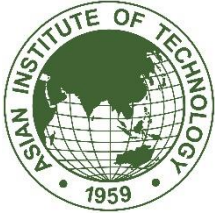


Artificial Neural Network
Forward Propagation

Dr. Mongkol Ekpanyapong

Black Block model

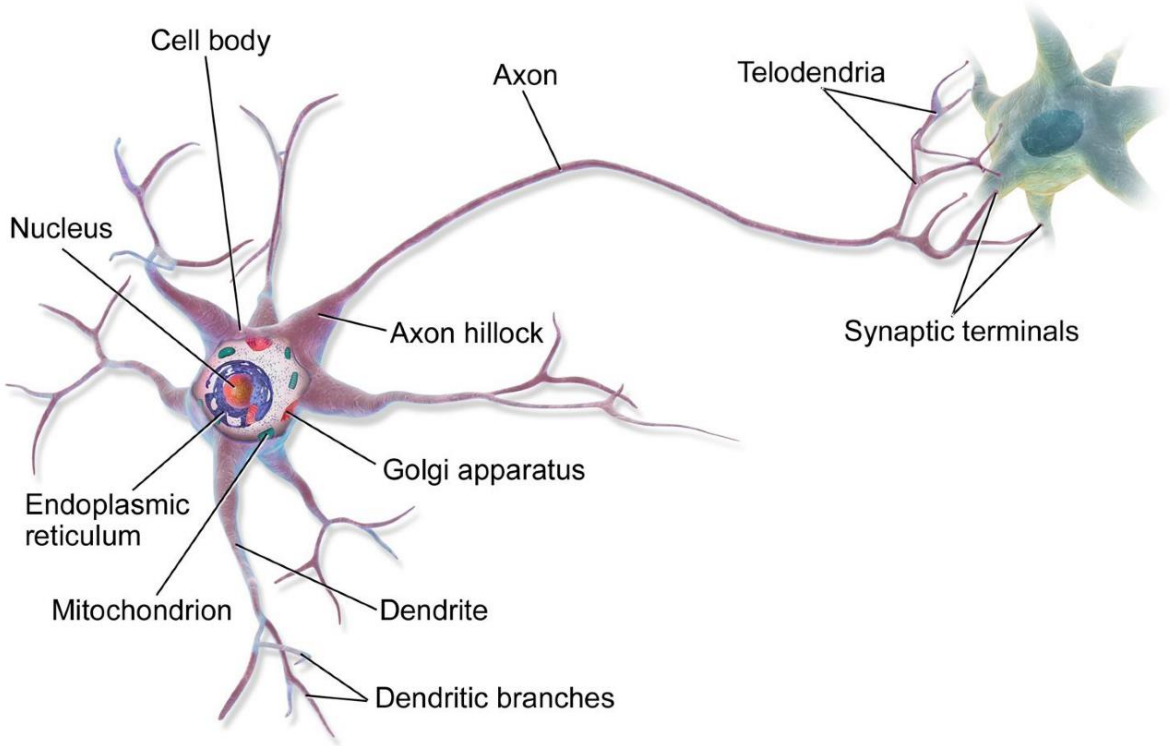
- Artificial Neural Network (ANN) is considered as a black box model

- In contrast with other machine learning such as decision tree or Support Vector Machine (SVM) that are considered as a white box model

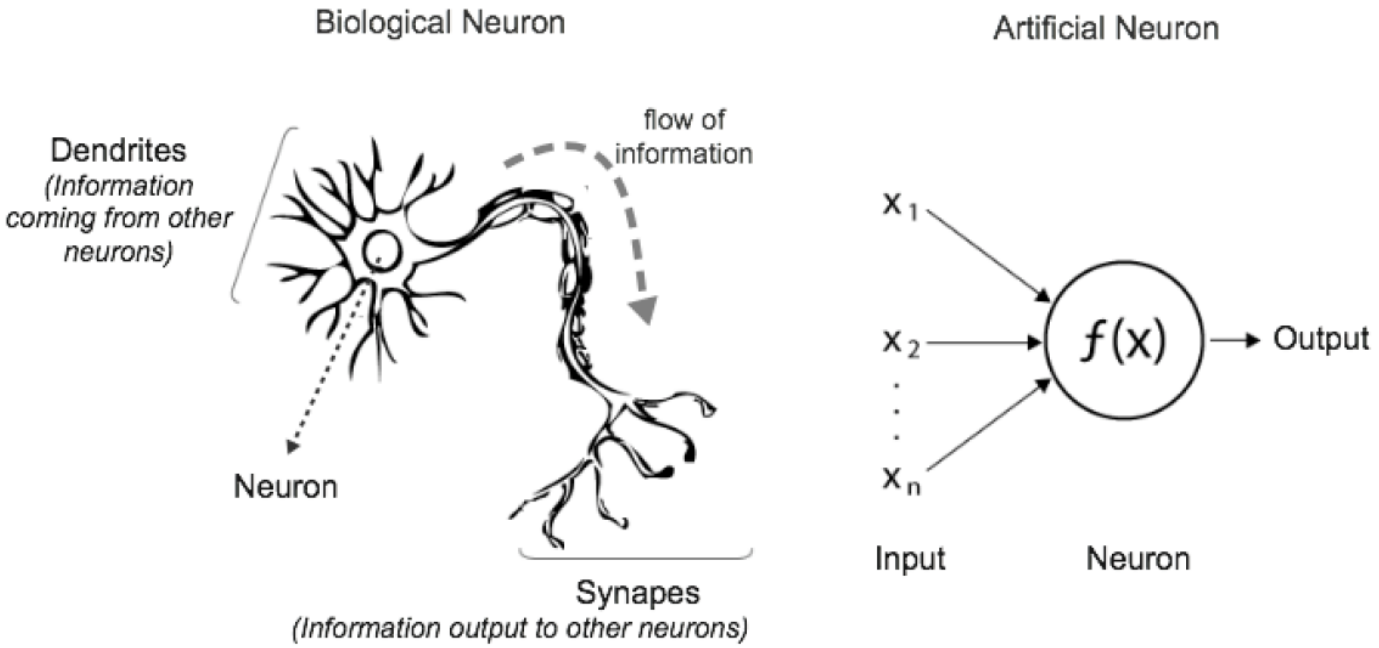
Perceptron Model

- It is the cell mainly found in the brain
- The cell consists of:
  - Cell body
  - Dendrits (branching extensions)
  - Axon (very long extension)
  - Telodendria (split off Axon)
  - Synapses (terminal of Telodendria), it will connect to the other celss

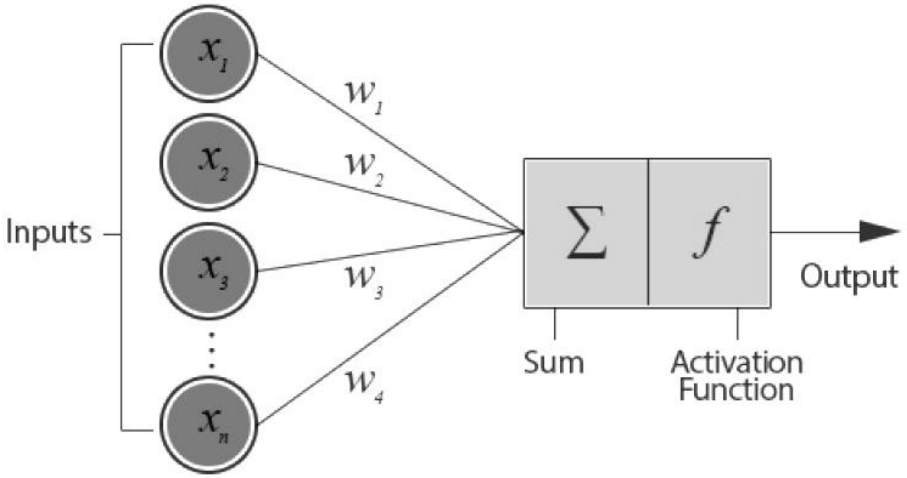
Perceptron Model



Model Comparison



ANN Model



- Input vector
- Weight vector
- Activation Function
- Output vector

Weighted Sum Function

- Linear combination can be used for weight calculation

$$z = \sum x_i \ .w_i \ + b \ (bias)$$

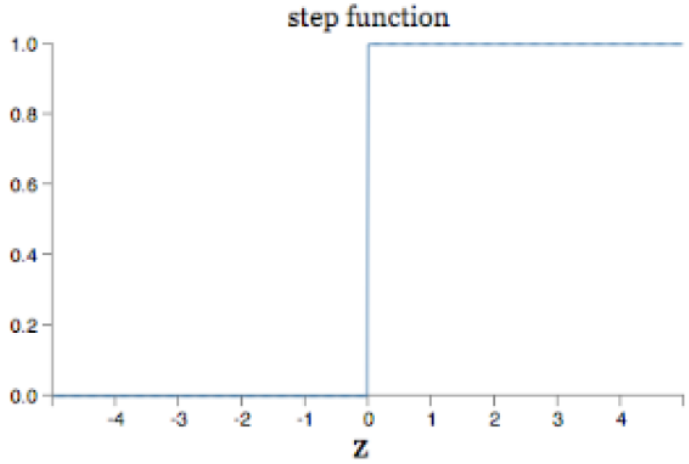
$$z = x_1 \ .w_1 + x_2.w_2 + x_3.w_3 + ...+ x_n \ .w_n \ + b$$

- Python code:

```
# X is the input vector (denoted with an uppercase X)
# w is the weights vector, b is y-intercept
z = np.dot(w.T,X) + b
```

