} - 3)$$

Input vector (x1,x2)	Class OR
(0,0)	0
(0,1)	1
(1,0)	1
(1,1)	1





$$y_2 = g(\mathbf{w}^T\mathbf{x} + b) = u((2 \quad 2) \cdot \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} - 1)$$

Perceptron Learning Algorithm

Algorithm: Perceptron Learning Algorithm

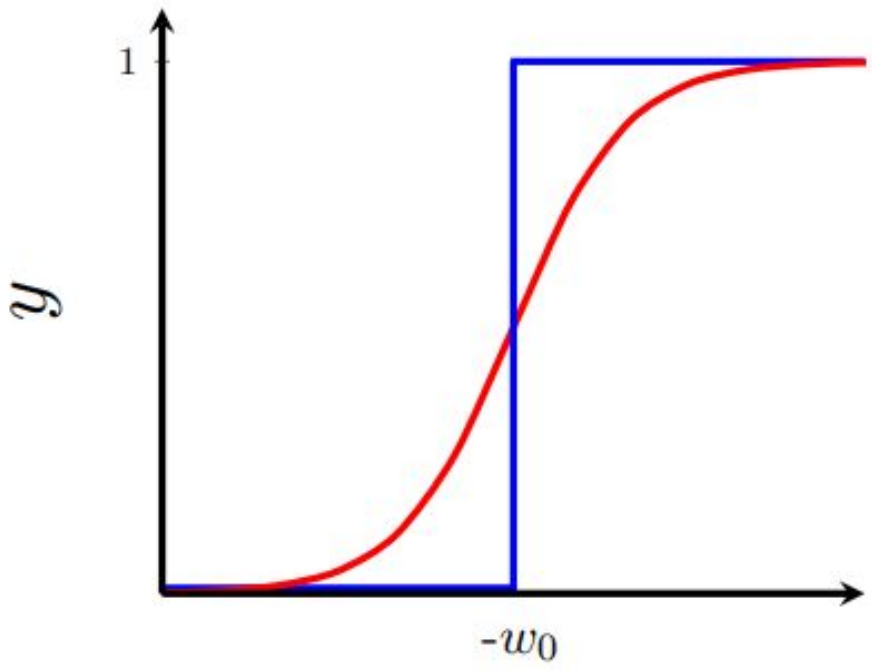
$P \leftarrow inputs \quad with \quad label \quad 1;$
$N \leftarrow inputs \quad with \quad label \quad 0;$
Initialize $\mathbf{w}$ randomly;
while !*convergence* **do**
    Pick random $\mathbf{x} \in P \cup N$;
    if $\mathbf{x} \in P \quad and \quad \mathbf{w}.\mathbf{x} < 0$ **then**
        $\mathbf{w} = \mathbf{w} + \mathbf{x}$;
    end
    if $\mathbf{x} \in N \quad and \quad \mathbf{w}.\mathbf{x} \geq 0$ **then**
        $\mathbf{w} = \mathbf{w} - \mathbf{x}$;
    end
end
//the algorithm converges when all the inputs are classified correctly

Implementing Perceptron in Python

Intuition to Sigmoid Neuron

- Enough about boolean functions!
- What about arbitrary functions of the form $y = f(x)$ where $x \in \mathbb{R}^n$ (instead of $\{0,1\}^n$) and $y \in \mathbb{R}$ (instead of $\{0,1\}$) ?
- Can we have a network which can (approximately) represent such functions ?
- Before answering the above question we will have to first graduate from ***perceptrons*** to ***sigmoidal neurons*** ...

Sigmoid Neuron



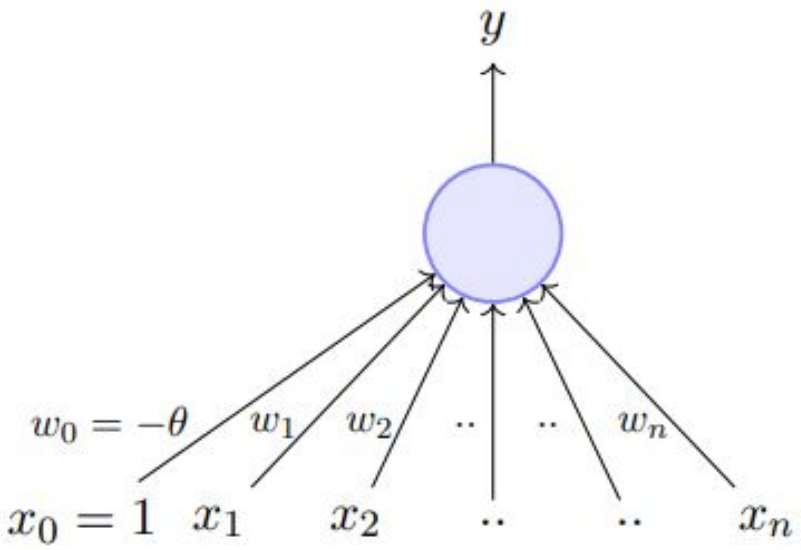
- Introducing sigmoid neurons where the output function is much smoother than the step function
- Here is one form of the sigmoid function called the logistic function

$$y = \frac{1}{1 + e^{-(w_0 + \sum_{i=1}^{n} w_i x_i)}}$$

- We no longer see a sharp transition around the threshold -w_0
- Also the output y is no longer binary but a real value between 0 and 1 which can be interpreted as a probability
- Instead of a like/dislike decision we get the probability of liking the movie

Perceptron vs Sigmoid Neuron

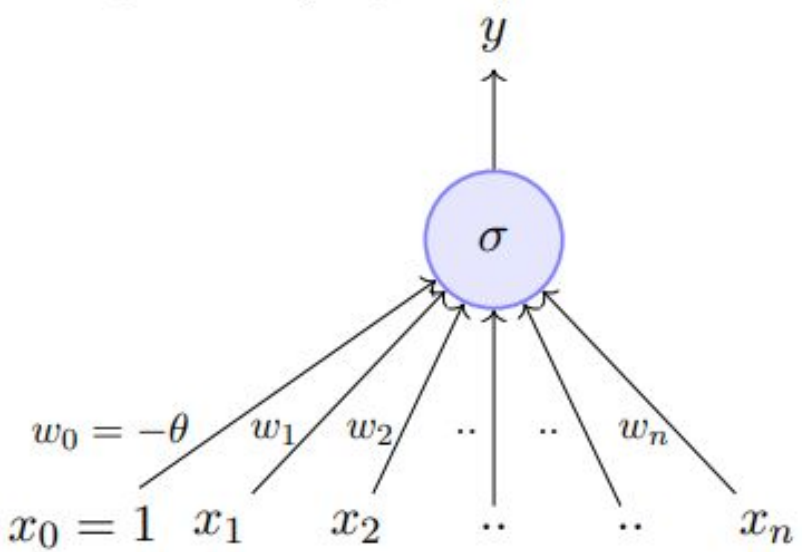
Perceptron



$$y = 1 \quad if \sum_{i=0}^{n} w_i * x_i \geq 0$$

$$= 0 \quad if \sum_{i=0}^{n} w_i * x_i < 0$$

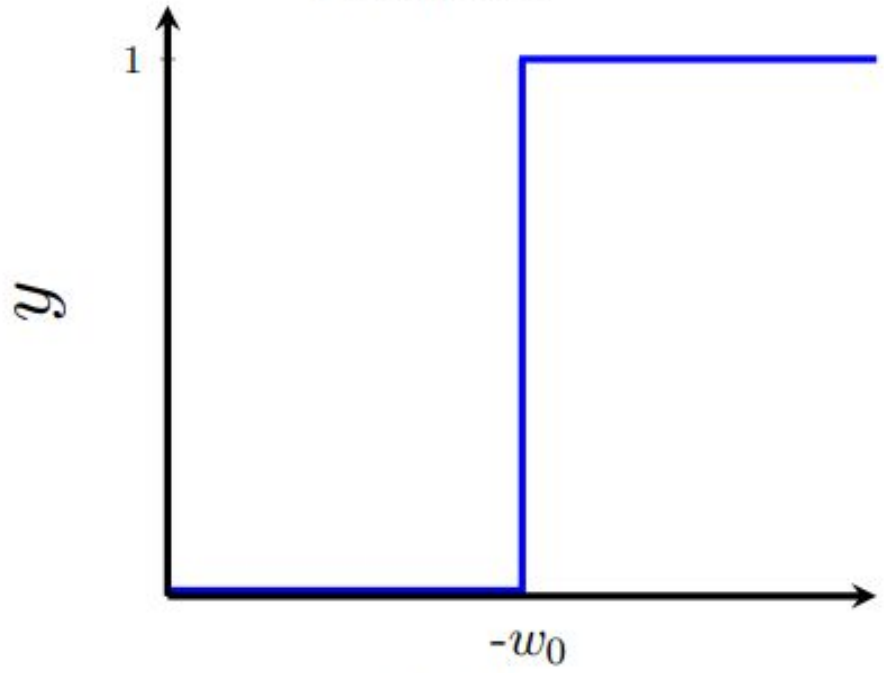
Sigmoid (logistic) Neuron



$$y = \frac{1}{1 + e^{-(\sum_{i=0}^{n} w_i x_i)}}$$

Perceptron vs Sigmoid Neuron

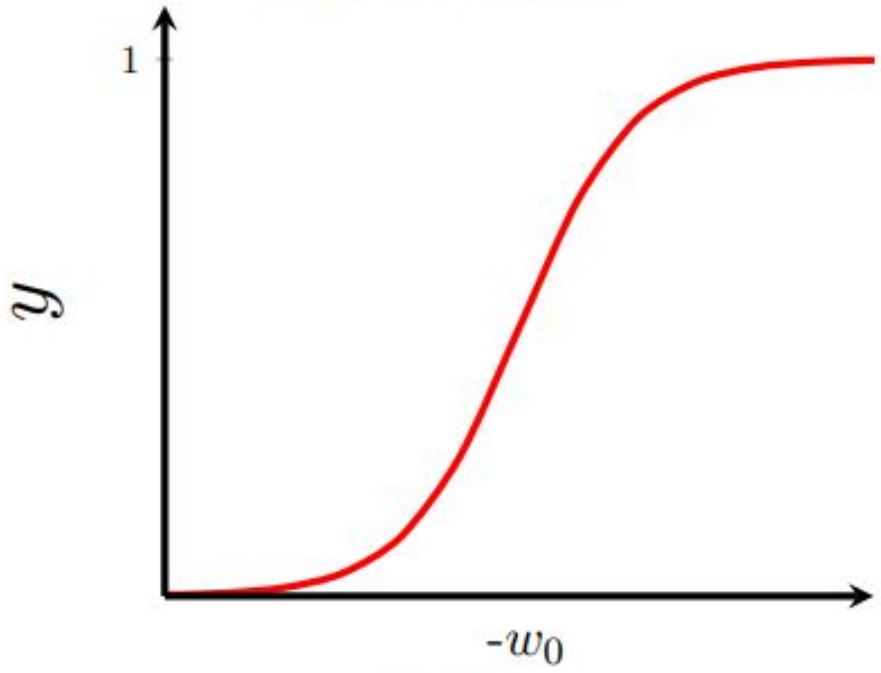
Perceptron



$$z = \sum_{i=1}^{n} w_i x_i$$

Not smooth, not continuous (at $w0$), **not differentiable**

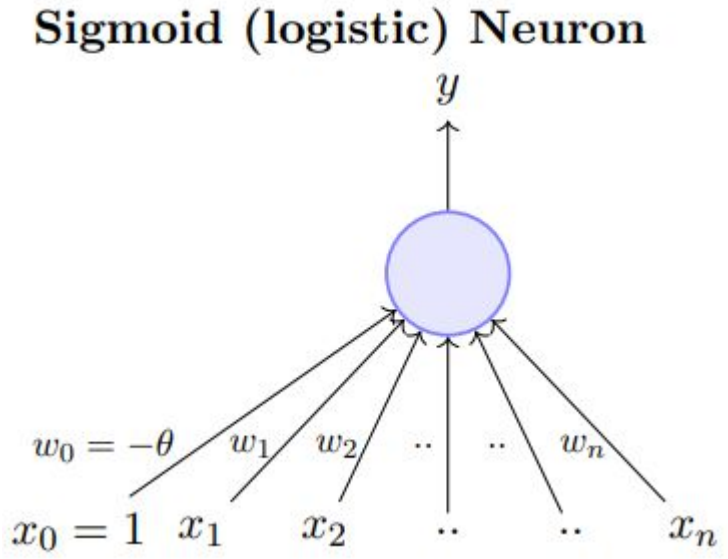
Sigmoid Neuron



$$z = \sum_{i=1}^{n} w_i x_i$$

Smooth, continuous, **differentiable**

Learning Algorithm for Sigmoid Neuron



- Well, just as we had an algorithm for learning the weights of a perceptron, we also need a way of learning the weights of a sigmoid neuron

Learning Setup

This brings us to a typical machine learning setup which has the following components...