Activation Function

- Activation function is used to introduce non-linearity in the system



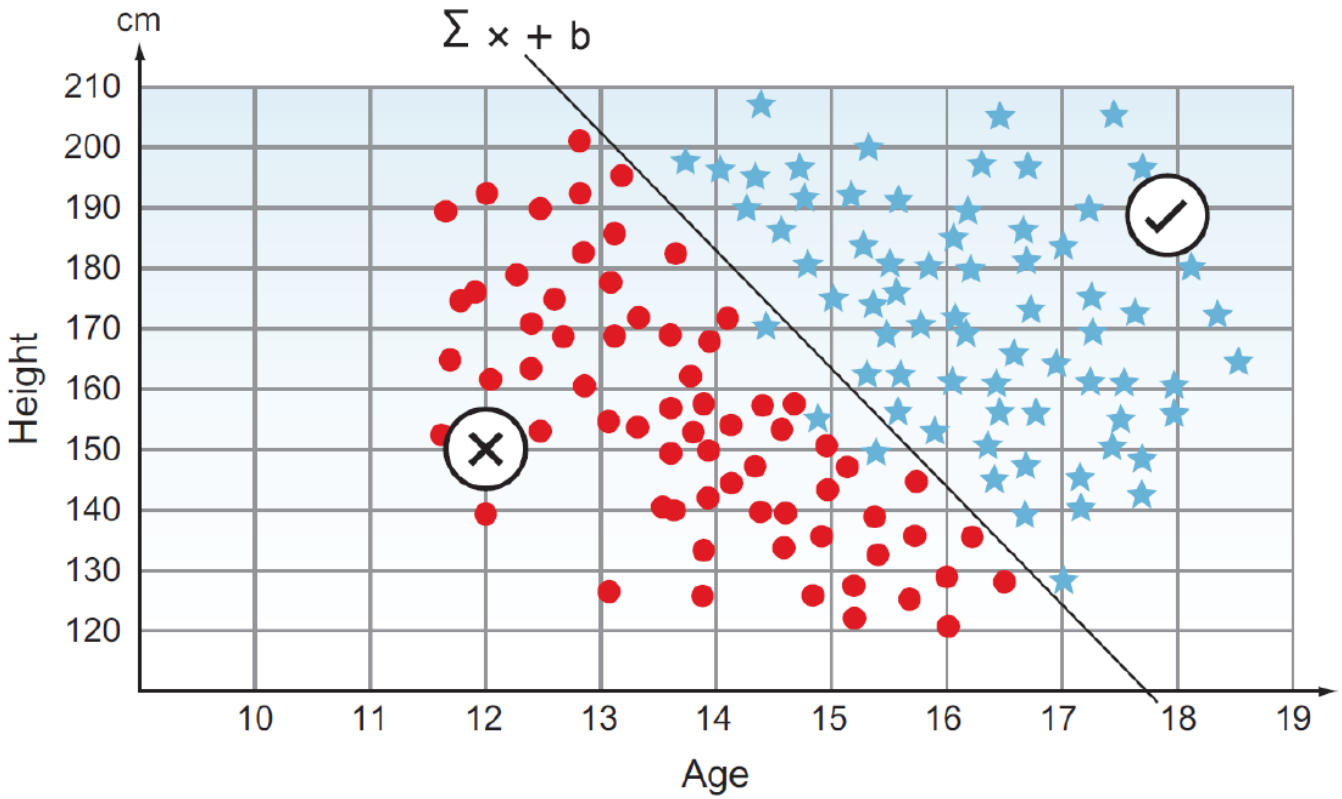
$$\text{output} = \begin{cases} 0 & \text{if } w \cdot x + b \leq 0 \\ 1 & \text{if } w \cdot x + b > 0 \end{cases}$$

```
# z is the weighted sum =  sum = ∑xi . wi + b
def step_function(z):
            if z <= 0:
                return 0
           else:
    return 1
```

How does Perceptron learn?

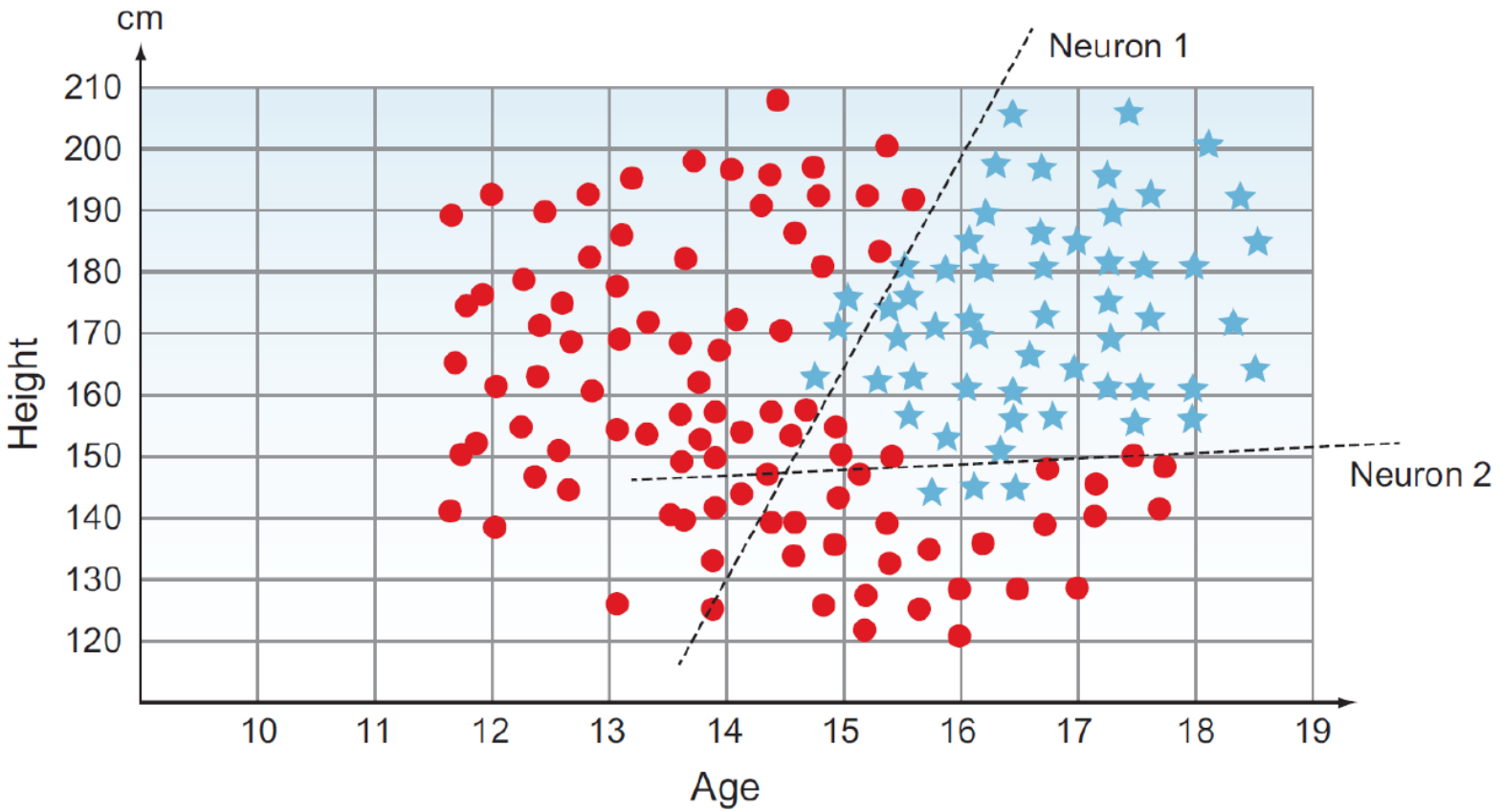
1. The neuron calculates the weighted sum and apply the activation function to make a prediction (feedforward process)
2. It then compares the prediction with the correct label to calculate the error
3. Update the weight: if the prediction is too high, it will adjust the weights to make a lower prediction next time
4. Repeat Step 1

Power of One Neural

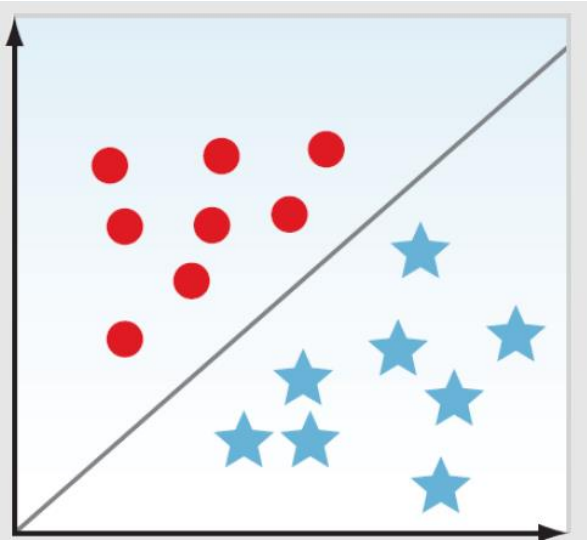


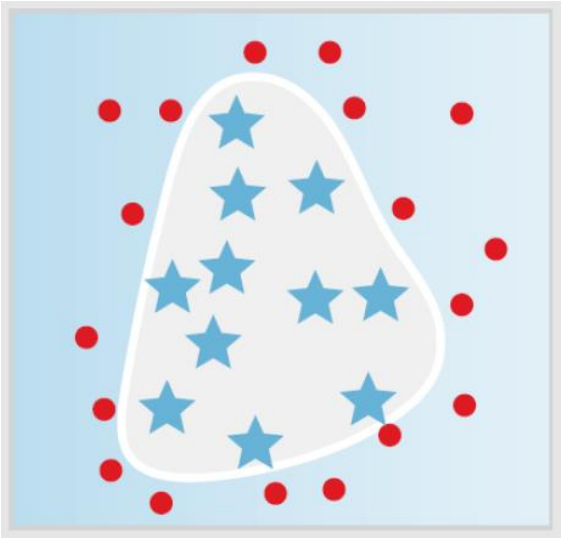
It can handle linearly separable problem

Adding more Neurals?



Linear vs. Non-Linear



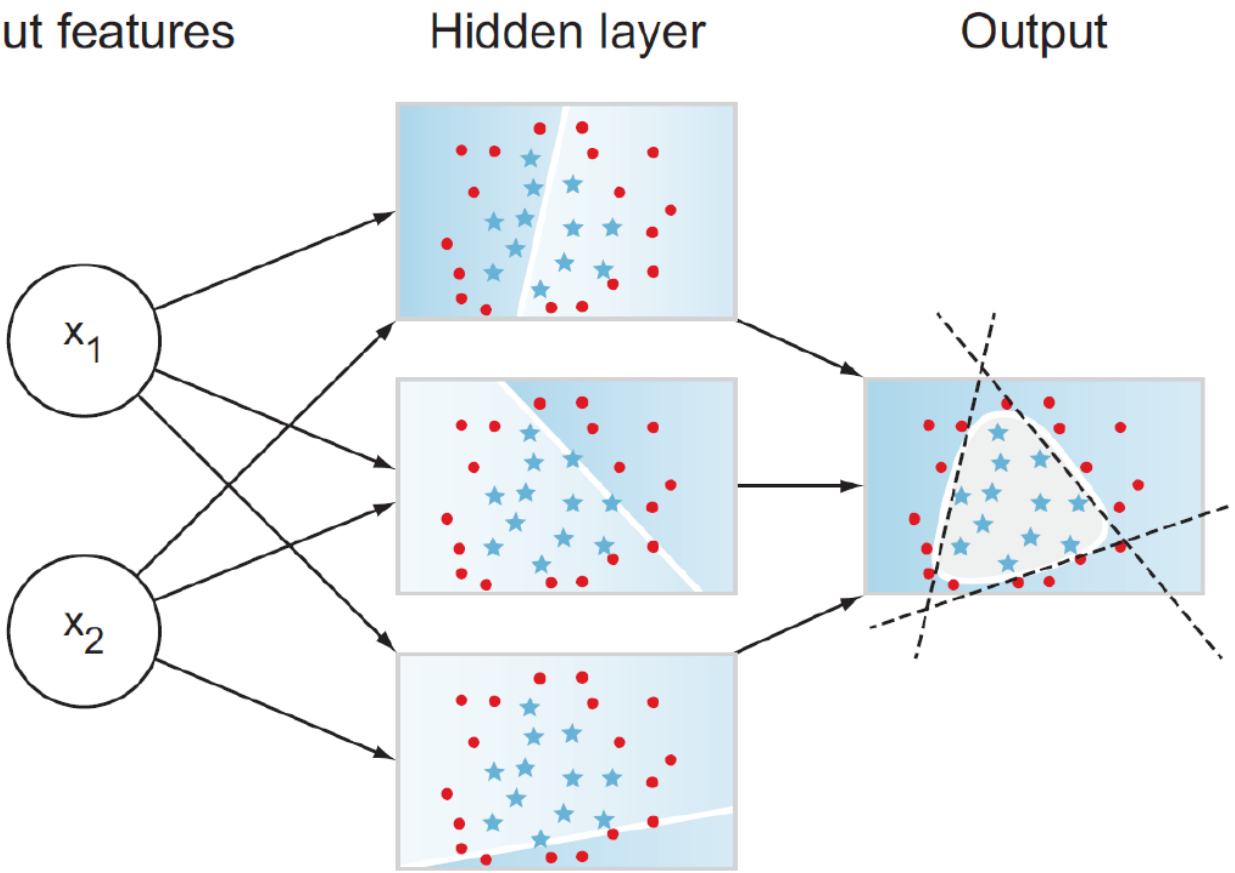


Linear

Non-Linear

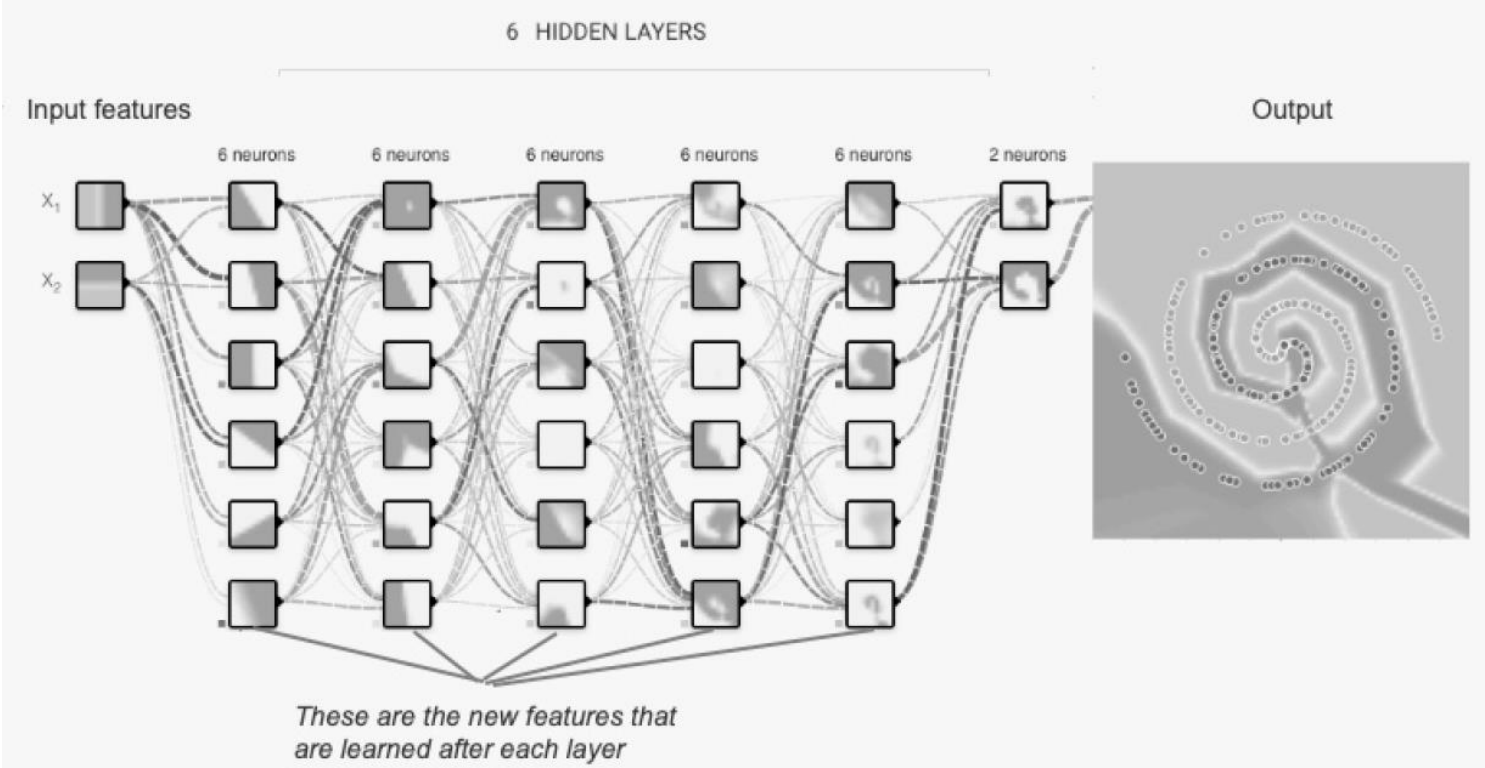
Multi-Layer Perceptron (MLP)

- Introduce the hidden layer



Multi-Layer Perceptron Architecture

- Can handle non-linear problem



Activation Function

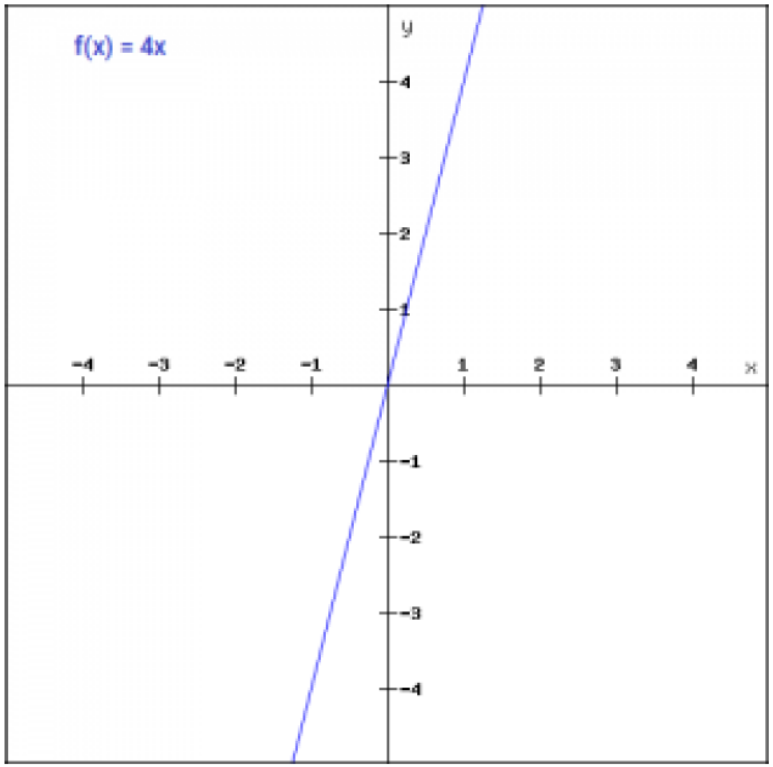
Also referred as transfer function or nonlinearities

List of activation function:

- Linear transfer function
- Step function
- Sigmoid/Logistic function
- Softmax function
- Hyperbolic Tangent Function (tanh)
- Rectified Linear Unit (ReLU)
- Leaky ReLU

Linear Transfer Function

activation(z) = z = wx + b

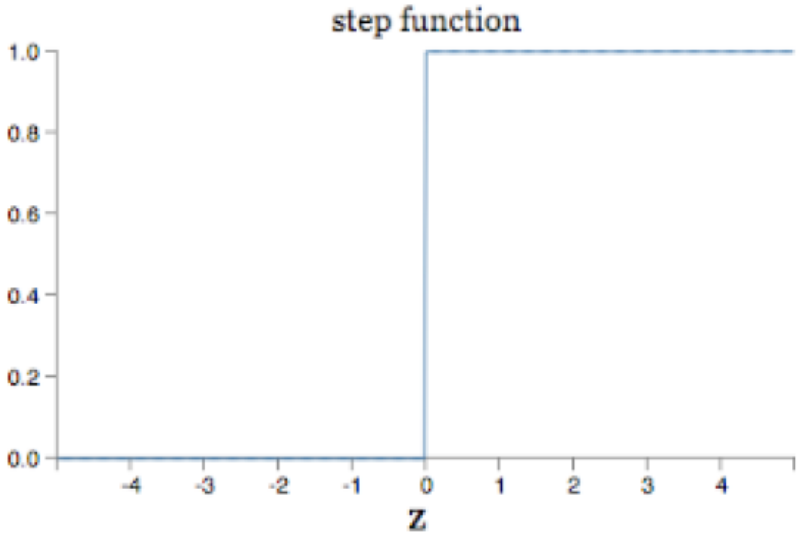


Step Function

Let y is the output

If the input x ≥ 0, y =1

else y = 0

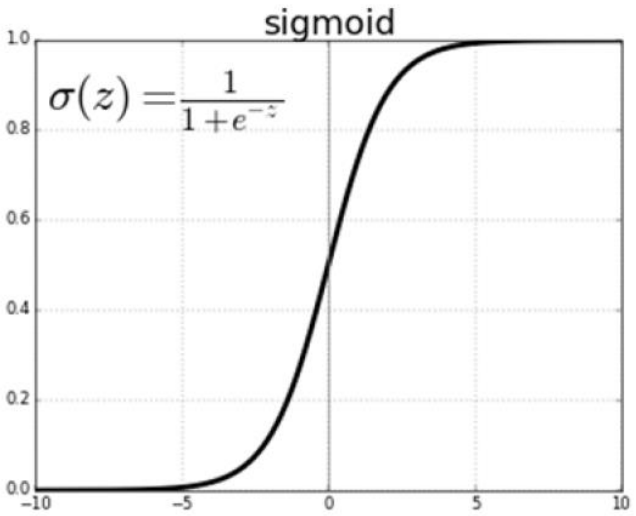


$$\text{output} = \begin{cases} 0 & \text{if } w \cdot x + b \leq 0 \\ 1 & \text{if } w \cdot x + b > 0 \end{cases}$$