- **Data:** $\{x_i, y_i\}_{i=1}^n$
- **Model:** Our approximation of the relation between $\mathbf{x}$ and y. For example,

$$\hat{y} = \frac{1}{1 + e^{-(\mathbf{w^T x})}}$$

$$or \quad \hat{y} = \mathbf{w^T x}$$

$$or \quad \hat{y} = \mathbf{x^T W x}$$

  or just about any function

- **Parameters:** In all the above cases, w is a parameter which needs to be learned from the data
- **Learning algorithm:** An algorithm for learning the parameters (w) of the model (for example, perceptron learning algorithm, gradient descent, etc.)
- **Objective/Loss/Error function:** To guide the learning algorithm - the learning algorithm should aim to minimize the loss function

Gradient Descent for Learning Sigmoid Neuron Params

Gradient Descent Rule

- The direction u that we intend to move in should be at 180° w.r.t. the gradient
- In other words, move in a direction opposite to the gradient

Parameter Update Equations

$$w_{t+1} = w_t - \eta \nabla w_t$$

$$b_{t+1} = b_t - \eta \nabla b_t$$

$$where, \nabla w_t = \frac{\partial \mathscr{L}(w,b)}{\partial w}_{at\ w=w_t, b=b_t}, \nabla b = \frac{\partial \mathscr{L}(w,b)}{\partial b}_{at\ w=w_t, b=b_t}$$

Gradient Descent Algorithm

Algorithm: gradient_descent()

$t \leftarrow 0$;
$max\_iterations \leftarrow 1000$;
while $t < max\_iterations$ **do**
  $w_{t+1} \leftarrow w_t - \eta \nabla w_t$;
  $b_{t+1} \leftarrow b_t - \eta \nabla b_t$;
  $t \leftarrow t + 1$;
end

$$
\begin{aligned}
\nabla w &= \frac{\partial}{\partial w}[\frac{1}{2} * (f(x) - y)^2] \\
&= \frac{1}{2} * [2 * (f(x) - y) * \frac{\partial}{\partial w}(f(x) - y)] \\
&= (f(x) - y) * \frac{\partial}{\partial w}(f(x)) \\
&= (f(x) - y) * \frac{\partial}{\partial w}\left(\frac{1}{1 + e^{-(wx+b)}}\right) \\
&= (f(x) - y) * f(x) * (1 - f(x)) * x
\end{aligned}
$$

$$
\begin{aligned}
&\frac{\partial}{\partial w}\left(\frac{1}{1 + e^{-(wx+b)}}\right) \\
&= \frac{-1}{(1 + e^{-(wx+b)})^2} \frac{\partial}{\partial w}(e^{-(wx+b)})) \\
&= \frac{-1}{(1 + e^{-(wx+b)})^2} * (e^{-(wx+b)}) \frac{\partial}{\partial w}(-(wx + b))) \\
&= \frac{-1}{(1 + e^{-(wx+b)})} * \frac{e^{-(wx+b)}}{(1 + e^{-(wx+b)})} * (-x) \\
&= \frac{1}{(1 + e^{-(wx+b)})} * \frac{e^{-(wx+b)}}{(1 + e^{-(wx+b)})} * (x) \\
&= f(x) * (1 - f(x)) * x
\end{aligned}
$$

So if there is only 1 point (x, y), we have,

$$\nabla w = (f(x) - y) * f(x) * (1 - f(x)) * x$$

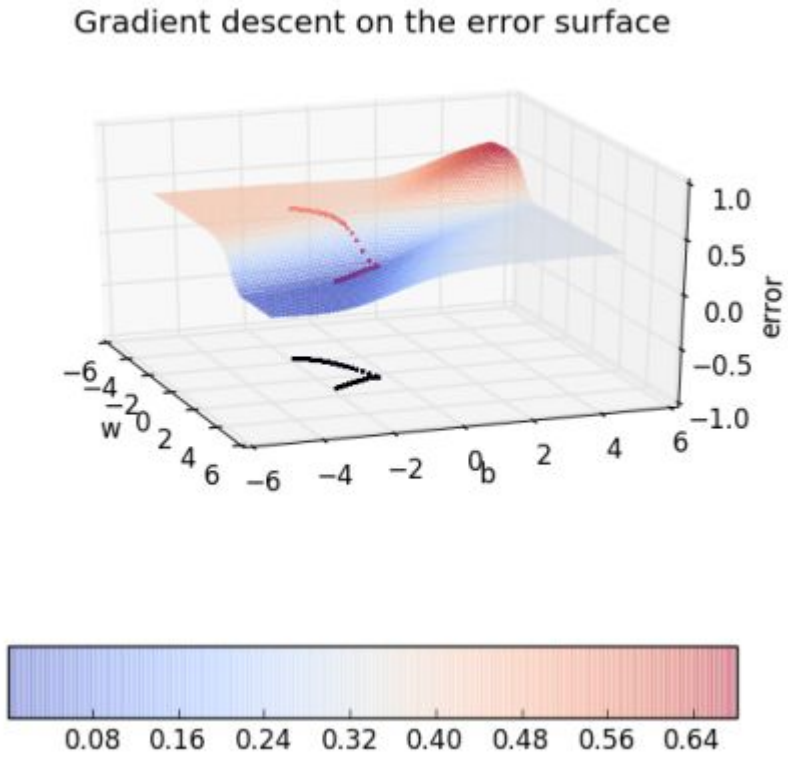
For two points,

$$\nabla w = \sum_{i=1}^{2} (f(x_i) - y_i) * f(x_i) * (1 - f(x_i)) * x_i$$

$$\nabla b = \sum_{i=1}^{2} (f(x_i) - y_i) * f(x_i) * (1 - f(x_i))$$

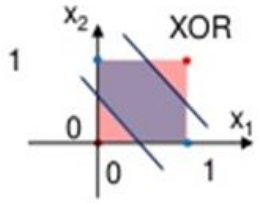
Implementing Gradient Descent

```
X = [0.5, 2.5]
Y = [0.2, 0.9]

def f(w,b,x) : #sigmoid with parameters w,b
    return 1.0 / (1.0 + np.exp(-(w*x + b)))

def error (w, b) :
    err = 0.0
    for x,y in zip(X,Y) :
        fx = f(w,b,x)
        err += 0.5 * (fx - y) ** 2
    return err

def grad_b(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx)

def grad_w(w,b,x,y) :
    fx = f(w,b,x)
    return (fx - y) * fx * (1 - fx) * x

def do_gradient_descent() :
    w, b, eta, max_epochs = -2, -2, 1.0, 1000
    for i in range(max_epochs) :
        dw, db = 0, 0
        for x,y in zip(X, Y) :
            dw += grad_w(w, b, x, y)
            db += grad_b(w, b, x, y)
        w = w - eta * dw
        b = b - eta * db
```

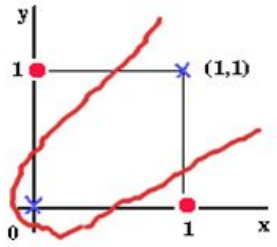


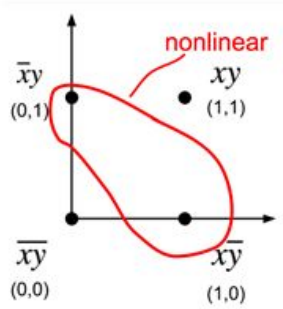
Perceptron / Sigmoid Neuron for XOR

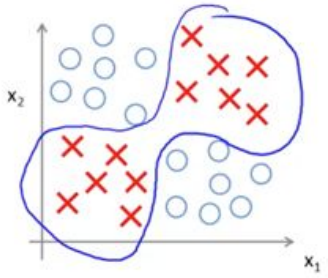
Input vector (x1,x2)	Class XOR
(0,0)	0
(0,1)	1
(1,0)	1
(1,1)	0



Find the weights of Perceptron for XOR

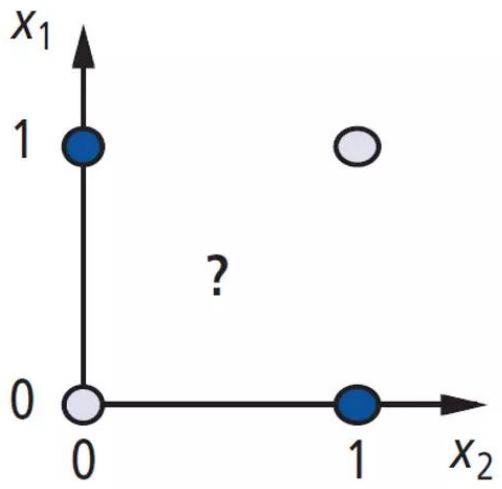


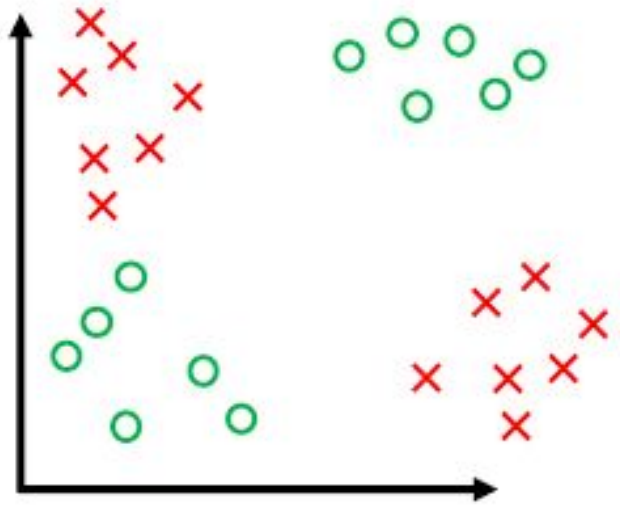




Perceptron – Failure

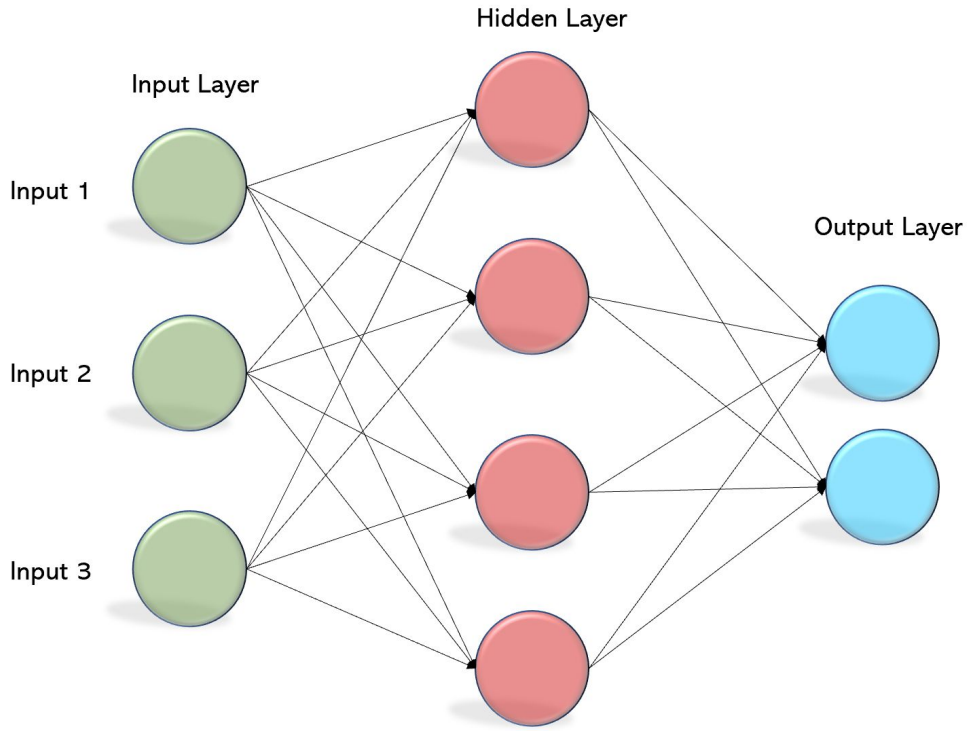
- A "single-layer" perceptron can't implement XOR.
- The reason is because the classes in XOR are not linearly separable.
- You cannot draw a straight line to separate the points (0,0),(1,1) from the points (0,1),(1,0).
- Led to invention of multi-layer networks.





Multi Layered Perceptron – MLP

- In the Multilayer perceptron, there can more than one linear layer (combinations of neurons).
- There are three different layers in a MLP / Deep Neural Network, namely:
  1. Input Layer
  2. Hidden Layer
  3. Output Layer



Structure of MLP

Input Layer:

- The input layer communicates with the external environment and presents a pattern to the neural network.
- Every neuron in the input layer represents an independent variable that influences the output.

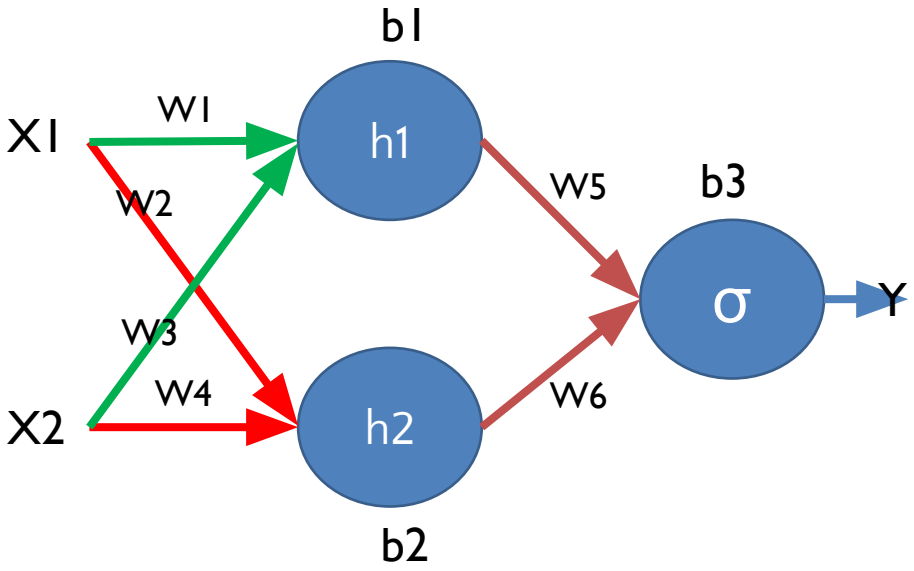
Hidden Layer:

- A Neural Network consists of several hidden layers, each consisting of a collection of neurons.
- The hidden layer is an intermediate layer found between the input layer and the output layer.
- This layer is responsible for extracting the features required from the input.
- There is no exact formula for calculating the number of the hidden layers as well as the number of neurons in each hidden layer.

Output Layer:

- The output layer of the neural network collects and transmits information in the desired format.