Sigmoid/Logistic Function

- It is commonly used in binary classifiers
- It is also called S-shape curve

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



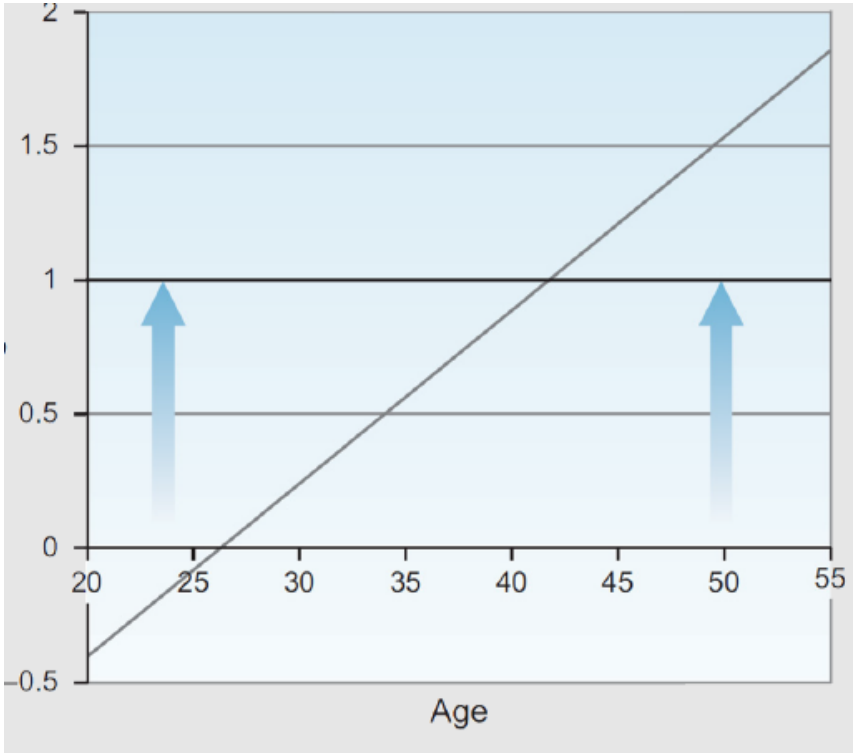

```python
# import numpy
import numpy as np

# sigmoid activation function
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
```

Sigmoid Function Explanation

- Consider you have medical condition of the patients having diabetes with only one feature age

Normalized
Number of patients



Sigmoid Function Explanation

- We don’t want negative probability
- We don’t want 38 and 43 to have the same probability

Exponential function can help

Softmax Function

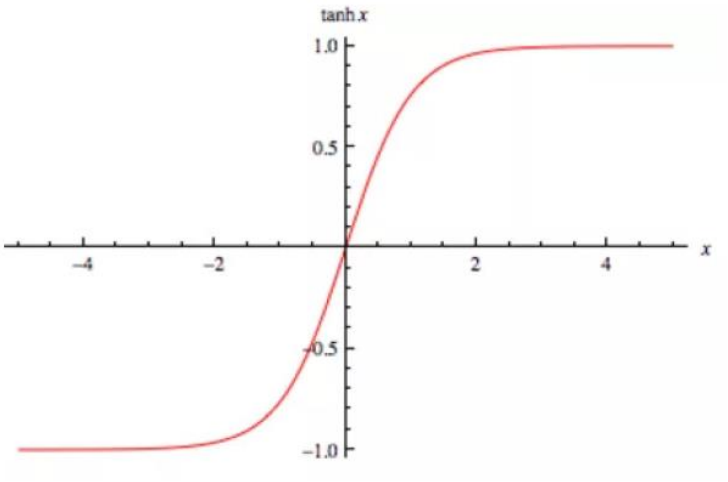
- Use in multi-class classification

$$\sigma(x_j) = \frac{e^{x_j}}{\sum_i e^{x_i}}$$

Hyperbolic Tangent Function(tanh)

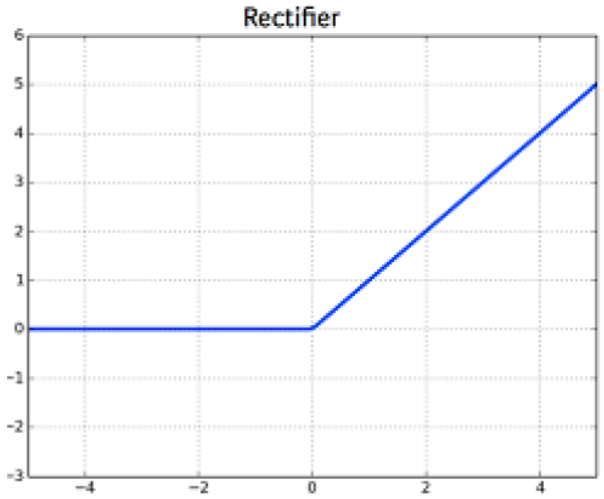
- Shifted version of sigmoid with value between -1 and 1
- It is usually used in hidden layers

$$tanh(x) = \frac{sinh(x)}{cosh(x)} = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$



Rectified Linear Unit (ReLU)

- Current state of the art of activation functions because of its simplicity



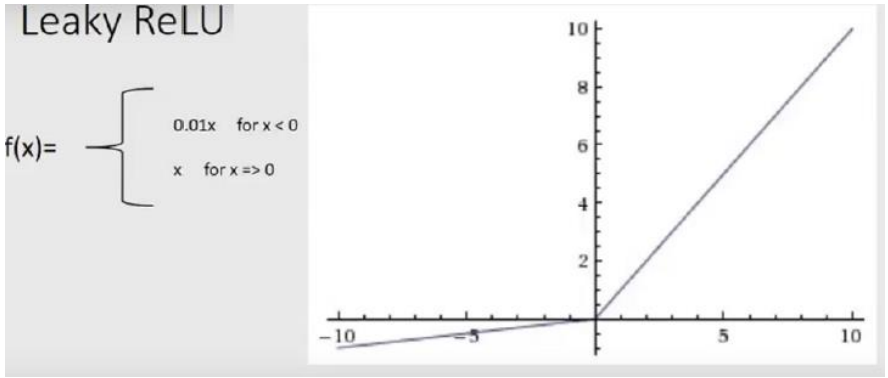
$$RELU(x) = \begin{cases} 0 \text{ if } x < 0 \\ x \text{ if } x >= 0 \end{cases}$$

Leaky LRU

- Provide some negative weight over ReLU

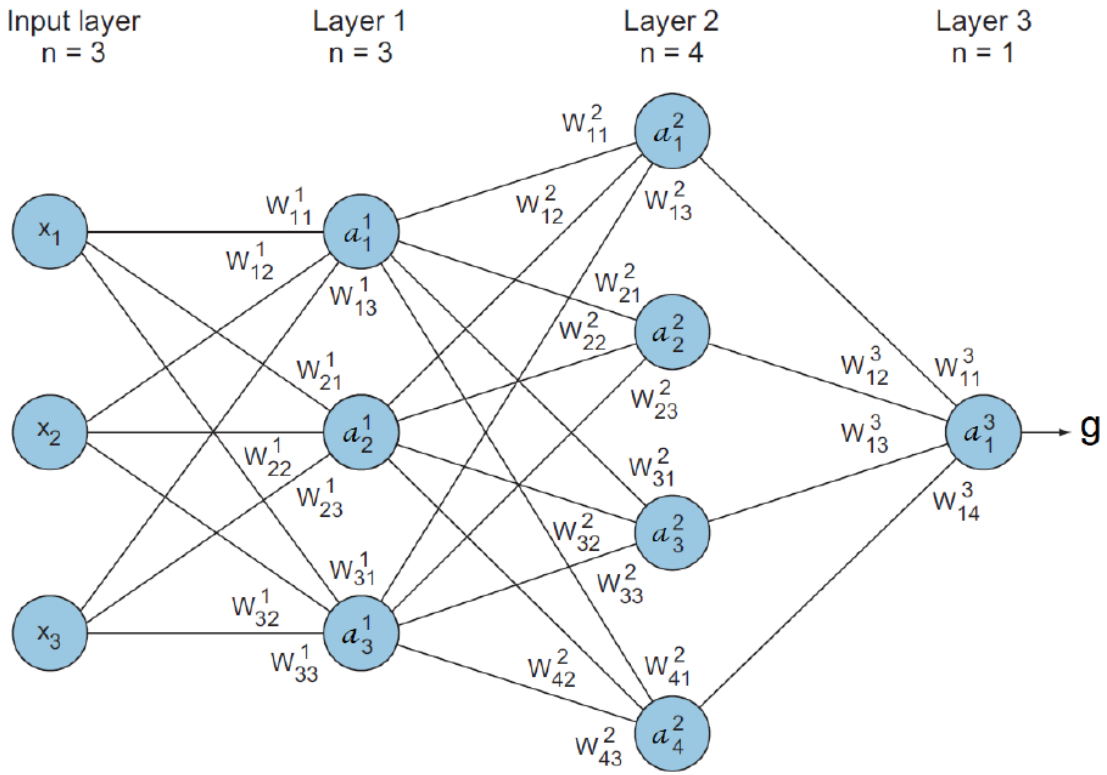
$$f(x) = max\ (0.01x, x)$$

```
# leaky relu activation function with
a 0.01 leak
def leaku_relu(x):
    if x < 0:
return x * 0.01
    else:
return x
```



Feedforward

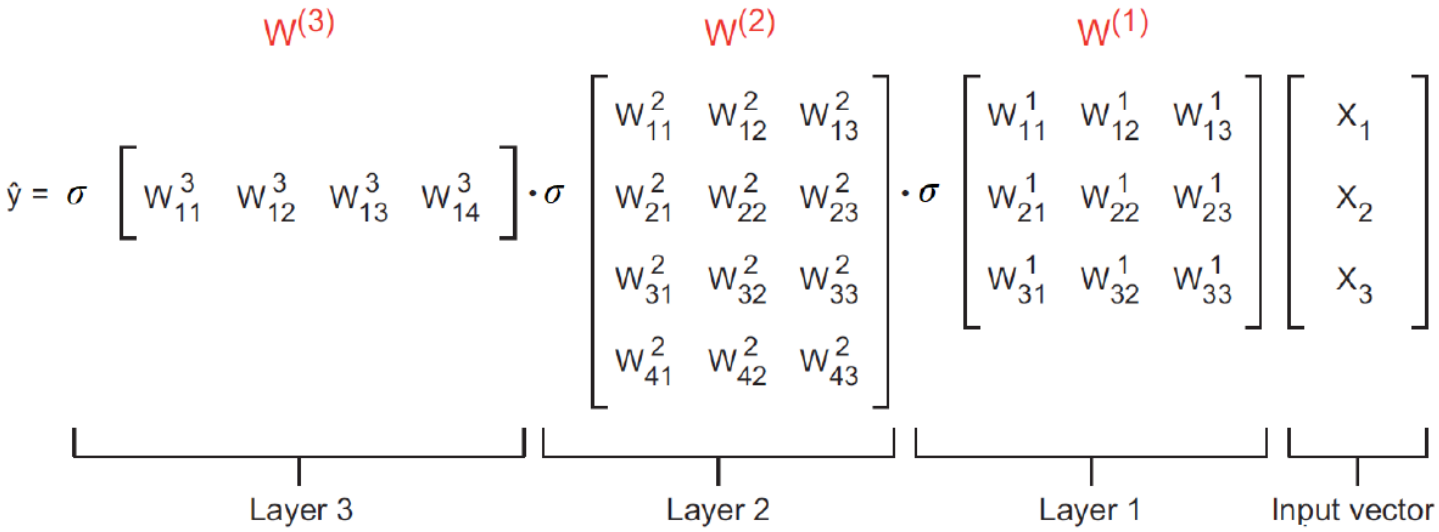
- The process of computing the linear combination and applying activation function is called Feedforward



Feedforward definition

- Layers:
- Weights and biases (w, b)
- Activation function (σ)
- Node value (a)

Feedforward calculation



Error Function

- It measures how wrong the neural network prediction is with respect to the expected output (the label)

- The error should always be positive (to avoid the error to cancel each other)

Mean Square Error

- It is commonly used in regression problems
- It is sensitive to the outliers

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

Notation	Meaning
$E\,(W,b)$	The loss function. Can be also annotated as $J\,(W,b)$ in other literature
W	Weights matrix. In some literature, the weights are denoted by the theta sign θ
b	Biases vector
N	Number of training examples
$\hat{y}i$	Prediction output. Also notated as $h_{W,\,b}(X)$ in some deep learning literature
yi	The correct output (the label)
$(\hat{y}i - yi)$	Usually called the residual

Mean Absolute Error

- Mean Absolute Error

$$E\,(W,b) = \frac{1}{N}\sum_{i=1}^{N} |\hat{y}_i - y_i|$$

- It is not continuous function

Cross Enthropy

- It is commonly used in classification problem

$$E\,(W,b) = -\sum_{i=1}^{m} y_i \, log(\,p_i)$$

- It amplifies the weight that are high probability

Example

- The real probability:

```
Probability(cat)        P(dog)          P(fish)
      0.0                1.0              0.0 .
```

- The first prediction:

```
Probability(cat)        P(dog)          P(fish)
      0.2                0.3              0.5
```

- The error function:

```
E = - (0.0 * log(0.2) + 1.0 * log(0.3) + 0.0 * log(0.5)) = 1.2
```

Example

- The second prediction

Probability(cat)	P(dog)	P(fish)
0.3	0.5	0.2

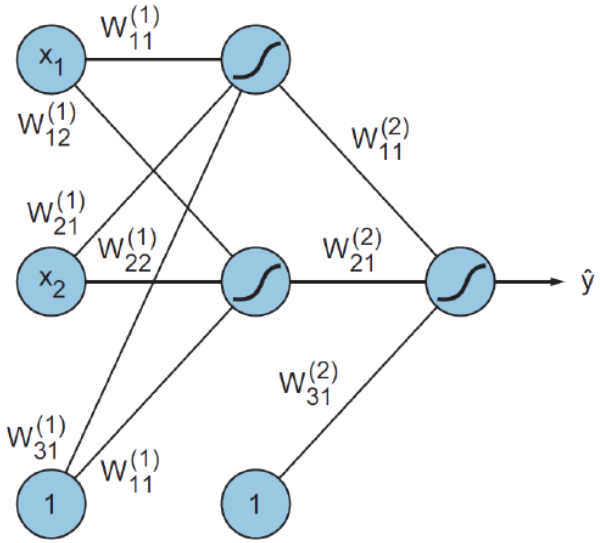
- The error function

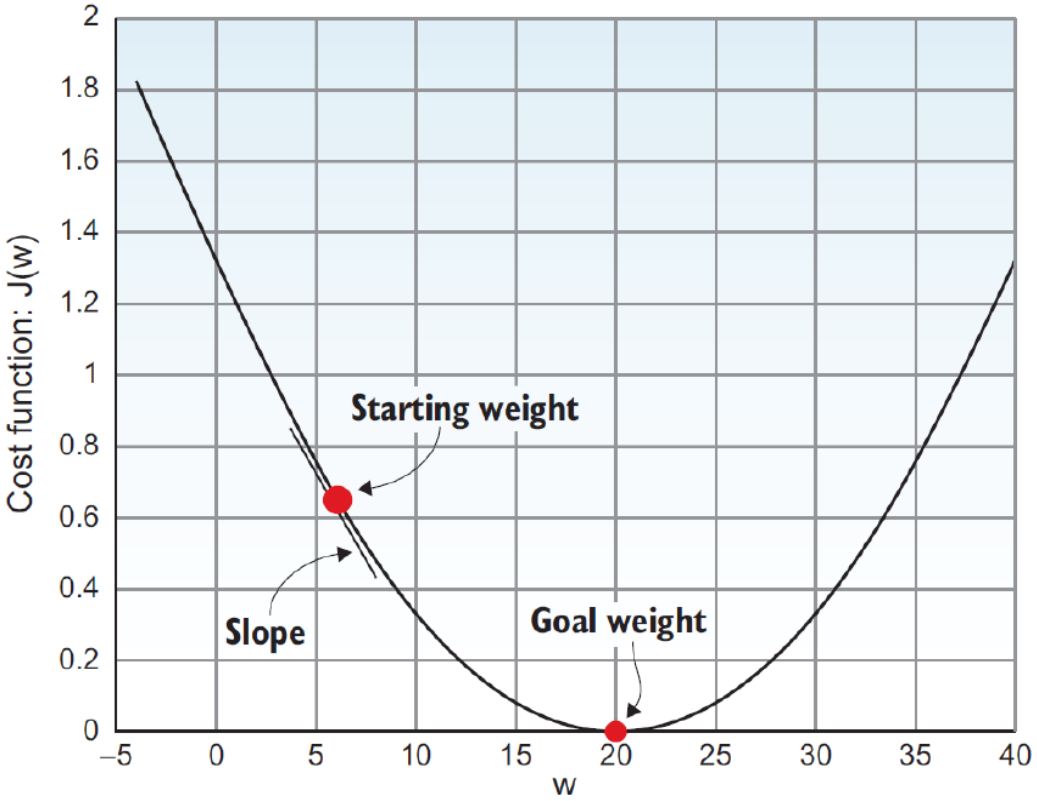
$$E = -(0.0*\log(0.3) + 1.0*\log(0.5) + 0.0*\log(0.2)) = 0.69$$

Optimization

- In neural networks, optimizing the error means updating the weights and biases until we find the optimal weights or the best values for weights to produce the minimum error