Learning XOR Problem



(X1, X2) = {(0,0),(1,1),(0,1),(1,0)}
Y = {0,0,1,1}
W1 = 20, W2 = -20, W3 = 20,
W4 = -20, W5 = 20, W6 = 20
b1 = -10, b2 = 30, b3 = -30

$\sigma(20x_1 + 20x_2 - 10)$

$\sigma(20*0 + 20*0 - 10) \approx 0$
$\sigma(20*1 + 20*1 - 10) \approx 1$
$\sigma(20*0 + 20*1 - 10) \approx 1$
$\sigma(20*1 + 20*0 - 10) \approx 1$

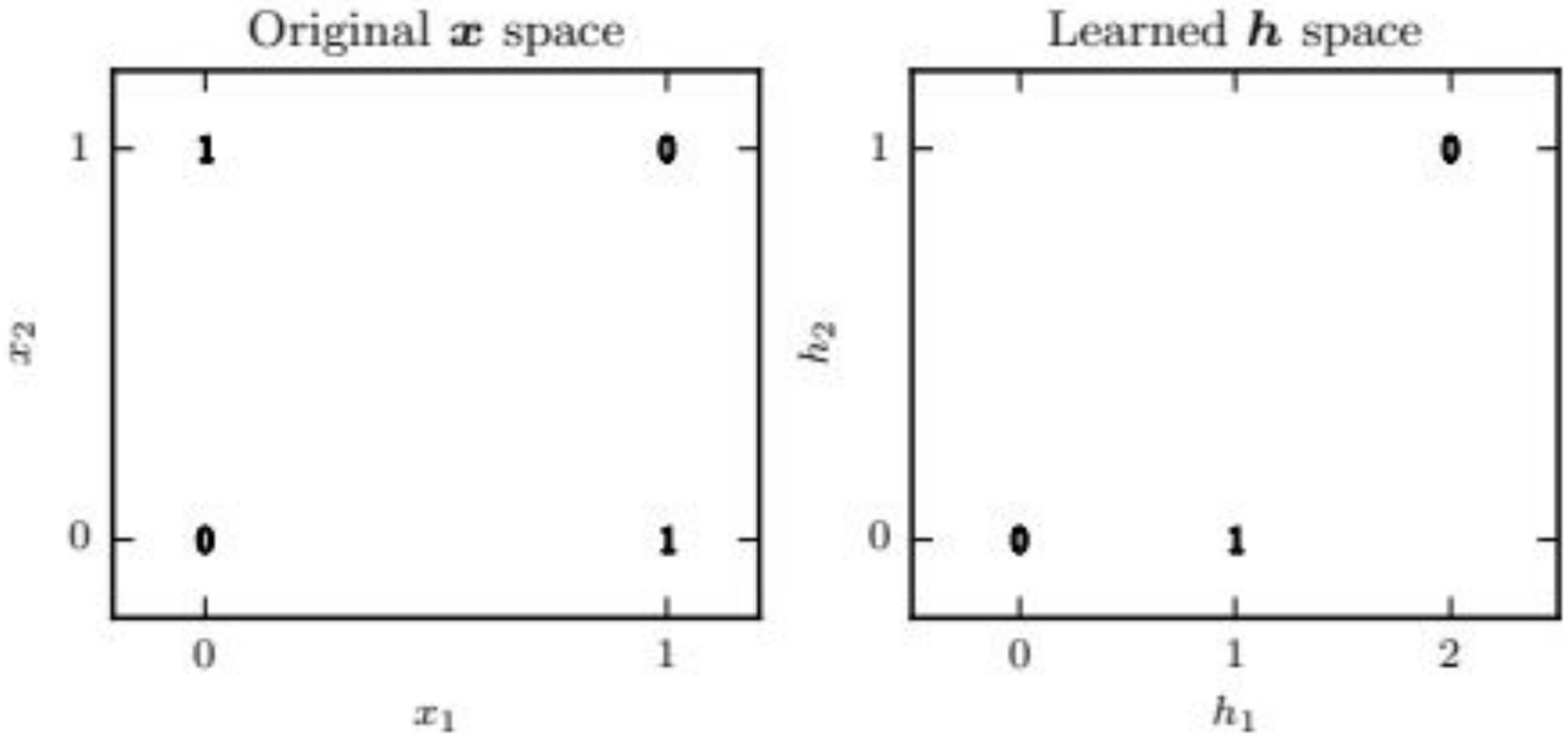
$\sigma(-20x_1 - 20x_2 + 30)$

$\sigma(-20*0 - 20*0 + 30) \approx 1$
$\sigma(-20*1 - 20*1 + 30) \approx 0$
$\sigma(-20*0 - 20*1 + 30) \approx 1$
$\sigma(-20*1 - 20*0 + 30) \approx 1$

$\sigma(20h_1 + 20h_2 - 30)$

$\sigma(20*0 + 20*1 - 30) \approx 0$
$\sigma(20*1 + 20*0 - 30) \approx 0$
$\sigma(20*1 + 20*1 - 30) \approx 1$
$\sigma(20*1 + 20*1 - 30) \approx 1$

Learning XOR Problem



Can you apply Perceptron Learning Rule for MLP…?

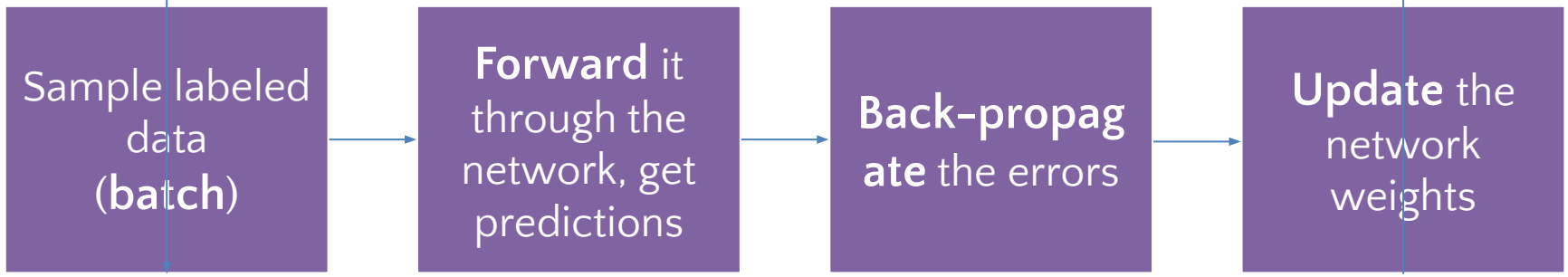
Back Propagation Learning Rule

- ***Training* of the *multilayer perceptron* proceeds in *two phases*:**
  - In the ***forward*** *phase*, the weights of the network are *fixed* and the input signal is *propagated* through the network, layer by layer, until it reaches the *output*.
  - In the ***backward*** *phase*, the error signal, which is produced by comparing the output of the network and the desired response, is *propagated* through the network, again layer by layer, but in the *backward direction*.

1. ***Initialization***
2. ***Presentation of training example***
3. ***Forward computation***
4. ***Backward computation***
5. ***Iteration***

Training Process in NN

- Optimize (min. or max.) **objective/cost function** $J(\theta)$
- Generate **error signal** that measures difference between predictions and target values
- Use error signal to change the **weights** and get more accurate predictions
- Subtracting a fraction of the **gradient** moves you towards the **(local) minimum of the cost function**

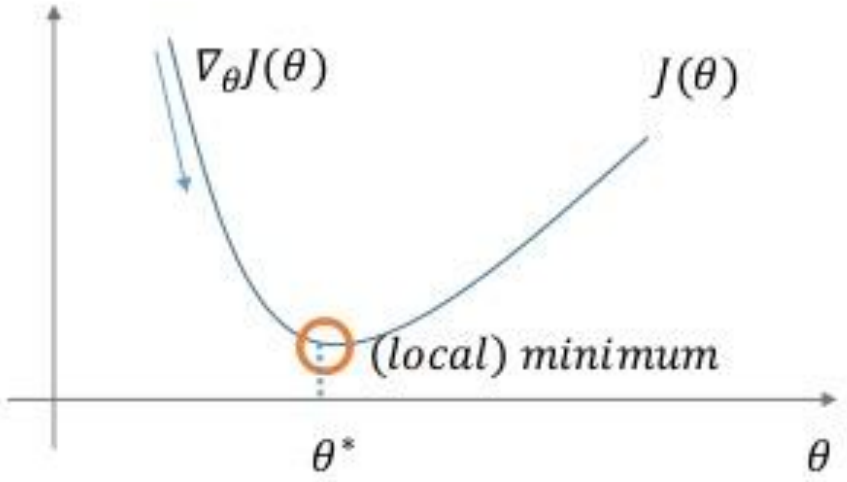


Gradient Descent – Optimization

- Gradient descent is a way to minimize an objective function $J(\theta)$
  - $J(\theta)$: Objective function
  - $\theta \in R^d$: Model's parameters
  - η : Learning rate. This determines the size of the steps we take to reach a (local) minimum.

Update equation

$$\theta = \theta - \eta * \nabla_\theta J(\theta)$$



Gradient Descent – Optimization

$$\theta_j = \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

Now,

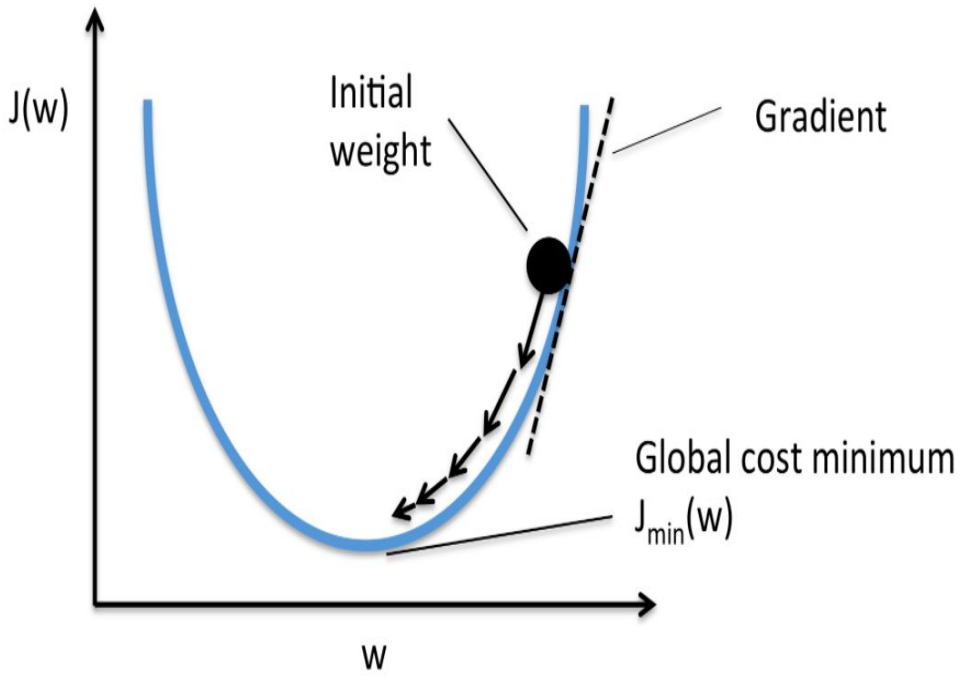
$$\frac{\partial}{\partial \theta} J_\theta = \frac{\partial}{\partial \theta} \frac{1}{2m} \sum_{i=1}^{m} \left[h_\theta(x_i) - y_i \right]^2$$

$$\frac{\partial}{\partial \theta} J_\theta = \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x_i) - y_i \right) \cdot \frac{\partial}{\partial \theta_j} \left(\theta x_i - y_i \right)$$

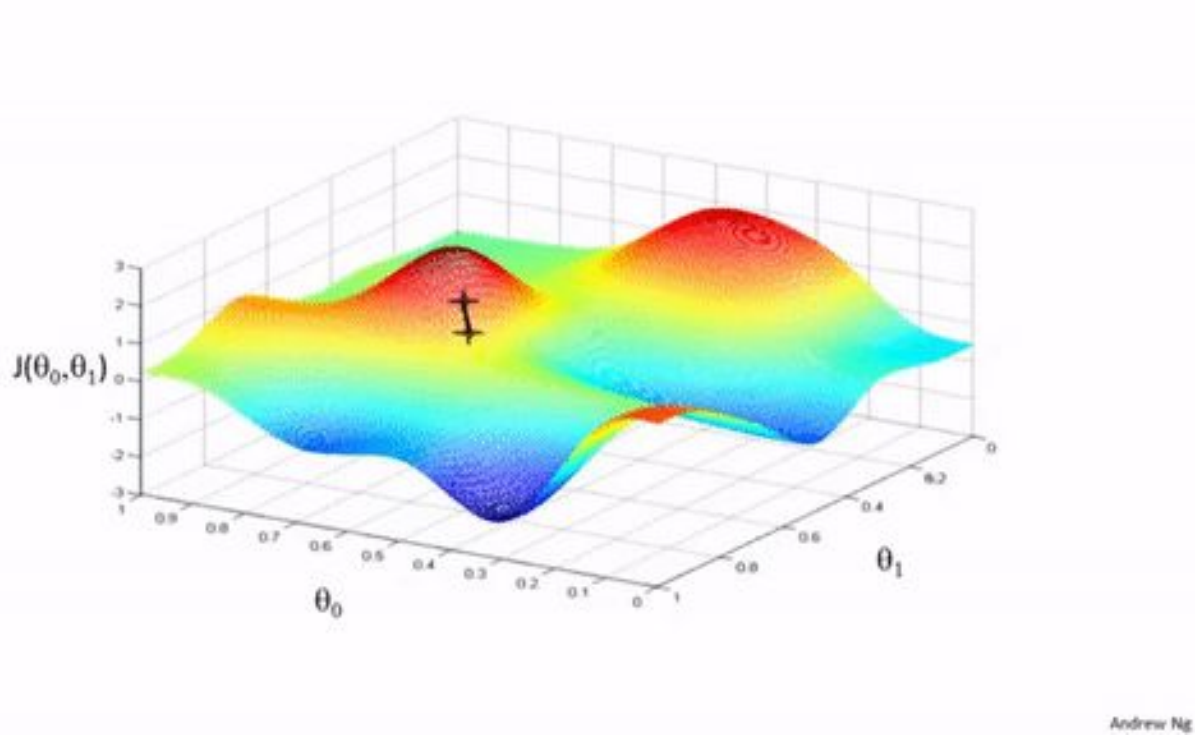
$$\frac{\partial}{\partial \theta} J_\theta = \frac{1}{m} \sum_{i=1}^{m} \left[\left(h_\theta(x_i) - y \right) x_i \right]$$