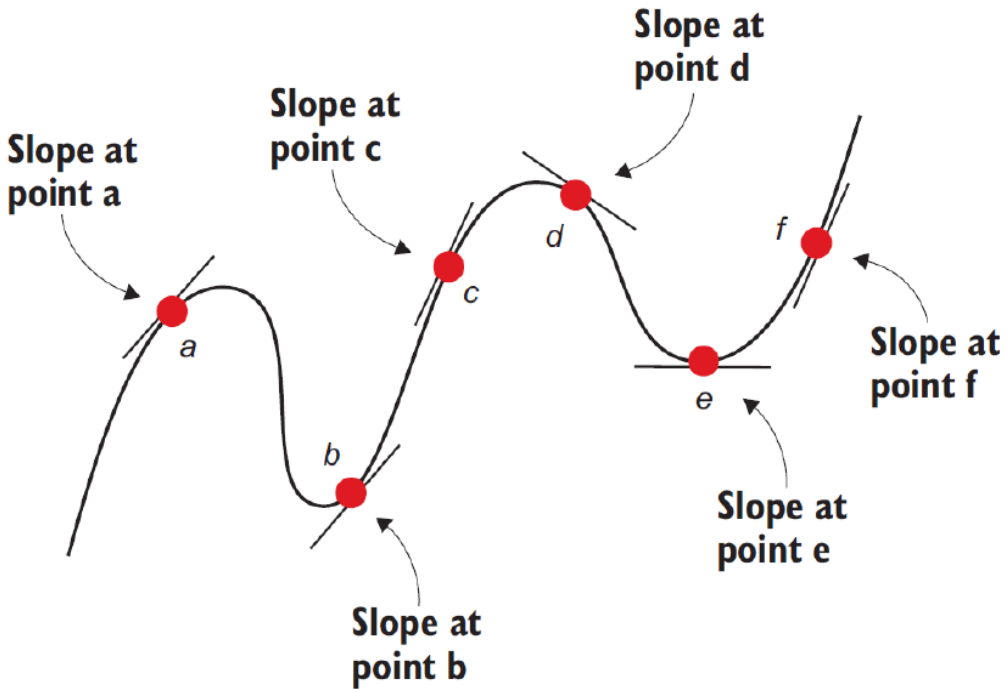
Weight Value





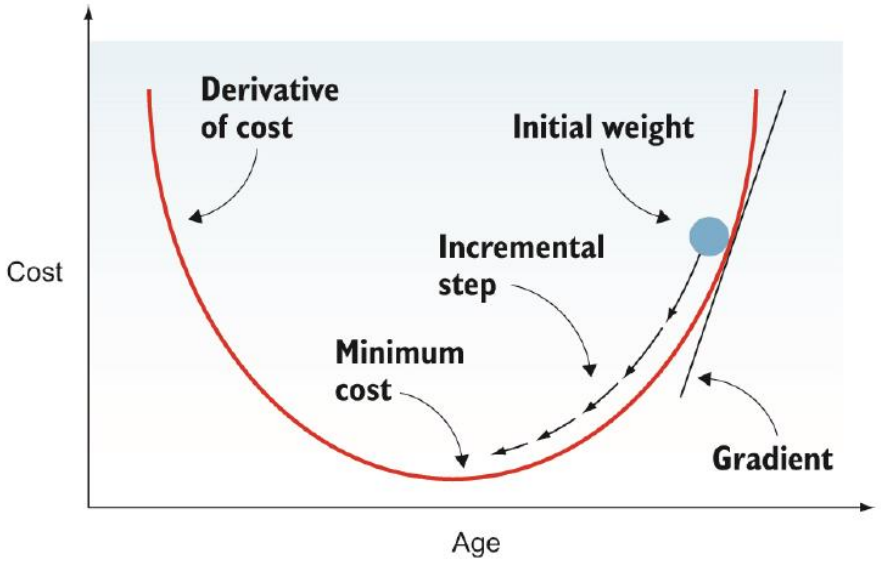
Batch Gradient Descent

- What is a gradient?



What is a gradient descent?

- Gradient descent means updating the weights iteratively to descent the slop of the error curve until we get the point with minimum error

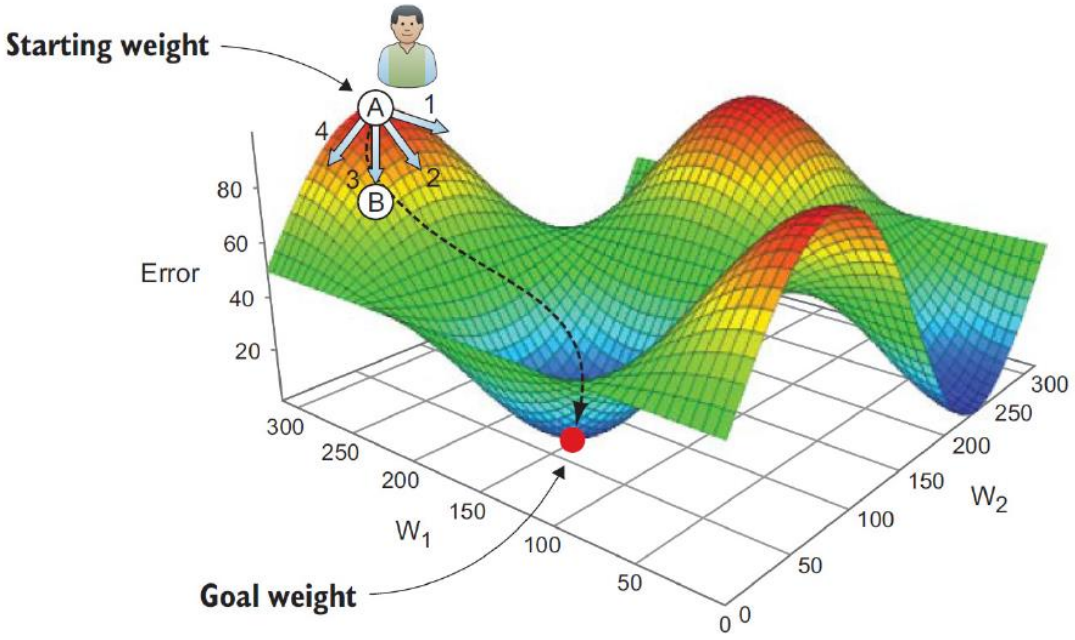


How does gradient descent work?

- The step direction (gradient)

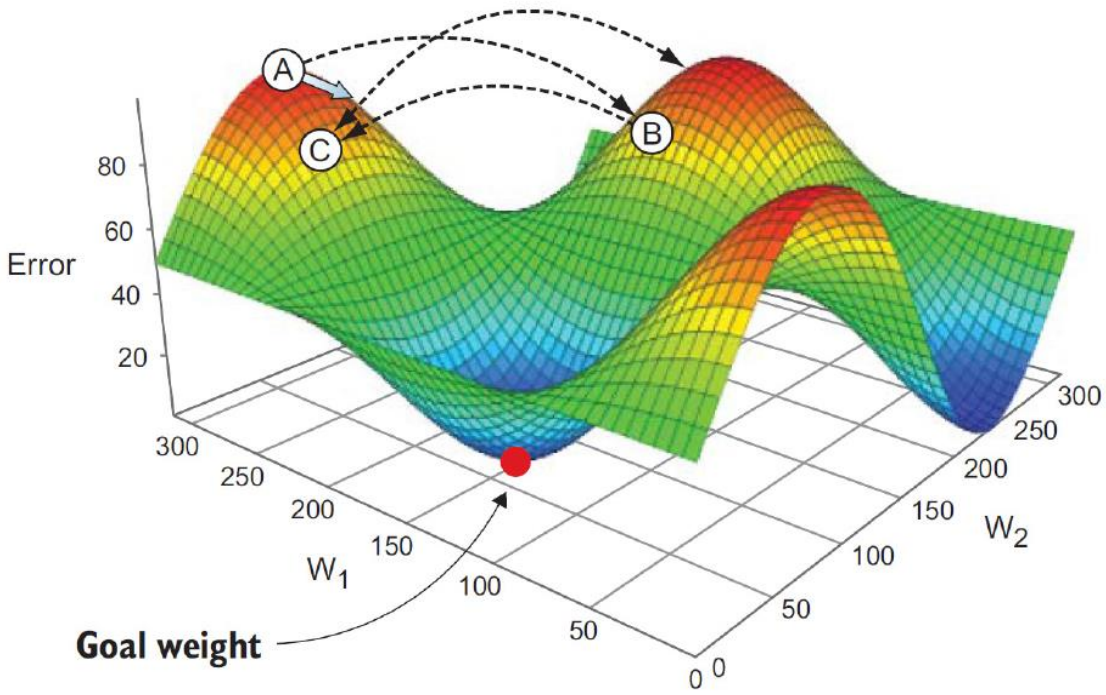
- The step size (learning rate)

Example


Starting weight
A
B
1
2
3
4
Error
80
60
40
20
300
250
200
150
100
50
0
W_1
W_2
Goal weight

The step size

- Impact of large step size



Gradient Descent

- Weight function

$$\Delta w_i = -\alpha \frac{dE}{dw_i}$$

- Weight update

$$w_{next-step} = w_{current} + \Delta w$$

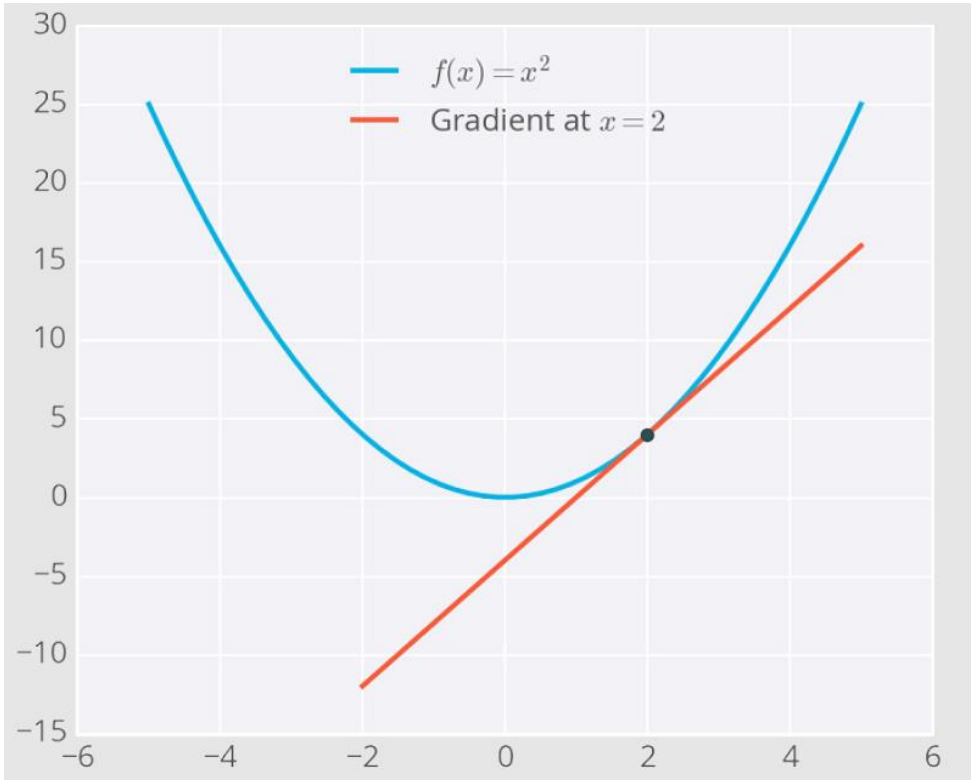
Weight update equation

```
error = ((input * weight) - goal_pred) ** 2
```

```
weight = weight - (alpha * derivative)
```

```
weight = weight - (input * (pred - goal_pred) * alpha)
```

Partial Derivative



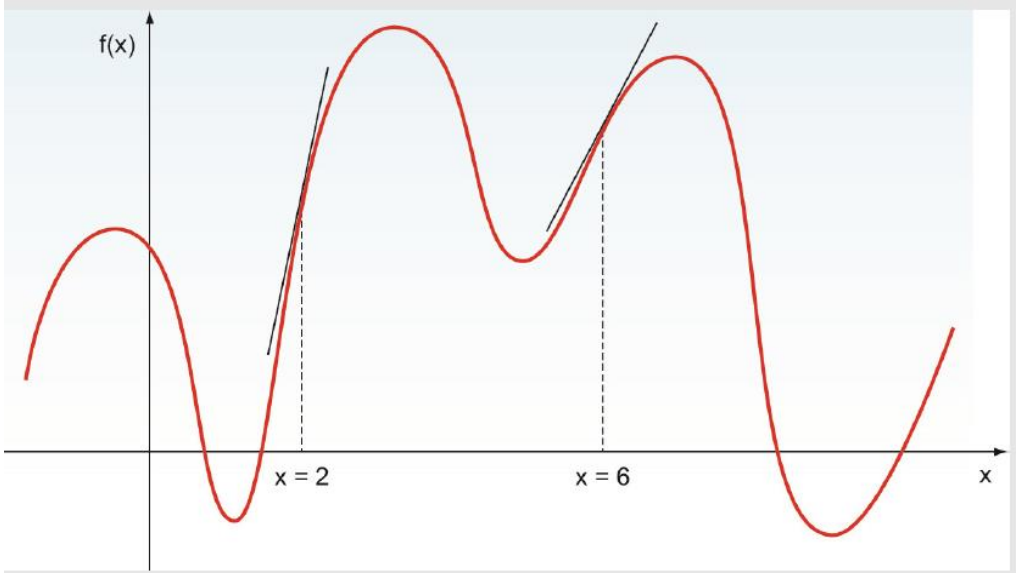
Derivative Rule

Constant Rule: $\frac{d}{dx}(c) = 0$	Difference Rule: $\frac{d}{dx}[f(x) - g(x)] = f'(x) - g'(x)$
Constant Multiple Rule: $\frac{d}{dx}[cf(x)] = cf'(x)$	Product Rule: $\frac{d}{dx}[f(x)g(x)] = f(x)g'(x) + g(x)f'(x)$
Power Rule: $\frac{d}{dx}(x^n) = nx^{n-1}$	Quotient Rule: $\frac{d}{dx}\left[\frac{f(x)}{g(x)}\right] = \frac{g(x)f'(x) - f(x)g'(x)}{[g(x)]^2}$
Sum Rule: $\frac{d}{dx}[f(x) + g(x)] = f'(x) + g'(x)$	Chain Rule: $\frac{d}{dx}f(g(x)) = f'(g(x))g'(x)$

Example

$$f(x) = 10\,x^5 + 4\,x^7 + 12\,x$$

$$f'(x) = 50\,x^4 + 28\,x^6 + 12$$



Batch Gradient Descent (BGD)

- It uses the entire training set to update the weight

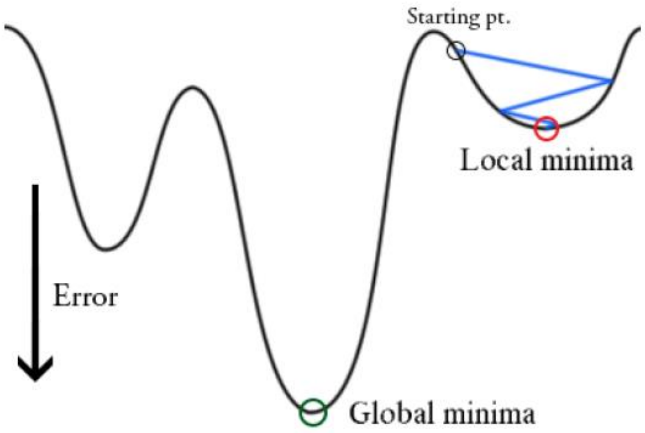
- The error function

$$L(W, b) = \frac{1}{N} \sum_{i=1}^{N} (\hat{y}_i - y_i)^2$$

- N is the total number of data in training set

Pitfall of Batch Gradient Descent

- Not all cost functions look like simple bowls



- To use the entire training set, the computation is very expensive and slow to train

Stochastic Gradient Descent (SGD)

- SGD is the most used optimization algorithms for machine learning
- SGD randomly picks one instance in the training set for each one stop and calculates the gradient based only on that single instance

Performance Comparison

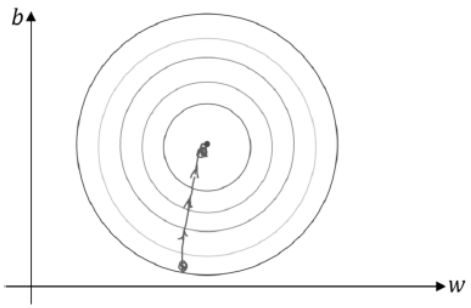
GD

1) Take ALL the data

2) Compute the gradient

3) Update the weights and take a step down

4) Repeat for n number of epochs (iterations)



Top View of the error mountain

Stochastic GD

1) randomly shuffle samples in the training set

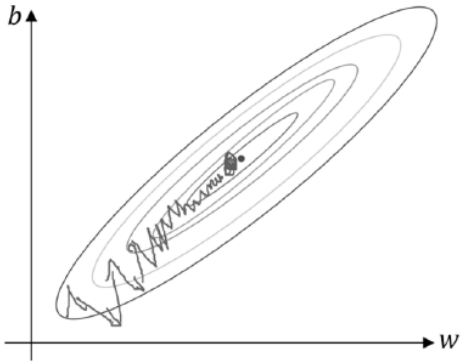
2) Pick one data instance

3) Compute the gradient

4) Update the weights and take a step down

5) Pick another one data instance

6) Repeat for n number of epochs (training iterations)



Top View of the error mountain

Mini-batch Gradient Descent (MN-GD)

- The compromise between Batch GD and Stochastic GD
- Group of training instead of a single instance
- It is faster comparing with BGD
- It reduces small error from SGD

Backpropagation

- Feedforward: get the linear combination and apply the activation function to get the output (y)

$$\hat{y} = W^{(3)} \circ \sigma \circ W^{(2)} \circ \sigma \circ W^{(1)} \circ \sigma \circ (x)$$

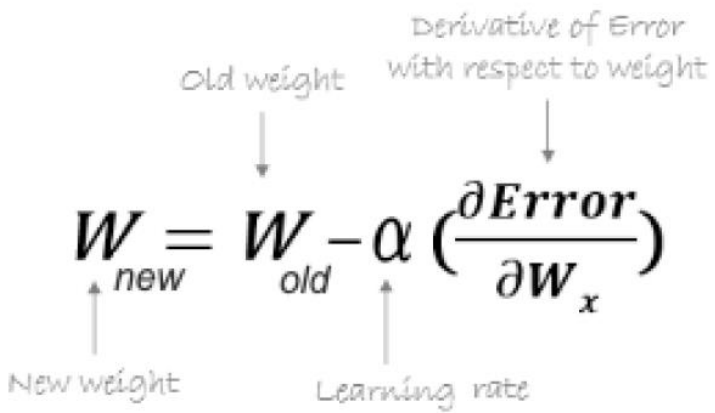
- Compare the prediction with the label to calculate the error or loss function

$$E\,(W, b) = \frac{1}{N} \sum_{i=1}^{N} |\hat{y}_i - y_i|$$

Backpropagation

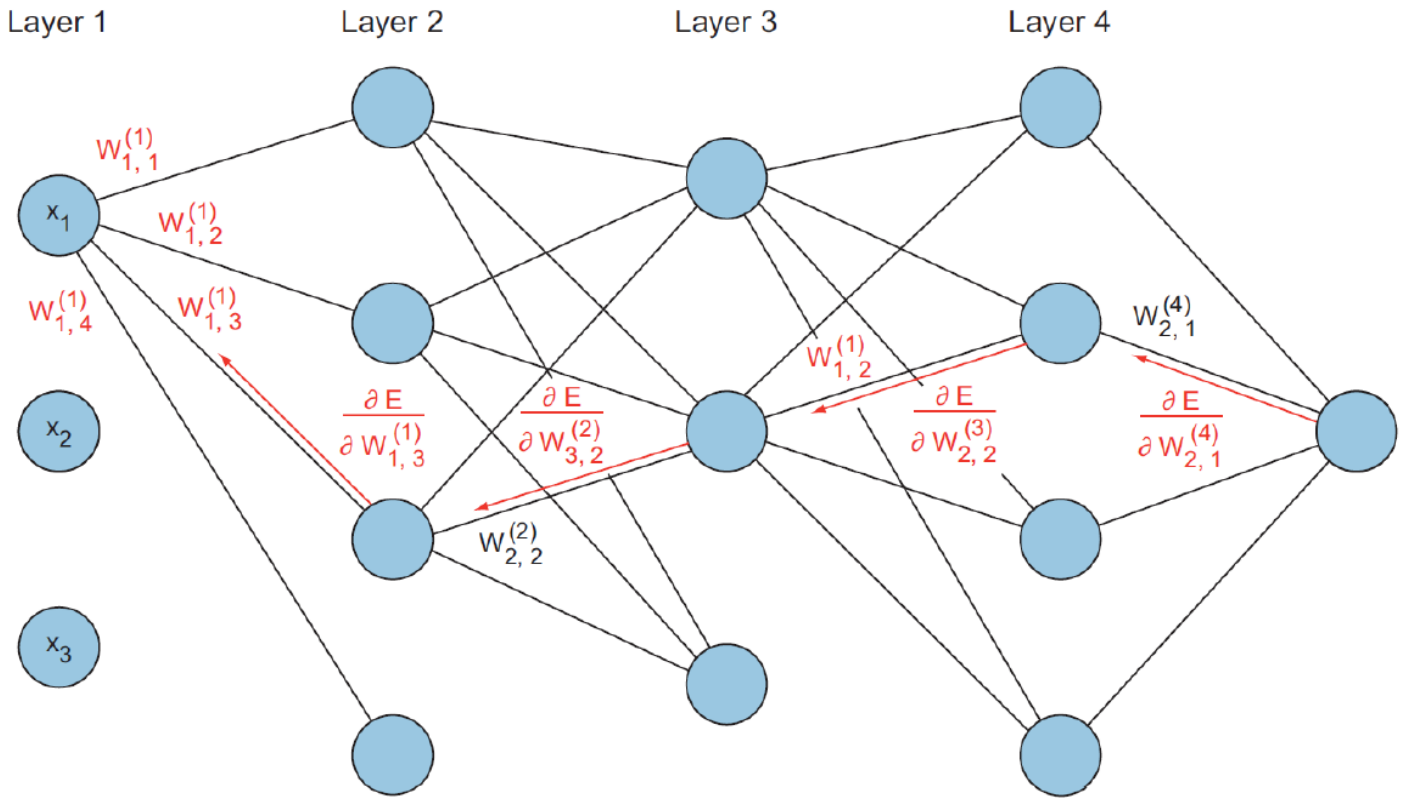
- Use gradient descent optimization algorithm to compute the weight update that optimizes the error function