Therefore,

$$\theta_j := \theta_j - \frac{\alpha}{m} \sum_{i=1}^{m} \left[\left(h_\theta(x_i) - y_i \right) x_i \right]$$

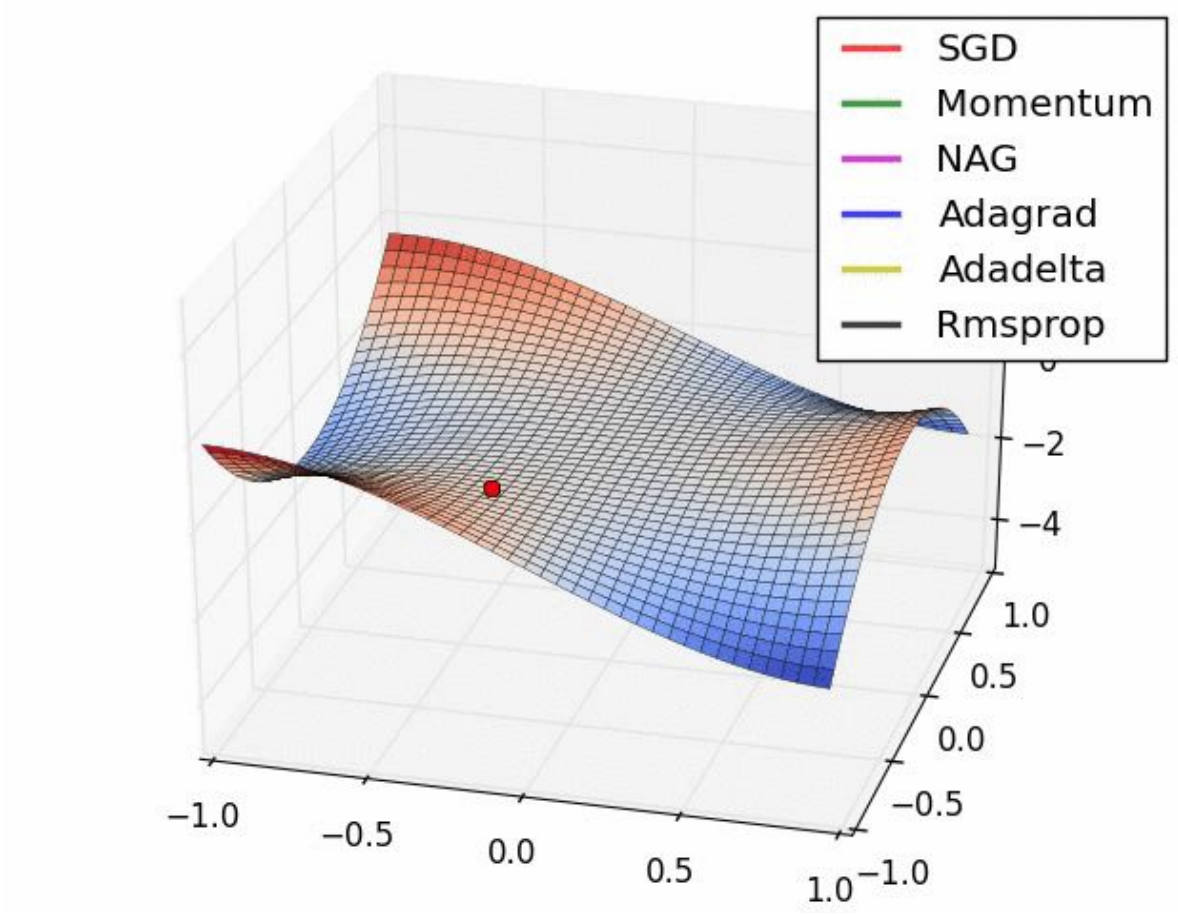


Error Surface



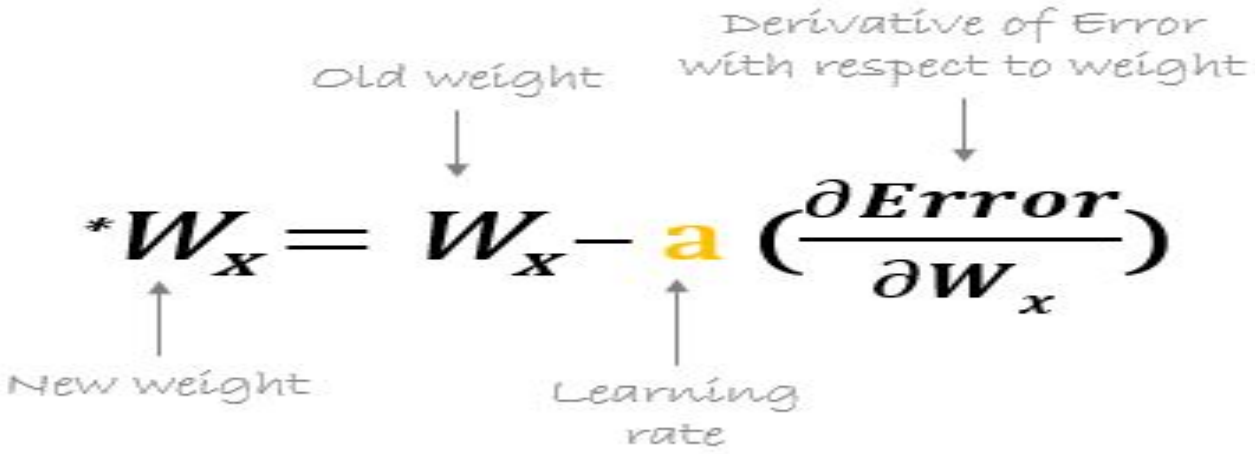
Andrew Ng

Optimizers

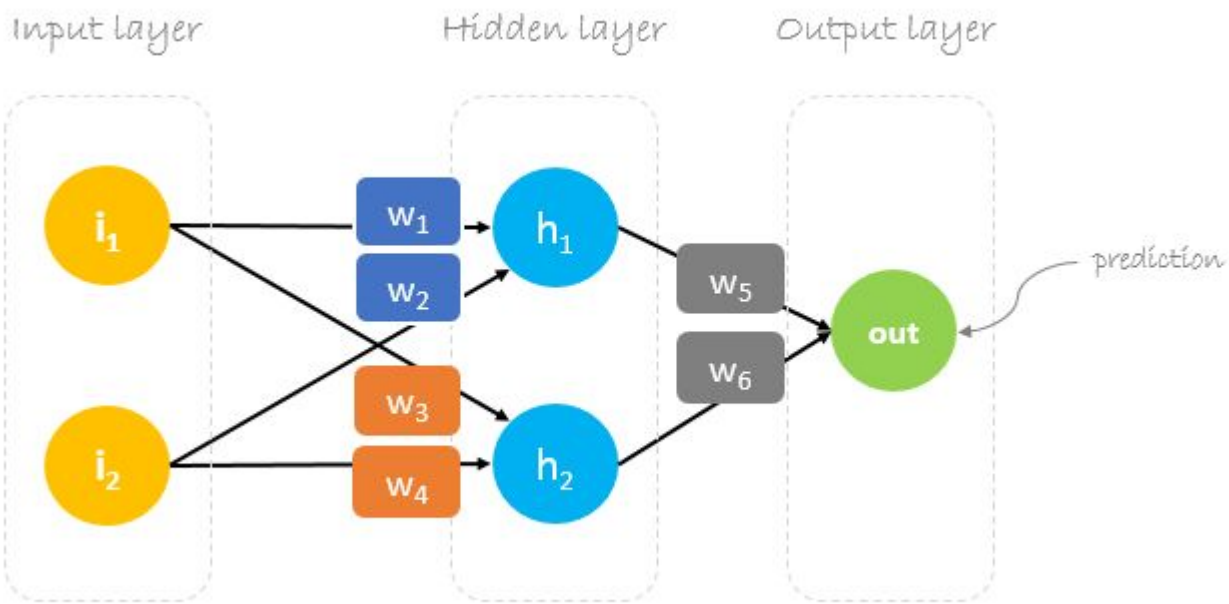

SGD
Momentum
NAG
Adagrad
Adadelta
Rmsprop
−2
−4
1.0
0.5
0.0
−0.5
−1.0
−1.0
−0.5
0.0
0.5
1.0

Back Propagation

- “Backward propagation of errors”, is a mechanism used to update the weights using gradient descent.
- Gradient descent is an iterative optimization algorithm to minimize the error function.
- The weights are updated in the negative direction of the gradient to find a local minimum of a function



Forward Propagation



$$\text{Out} = f((w_5 * h_1) + (w_6 * h_2))$$

$$\text{Out} = f((w_5 * g_2(w_1 * i_1 + w_2 * i_2) + (w_6 * g_2(w_3 * i_1 + w_4 * i_2))$$

What can be varied to minimize error?

Forward Pass – Example

Our Single Sample:

Inputs = [2, 3]

Output = [1]

Our Initial Weights:

w1 = 0.11

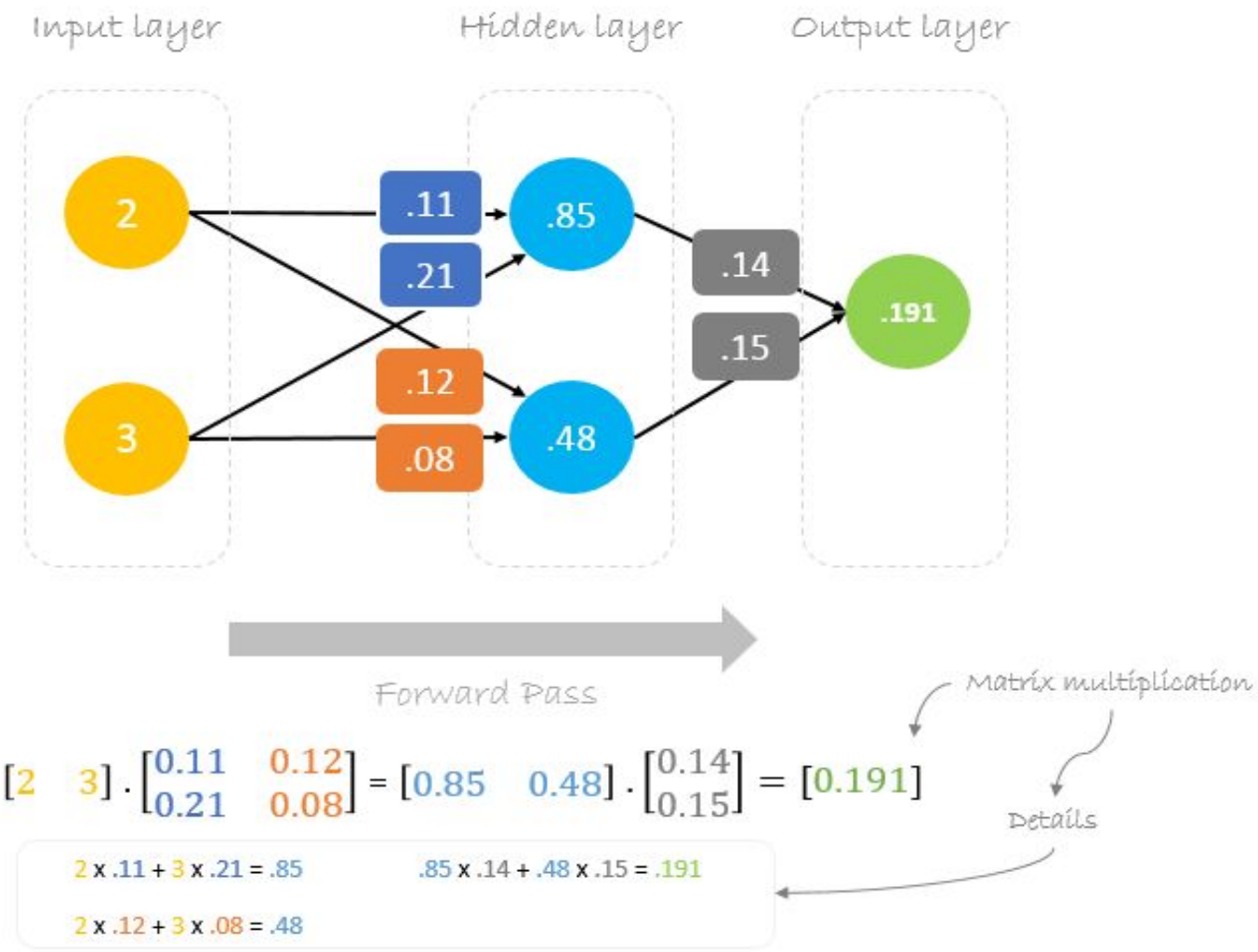
w2 = 0.21

w3 = 0.12

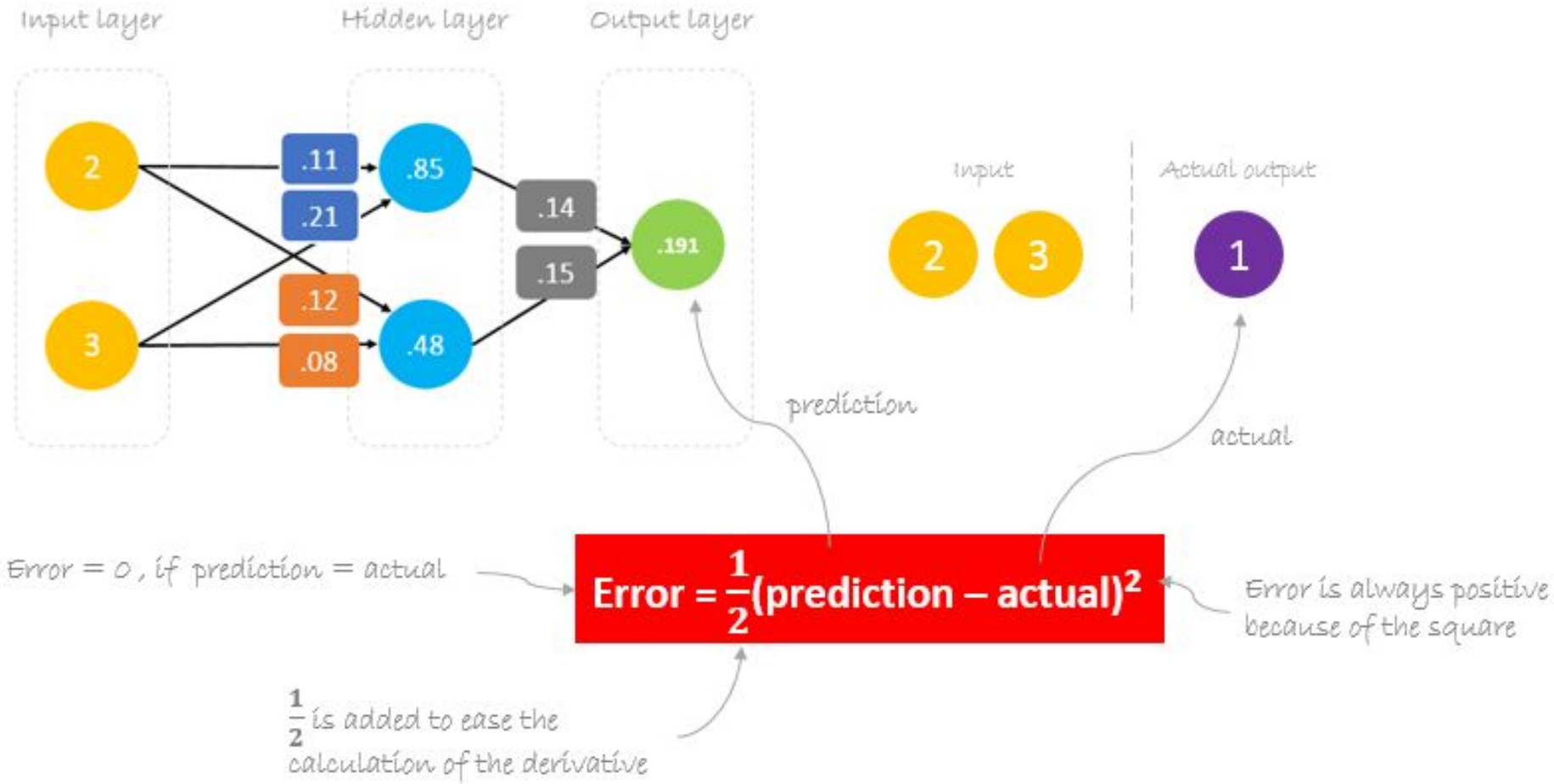
w4 = 0.08

w5 = 0.14

w6 = 0.15

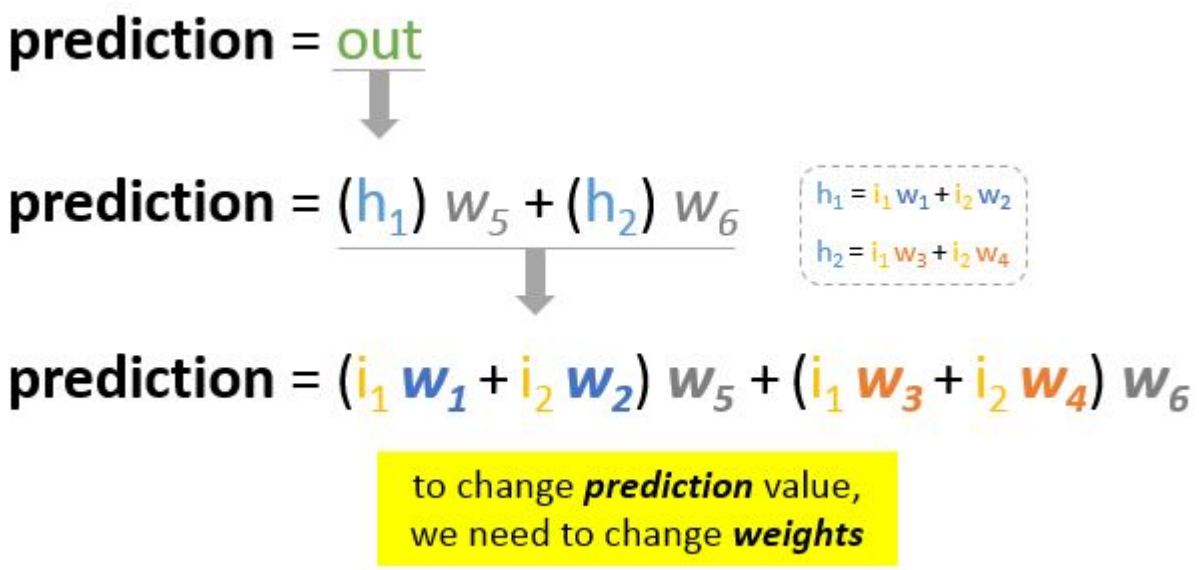


Error Calculation



Error Calculation

- Objective is to ***reduce the Error*** – Difference between Actual & Predicted.



Backword Pass – Update W6

$$^{*}W_6 = W_6 - a\left(\frac{\partial Error}{\partial W_6}\right)$$

$$\frac{\partial Error}{\partial W_6} = \frac{\partial Error}{\partial prediction} * \frac{\partial prediction}{\partial W_6}$$

chain rule

Error = $\frac{1}{2}$(prediction – actual)^2

prediction = $(i_1 w_1 + i_2 w_2) w_5 + (i_1 w_3 + i_2 w_4) w_6$

$$\frac{\partial Error}{\partial W_6} = \frac{\frac{1}{2}(predictoin - actula)^2}{\partial prediciton} * \frac{\partial\left(i_1 w_1 + i_2 w_2\right) w_5 + \left(i_1 w_3 + i_2 w_4\right) w_6}{\partial W_6}$$

$$\frac{\partial Error}{\partial W_6} = 2 * \frac{1}{2}(\mathbf{predictoin - actula}) \frac{\partial(predictoin - actula)}{\partial prediciton} * (i_1 w_3 + i_2 w_4)$$

$h_2 = i_1 w_3 + i_2 w_4$

$$\frac{\partial Error}{\partial W_6} = (\mathbf{predictoin - actula}) * (h_2)$$

Δ = prediction – actual

delta

$$\frac{\partial Error}{\partial W_6} = \Delta h_2$$

Backword Pass – Update W1

$$\frac{\partial Error}{\partial W_1} = \frac{\partial Error}{\partial prediction} * \frac{\partial prediction}{\partial h_1} * \frac{\partial h_1}{\partial W_1}$$ ← chain rule

$$\frac{\partial Error}{\partial W_1} = \frac{\partial \frac{1}{2}(predictoin - actula)^2}{\partial prediciton} * \frac{\partial (h_1)\, w_5 + (h_2)\, w_6}{\partial h_1} * \frac{\partial\, i_1 w_1 + i_2 w_2}{\partial w_1}$$

$$\frac{\partial Error}{\partial W_1} = 2 * \frac{1}{2}(\mathbf{predictoin - actula}) \frac{\partial (predictoin - actula)}{\partial prediciton} * (w_5) * (i_1)$$

$$\frac{\partial Error}{\partial W_1} = (\mathbf{predictoin - actula}) * (w_5 i_1)$$

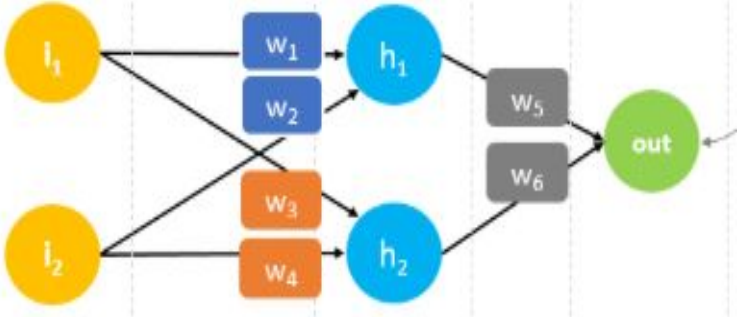
$$\frac{\partial Error}{\partial W_1} = \Delta\, w_5 i_1$$

Error = $\frac{1}{2}$(prediction – actual)²

prediction = $(h_1)\, w_5 + (h_2)\, w_6$

$h_1 = i_1 w_1 + i_2 w_2$

Δ = prediction – actual ← delta



Weight Update through Error propagation

updated weights

$$^{*}w_6 = w_6 - a\,(h_2 \,.\, \Delta)$$

$$^{*}w_5 = w_5 - a\,(h_1 \,.\, \Delta)$$

$$^{*}w_4 = w_4 - a\,(i_2 \,.\, \Delta w_6)$$

$$^{*}w_3 = w_3 - a\,(i_1 \,.\, \Delta w_6)$$

$$^{*}w_2 = w_2 - a\,(i_2 \,.\, \Delta w_5)$$

$$^{*}w_1 = w_1 - a\,(i_1 \,.\, \Delta w_5)$$

$$\begin{bmatrix} w_5 \\ w_6 \end{bmatrix} = \begin{bmatrix} w_5 \\ w_6 \end{bmatrix} - a\,\Delta \begin{bmatrix} h_1 \\ h_2 \end{bmatrix} = \begin{bmatrix} w_5 \\ w_6 \end{bmatrix} - \begin{bmatrix} a h_1 \Delta \\ a h_2 \Delta \end{bmatrix}$$

$$\begin{bmatrix} w_1 & w_3 \\ w_2 & w_4 \end{bmatrix} = \begin{bmatrix} w_1 & w_3 \\ w_2 & w_4 \end{bmatrix} - a\,\Delta \begin{bmatrix} i_1 \\ i_2 \end{bmatrix} . \begin{bmatrix} w_5 & w_6 \end{bmatrix} = \begin{bmatrix} w_1 & w_3 \\ w_2 & w_4 \end{bmatrix} - \begin{bmatrix} a\,i_1 \Delta w_5 & a\,i_1 \Delta w_6 \\ a\,i_2 \Delta w_5 & a\,i_2 \Delta w_6 \end{bmatrix}$$

Backward Pass – Example

- **Learning rate:** is a Hyperparameter which means that we need to manually guess its value.

$\Delta = 0.191 - 1 = -0.809$ ← **Delta** = prediction - actual

$a = 0.05$ ← **Learning rate,** we smartly guess this number

$$\begin{bmatrix} w_5 \\ w_6 \end{bmatrix} = \begin{bmatrix} 0.14 \\ 0.15 \end{bmatrix} - 0.05(-0.809) \begin{bmatrix} 0.85 \\ 0.48 \end{bmatrix} = \begin{bmatrix} 0.14 \\ 0.15 \end{bmatrix} - \begin{bmatrix} -0.034 \\ -0.019 \end{bmatrix} = \begin{bmatrix} 0.17 \\ 0.17 \end{bmatrix}$$

$$\begin{bmatrix} w_1 & w_3 \\ w_2 & w_4 \end{bmatrix} = \begin{bmatrix} .11 & .12 \\ .21 & .08 \end{bmatrix} - 0.05(-0.809) \begin{bmatrix} 2 \\ 3 \end{bmatrix} . \begin{bmatrix} 0.14 & 0.15 \end{bmatrix} = \begin{bmatrix} .11 & .12 \\ .21 & .08 \end{bmatrix} - \begin{bmatrix} -0.011 & -0.012 \\ -0.017 & -0.018 \end{bmatrix} = \begin{bmatrix} .12 & .13 \\ .23 & .10 \end{bmatrix}$$

Back Propagation Algorithm

Algorithm: back_propagation$(h_1, h_2, ..., h_{L-1}, a_1, a_2, ..., a_L, y, \hat{y})$

//Compute output gradient ;

$\nabla_{a_L} \mathscr{L}(\theta) = -(e(y) - \hat{y})$;

for $k = L$ *to* 1 **do**

  // Compute gradients w.r.t. parameters ;

  $\nabla_{W_k} \mathscr{L}(\theta) = \nabla_{a_k} \mathscr{L}(\theta) h_{k-1}^T$;

  $\nabla_{b_k} \mathscr{L}(\theta) = \nabla_{a_k} \mathscr{L}(\theta)$;

  // Compute gradients w.r.t. layer below ;

  $\nabla_{h_{k-1}} \mathscr{L}(\theta) = W_k^T (\nabla_{a_k} \mathscr{L}(\theta))$;

  // Compute gradients w.r.t. layer below (pre-activation);

  $\nabla_{a_{k-1}} \mathscr{L}(\theta) = \nabla_{h_{k-1}} \mathscr{L}(\theta) \odot [\ldots, g'(a_{k-1,j}), \ldots]$;

end

Animation

https://hmkcode.com/netflow/

Loss Functions

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

Where $\hat{y}$ is the predicted value and y is the true value

$$\text{Binary cross entropy} = -(y\log(\hat{y}) + (1-y)\log(1-\hat{y}))$$

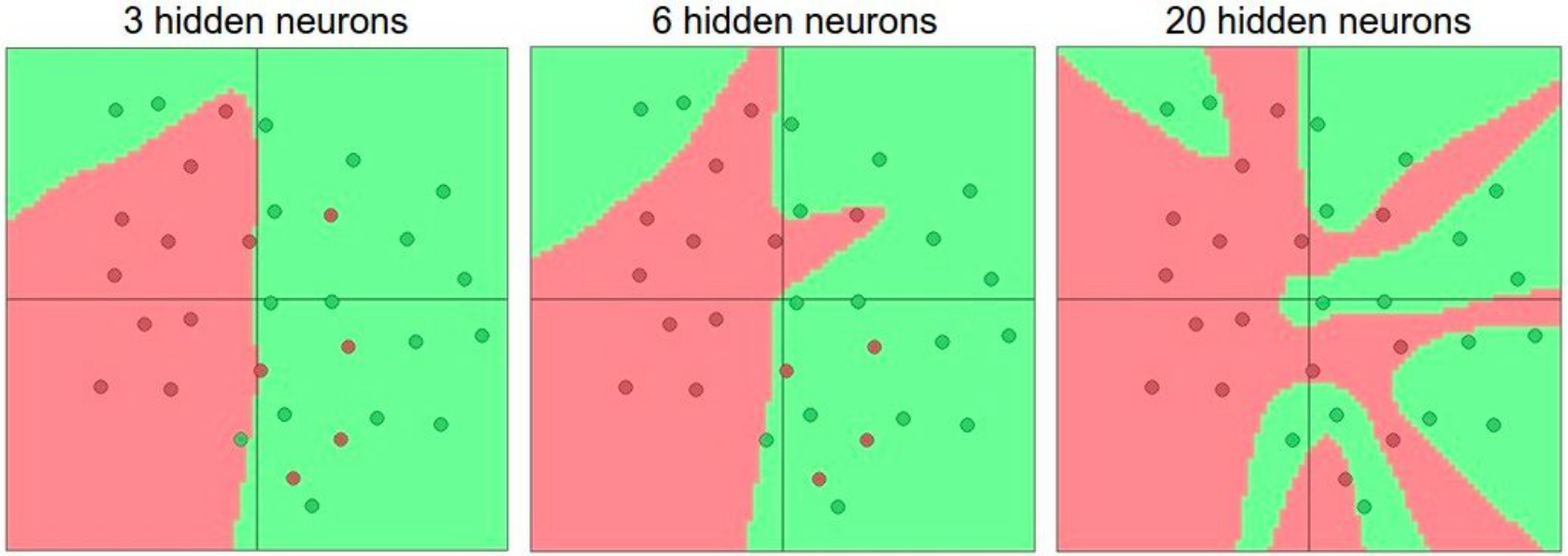
Where $\hat{y}$ is the predicted value and y is the true value

$$\text{Cross entropy} = -\sum_{i}^{M} y_i \log(\hat{y}_i)$$

Where $\hat{y}$ is the predicted value, y is the true value and M is the number of classes

Activation Functions

- Non-linearities needed to learn complex (non-linear) representations of data, otherwise the NN would be just a linear function



- More layers and neurons can approximate more complex functions

Activation: Sigmoid

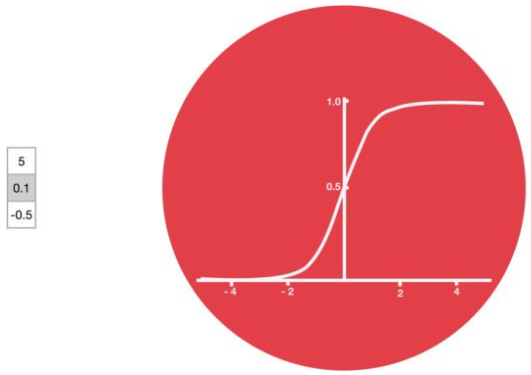
- Takes a real-valued number and “squashes” it into range between 0 and 1.

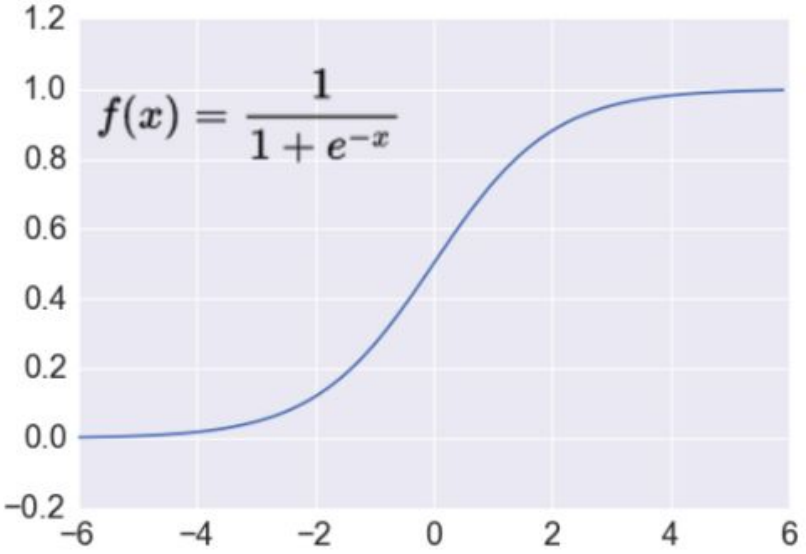
\+ Nice interpretation as the **firing rate** of a neuron

- 0 = not firing at all
- 1 = fully firing

\- Sigmoid neurons **saturate** and **kill gradients**, thus NN will barely learn

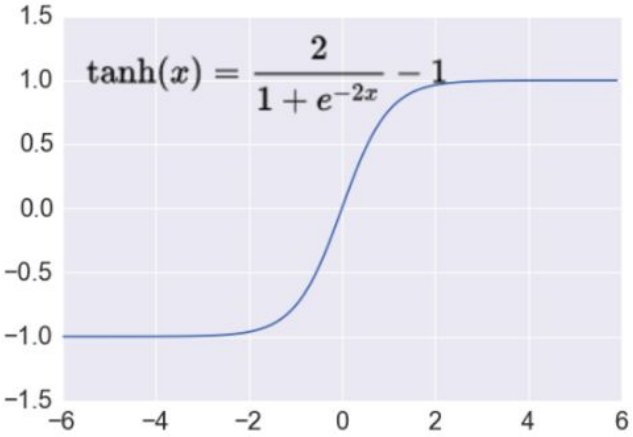
- when the neuron’s activation are 0 or 1 (saturate)
  - 🙁 gradient at these regions almost zero
  - 🙁 almost no signal will flow to its weights
  - 🙁 if initial weights are too large then most neurons would saturate





Activation: Tanh

- Takes a real-valued number and “squashes” it into range between -1 and 1.



- Like sigmoid, tanh neurons **saturate**
- Unlike sigmoid, output is **zero-centered**
- Tanh is a **scaled sigmoid**: $\tanh(x) = 2sigm(2x) - 1$

Activation: ReLU

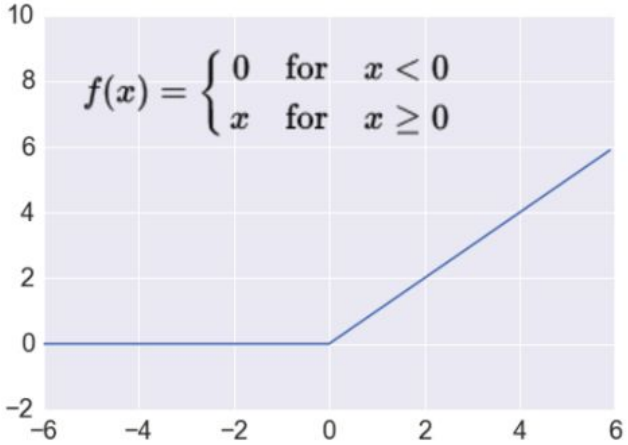
- Takes a real-valued number and thresholds it at zero
- Most Deep Networks use ReLU nowadays

🙂 Trains much **faster**
  - accelerates the convergence of SGD
  - due to linear, non-saturating form

🙂 Less expensive operations
  - compared to sigmoid/tanh (exponentials etc.)
  - implemented by simply thresholding a matrix at zero

🙂 More **expressive**

🙂 Prevents the **gradient vanishing problem**

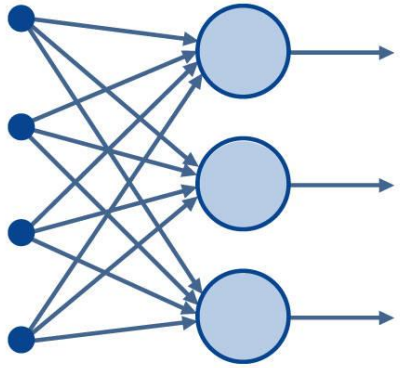


Tips to choose activation and loss functions

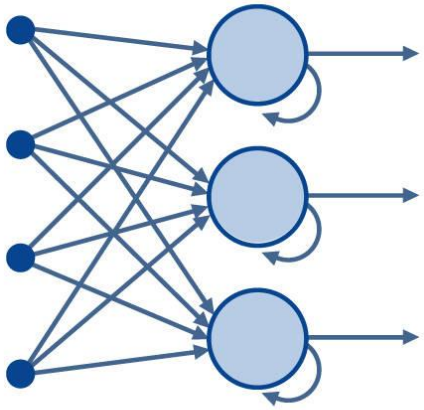
Problem Type	Output Type	Final Activation Function	Loss Function
Regression	Numerical value	Linear	Mean Squared Error (MSE)
Classification	Binary outcome	Sigmoid	Binary Cross Entropy
Classification	Single label, multiple classes	Softmax	Cross Entropy
Classification	Multiple labels, multiple classes	Sigmoid	Binary Cross Entropy

Types of NNs based on Architectures

- The architecture of a Neural Network can be broadly classified into two, namely:
  1. Feed Forward Artificial Neural Network (DNN, CNN..)
  2. Recurrent Neural Network (RNN, LSTM, GRU..)



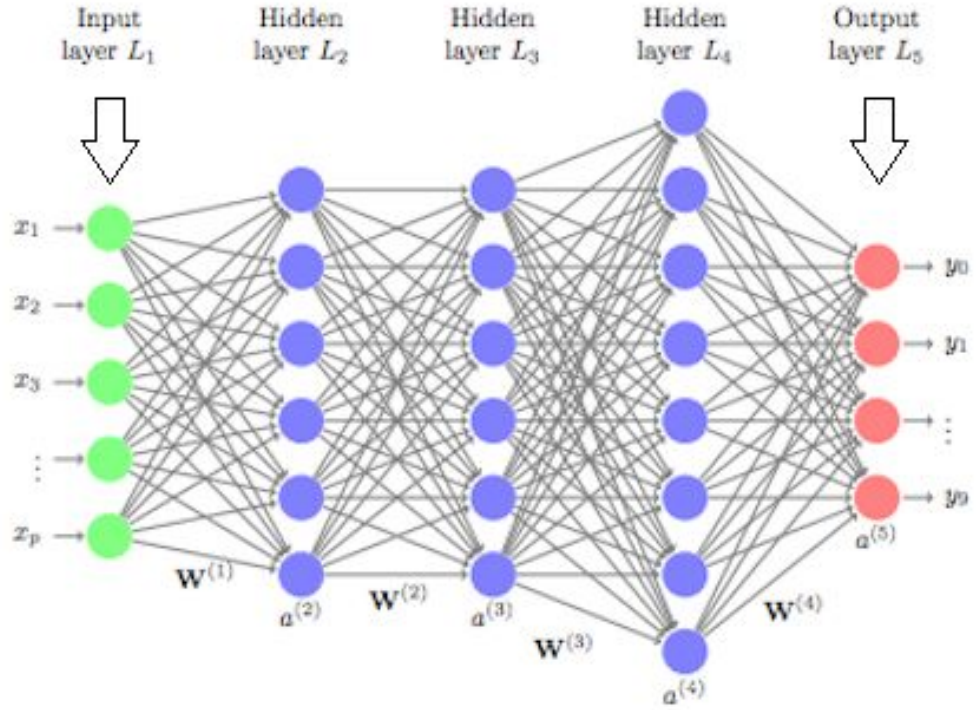
- The information must flow from input to output only in one direction.
- No Feedback loops must be present.



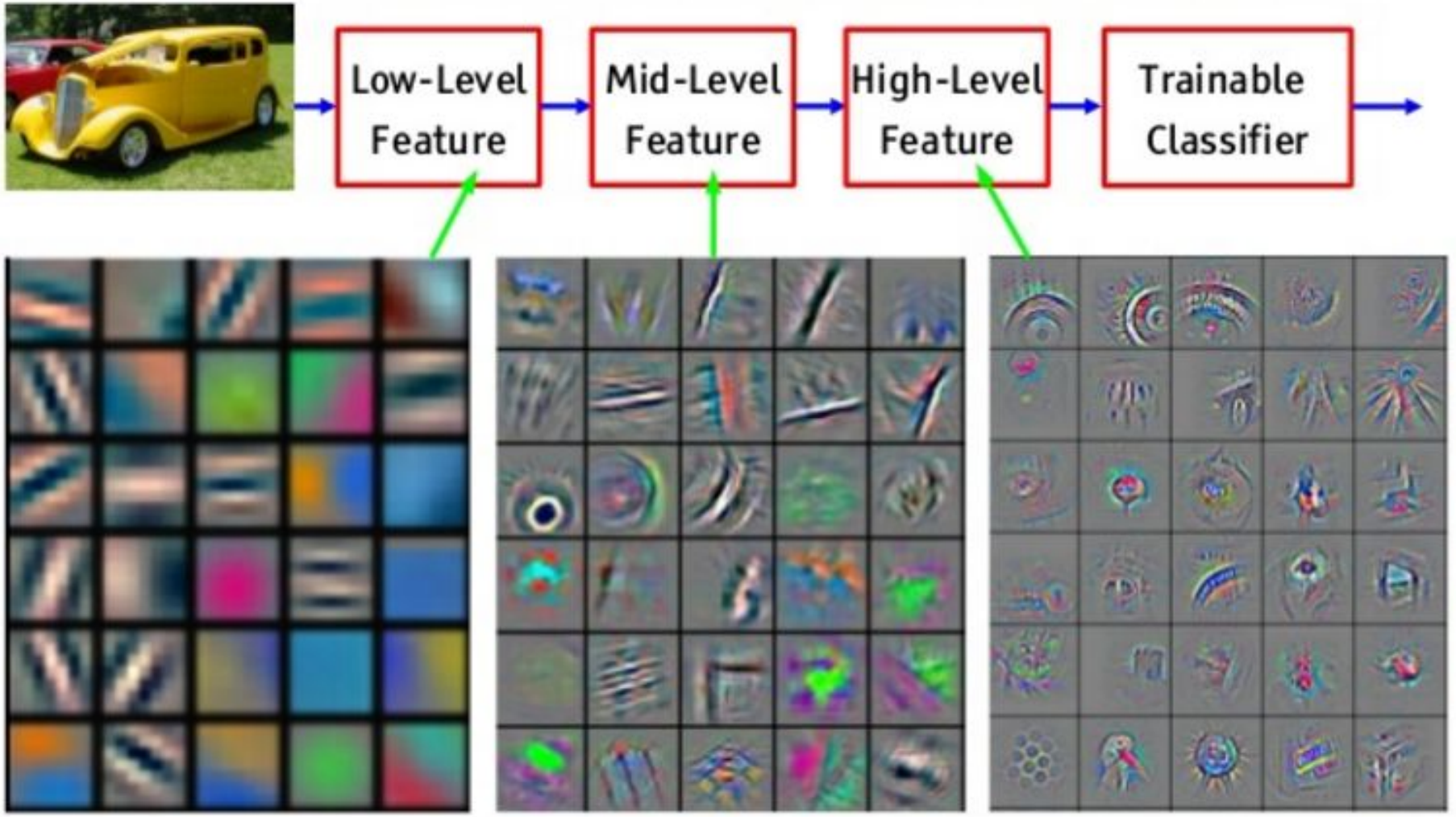
- Information can be transmitted in both directions.
- Feedback loops are allowed.

Deep Neural Network (DNN)

- A Deep Neural Network (DNN) is an Artificial Neural Network (ANN) with multiple layers between the input and output layers.
- The DNN finds the correct mathematical manipulation to turn the input into the output, whether it be a linear relationship or a non-linear relationship.



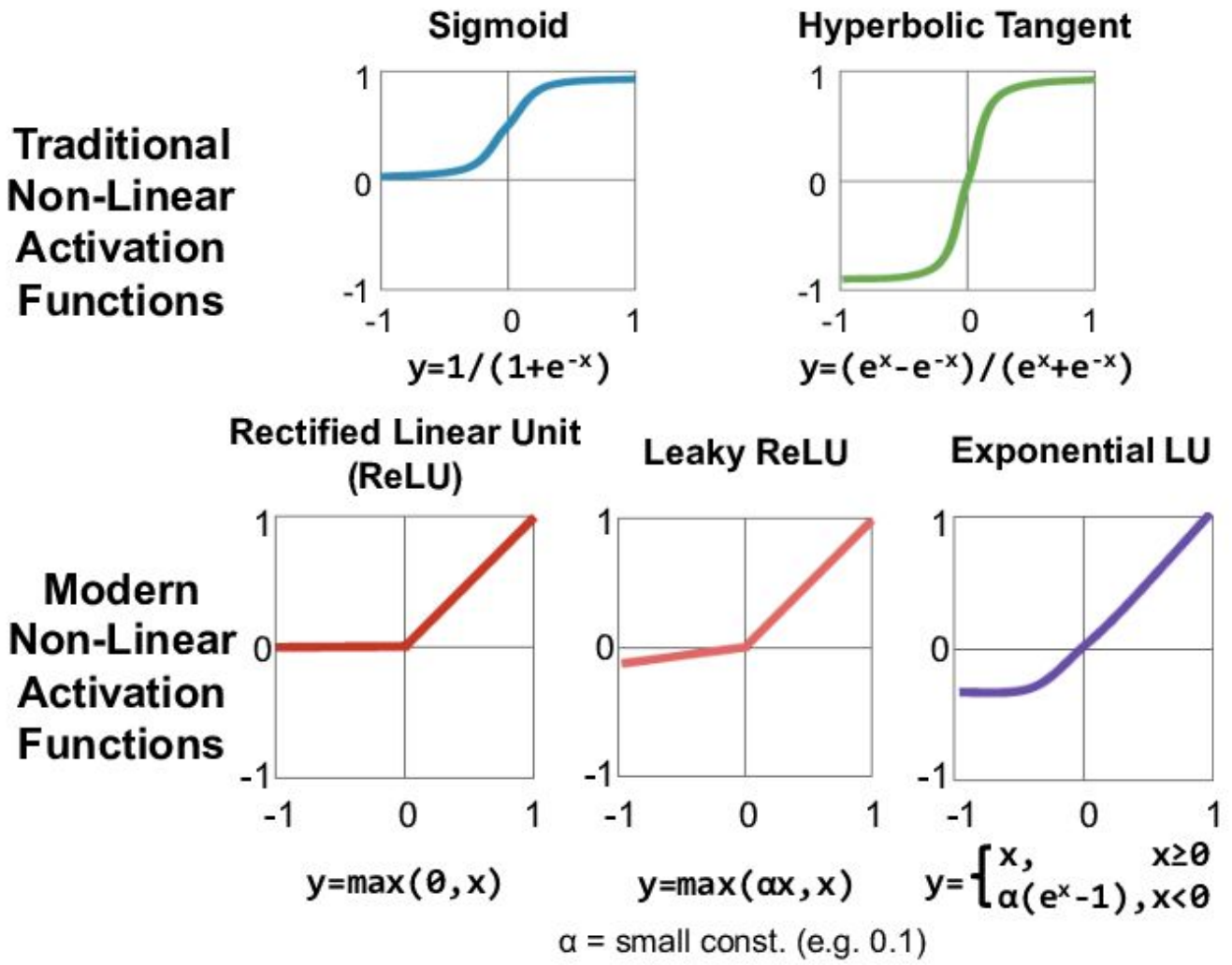
What DNNs Learn…



MLP vs DNN

- Number of Layers
- Datasets (Size)
- Computing Resources (Memory & Time efficient)
- Weight Initialization
- Non-linear activation Functions

MLP vs DNN


Traditional Non-Linear Activation Functions
Sigmoid
y=1/(1+e^-x)
Hyperbolic Tangent
y=(e^x-e^-x)/(e^x+e^-x)
Modern Non-Linear Activation Functions
Rectified Linear Unit (ReLU)
y=max(0,x)
Leaky ReLU
y=max(αx,x)
Exponential LU
y={x, x≥0; α(e^x-1), x<0
α = small const. (e.g. 0.1)

Implementing DNN in Python

Bias, Variance & their trade-off

- The main goal of ML/DL models is learn patterns from the data provided and make predictions which are close to actual values.
- There are chances that model may barely learn / over learn the patterns from the data.
- This results in poor estimates (predictions) on testing data – referred as Errors.
- Main reason for this is Bias & Variance Erros – reducible errors

Bias, Variance & their trade-off

- Difference occurred between model predictions and the actual values/expected values is known as bias errors / Errors due to bias. It is inability of ML algorithms to capture the true relationship between the data points.
- Each algorithm begins with some amount of bias, as it occurs from assumptions in the model, which makes the target function barely learn.
- **Low Bias:** A low bias model will make fewer assumptions about the form of the target function.
- **High Bias:** A model with a high bias makes more assumptions, and the model becomes unable to capture the important features of our dataset. A high bias model also cannot perform well on new data. High bias mainly occurs due to a much simple model, and leads to **Underfitting**. This can be handled by increase the input features or increase the complexity of the model for training

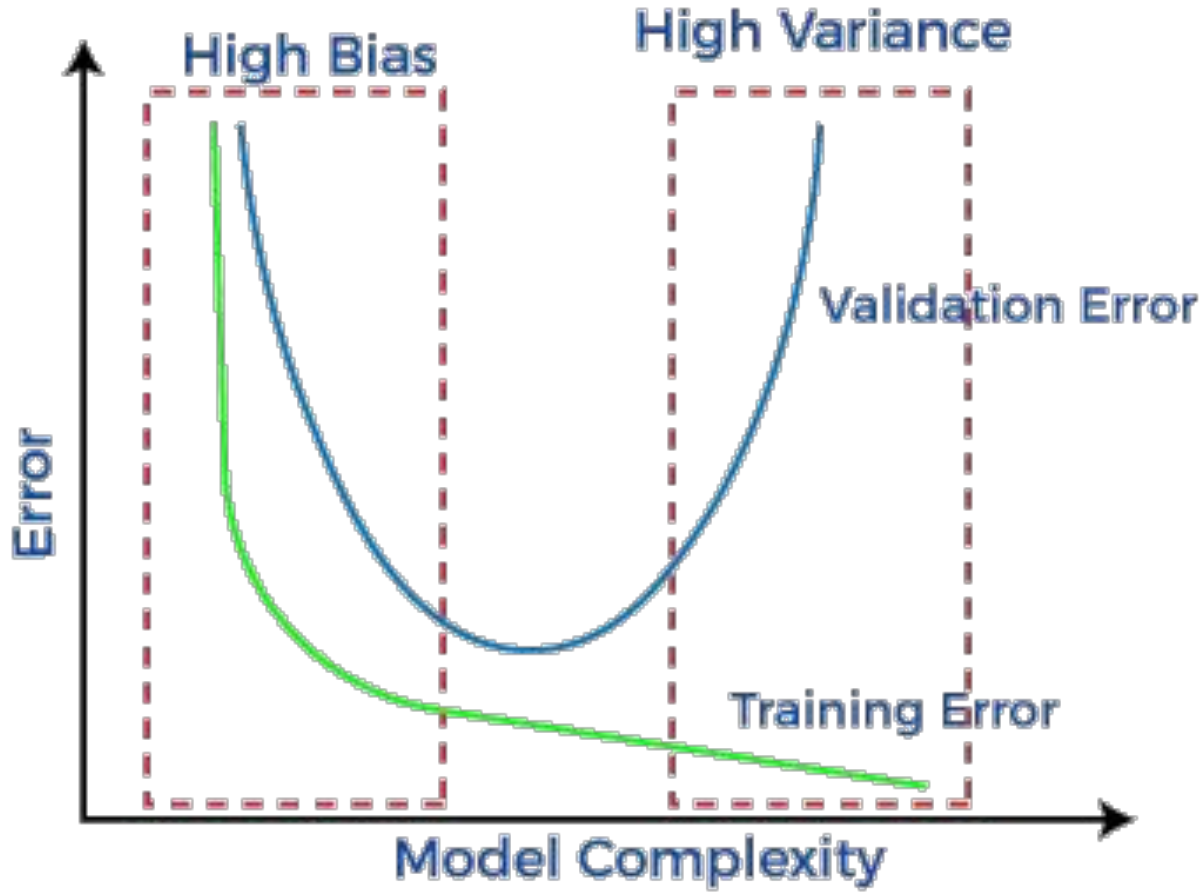
Bias, Variance & their trade-off

- The variance would specify the amount of variation in the prediction if the different training data was used. Variance tells that how much a random variable is different from its expected value.
- Ideally, a model should not vary too much from one training dataset to another, which means the algorithm should be good in understanding the hidden mapping between inputs and output variables.
- **Low variance** means there is a small variation in the prediction of the target function with changes in the training data set.
- **High variance** shows a large variation in the prediction of the target function with changes in the training dataset. Model with high variance learns a lot and perform well with the training dataset, and does not generalize well with the unseen dataset. good results with the training dataset but high error rates on the test dataset – **Overfitting.** This can be reduced by reducing the input features or by simplifying the complexity of model.

Bias, Variance & their trade-off

- **Low-Bias, Low-Variance:** The combination of low bias and low variance shows an ideal machine learning model. However, it is not possible practically.
- **Low-Bias, High-Variance:** With low bias and high variance, model predictions are inconsistent and accurate on average. This case occurs when the model learns with a large number of parameters and hence leads to an overfitting
- **High-Bias, Low-Variance:** With High bias and low variance, predictions are consistent but inaccurate on average. This case occurs when a model does not learn well with the training dataset or uses few numbers of the parameter. It leads to underfitting problems in the model.
- **High-Bias, High-Variance:** With high bias and high variance, predictions are inconsistent and also inaccurate on average.

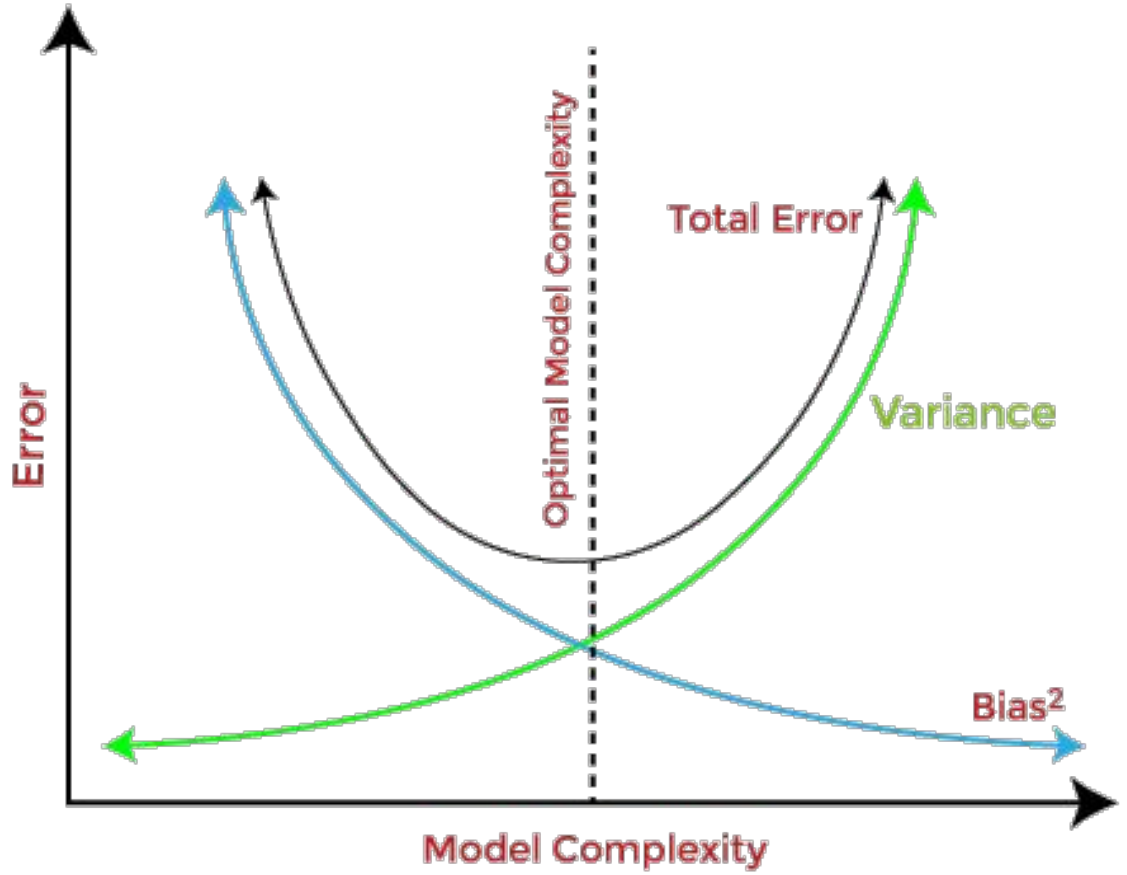
Bias, Variance & their trade-off



Bias, Variance & their trade-off

- While building the machine learning model, it is really important to take care of bias and variance in order to avoid overfitting and underfitting in the model.
- If the model is very simple with fewer parameters, it may have low variance and high bias.
- Whereas, if the model has a large number of parameters, it will have high variance and low bias.
- So, it is required to make a balance between bias and variance errors, and this balance between the bias error and variance error is known as the Bias-Variance trade-off.

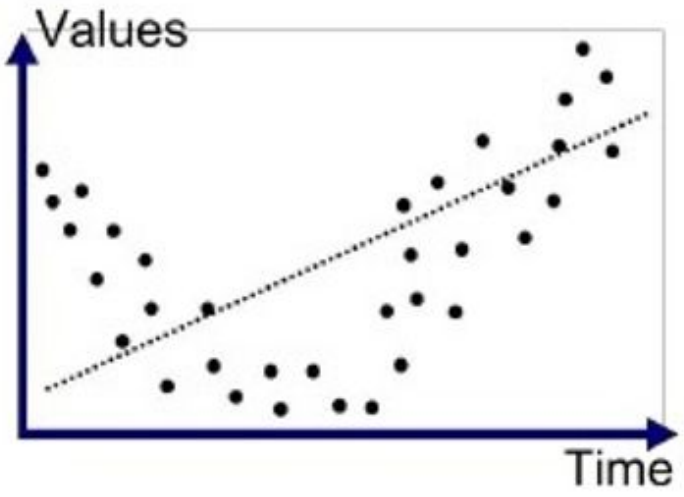
Bias, Variance & their trade-off



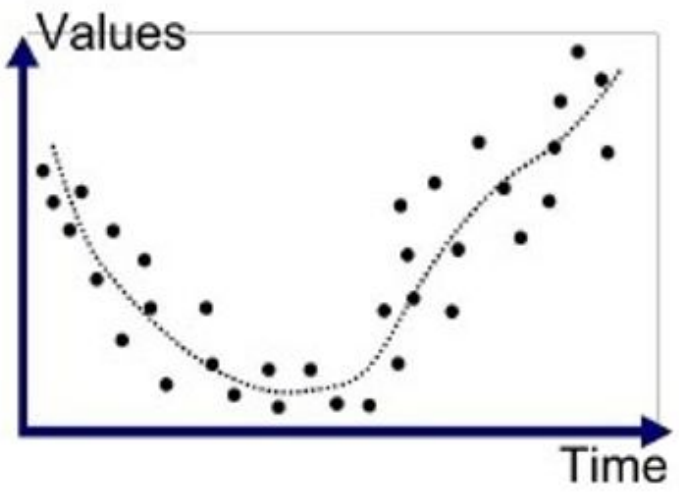
Problem With Overfitting / Underfitting in NNs

- Large neural nets trained on relatively small datasets can overfit the training data.
- This has the effect of the model learning the statistical noise in the training data, which results in poor performance when the model is evaluated on new data, e.g. a test dataset.
- Generalization error increases due to overfitting.

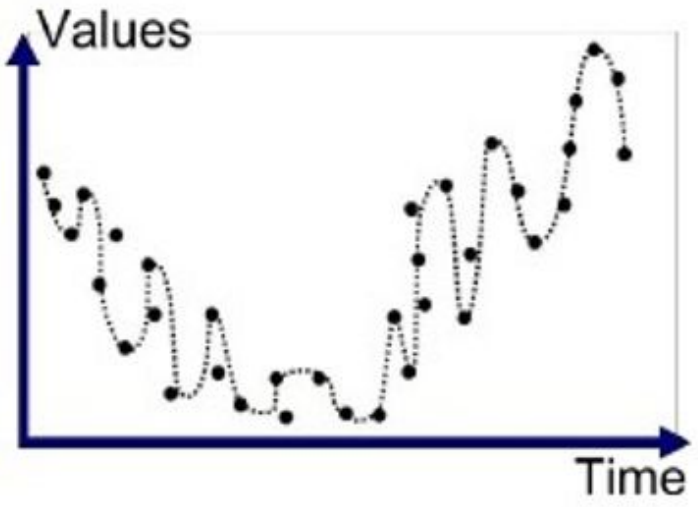
Problem With Overfitting / Underfitting



Underfitted



Good Fit/Robust

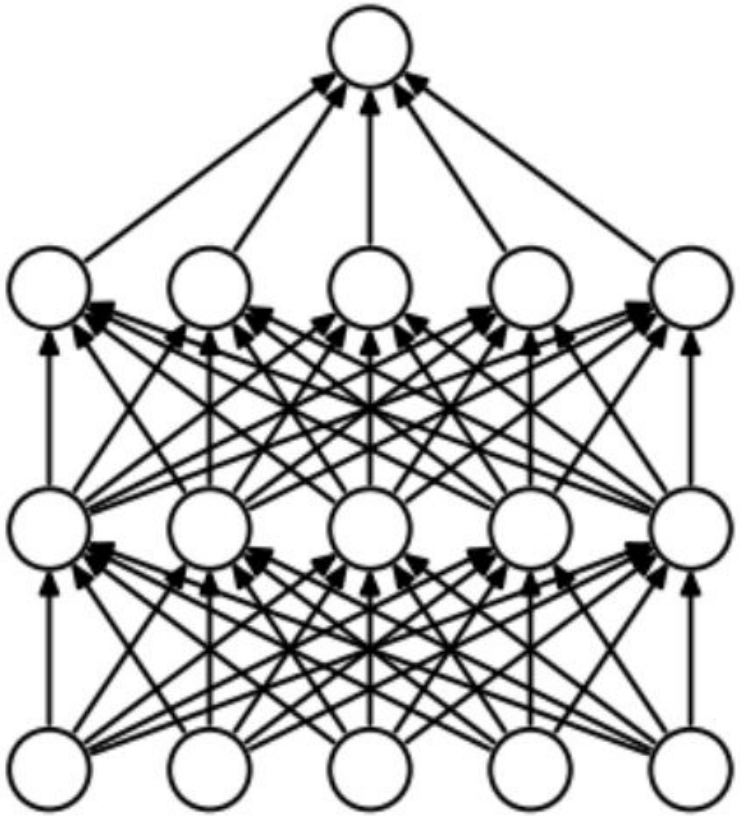


Overfitted

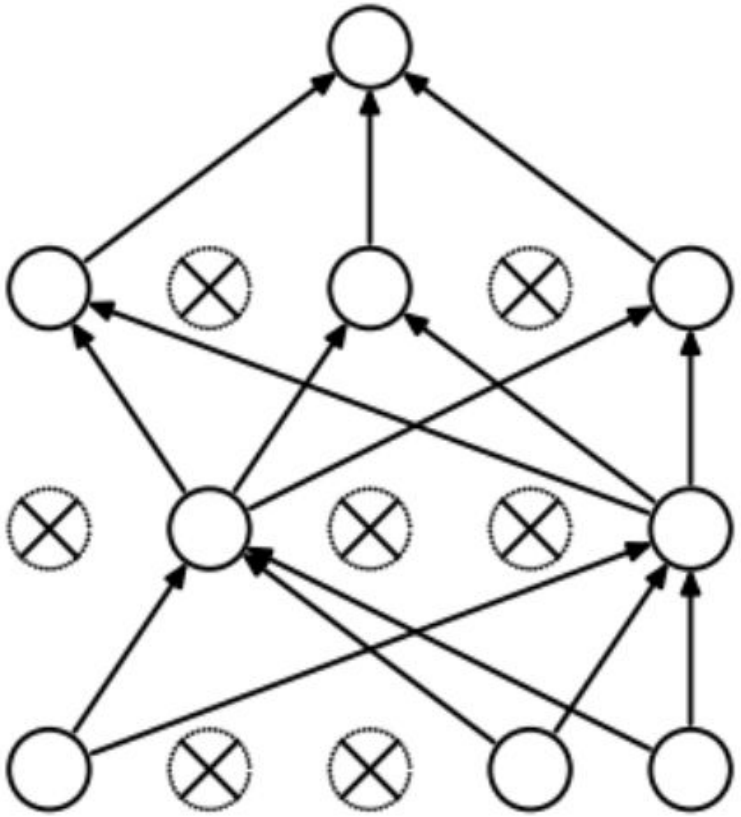
Role of Dropout

- Randomly Drop Nodes
- Dropout is a regularization method that approximates training a large number of neural networks with different architectures in parallel.
- During training, some number of layer outputs are randomly ignored or “dropped out.”
- By dropping a unit out, we mean temporarily removing it from the network, along with all its incoming and outgoing connections.
- Dropout simulates a sparse activation from a given layer, which interestingly, in turn, encourages the network to actually learn a sparse representation as a side-effect.
- As such, it may be used as an alternative to activity regularization for encouraging sparse representation
- Dropout: A Simple Way to Prevent Neural Networks from Overfitting, 2014.

Role of Dropout



(a) Standard Neural Net

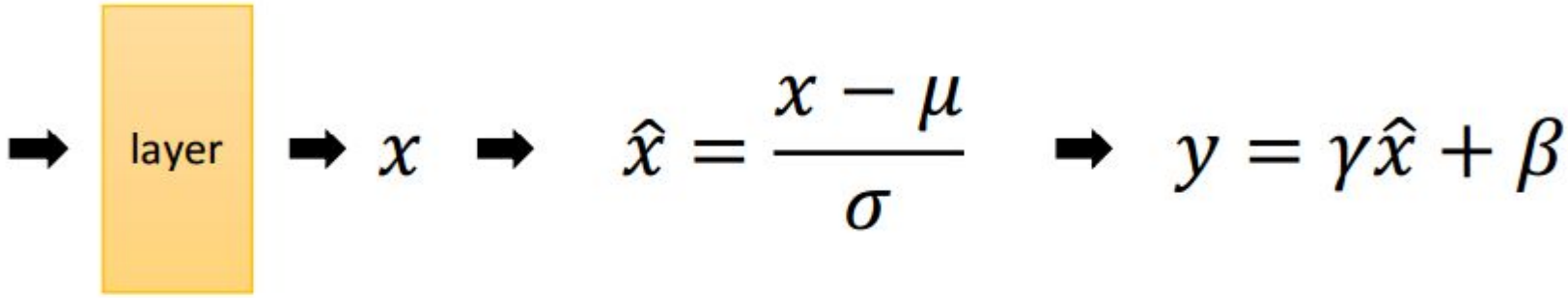


(b) After applying dropout.

Batch Normalization

- Training deep neural networks, e.g. networks with tens of hidden layers, is challenging.
- Training Deep Neural Networks is complicated by the fact that the distribution of each layer's inputs changes during training, as the parameters of the previous layers change.
- This slows down the training by requiring lower learning rates and careful parameter initialization, and makes it notoriously hard to train models with saturating nonlinearities.
- Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, 2015.
- The authors of the paper introducing batch normalization refer to change in the distribution of inputs during training as "internal covariate shift."

Batch Normalization



- μ: mean of x in mini-batch
- σ: std of x in mini-batch
- γ: scale
- β: shift

- μ, σ: functions of x, analogous to responses
- γ, β: parameters to be learned, analogous to weights

Problem with Deep Neural Networks

- The downside of the Deep / Convolutional Neural Networks (CNN) is that they need enormous amounts of data for training.
- This is usually scarce for most of the real-time applications.
- This problem can be addressed by using Transfer Learning.
- Traditionally Data Augmentation Techniques were also used to increase the size of training data and then train the models for desired task.
- Transfer Learning shows effective results when in comparison with data augmentation techniques.

Implementing Regularized NNs

Web Application Development for ML/DL Models

- Crest RESTful Webservices for ML/DL models using Python Flask / Django and reuse them in any web applications – Language independent.

Project Structure for FLASK Applications

- Project Root Folder
  - template - html/css/js files
  - static – static data to be used within the application
  - app.py - project starter file

Thank You