$$\Delta w_i = -\alpha \frac{dE}{dw_i}$$



Backpropagation

- It is based on the chain rule



Chain Rule in Derivatives

- Chain Rule

$$\text{Chain Rule: } \frac{d}{dx} f(g(x)) = f'(g(x))g'(x)$$

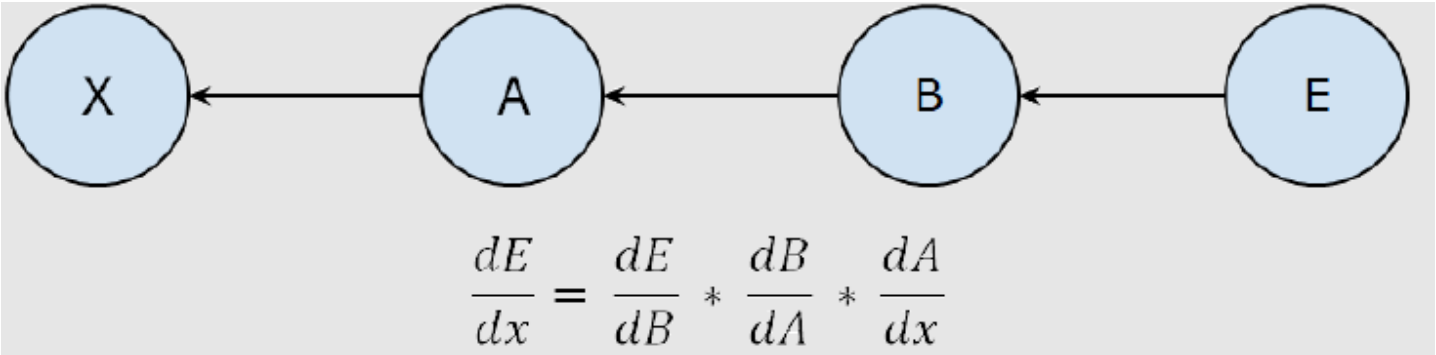
- The chain rule is a formula for calculating the derivatives of functions that are composed of functions inside other functions

$$\frac{d}{dx} f(g(x)) = \frac{d}{dx} \text{outside function} * \frac{d}{dx} \text{inside function}$$

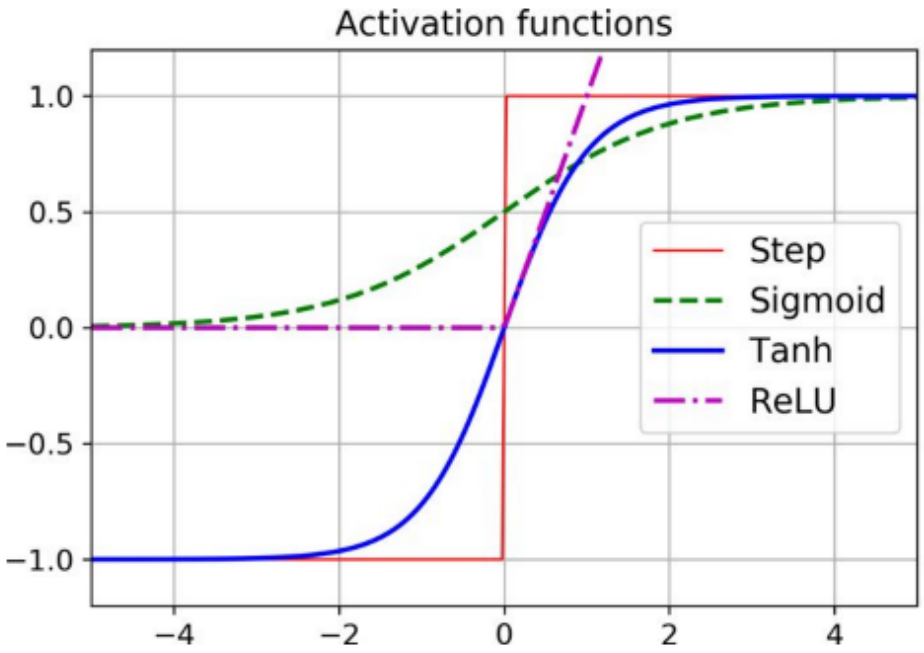
$$= \frac{d}{dx} f(g(x)) * \frac{d}{dx} g(x)$$

Example of Chain Rule

- We want to calculate dE/dx



Derivative function



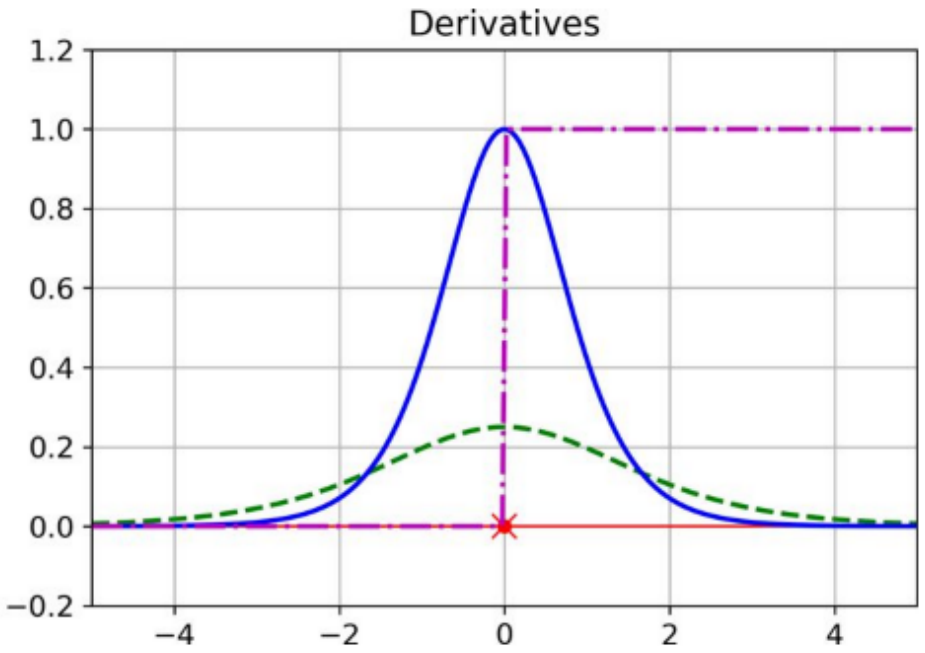


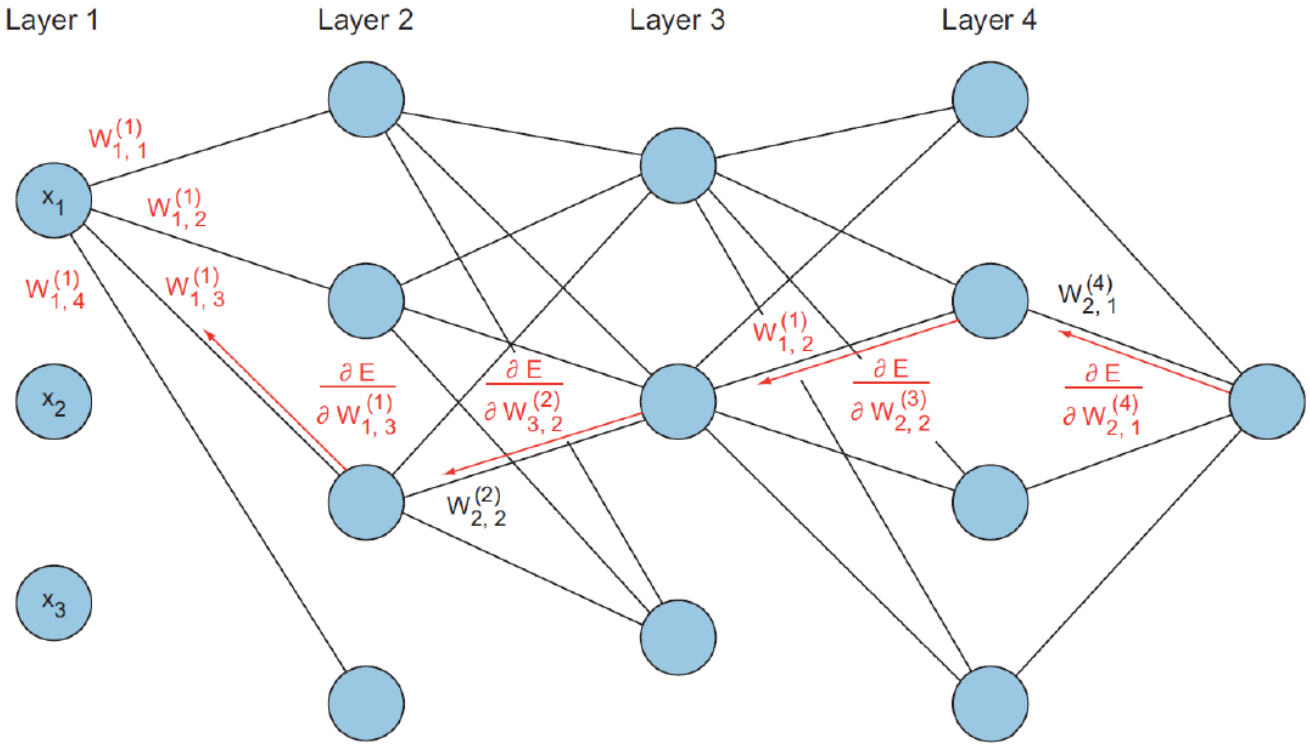
Figure 10-8. Activation functions and their derivatives

Python

Function	Forward prop	Backprop delta
relu	`ones_and_zeros = (input > 0)` `output = input*ones_and_zeros`	`mask = output > 0` `deriv = output * mask`
sigmoid	`output = 1/(1 + np.exp(-input))`	`deriv = output*(1-output)`
tanh	`output = np.tanh(input)`	`deriv = 1 - (output**2)`
softmax	`temp = np.exp(input)` `output /= np.sum(temp)`	`temp = (output - true)` `output = temp/len(true)`

Backpropagation Example

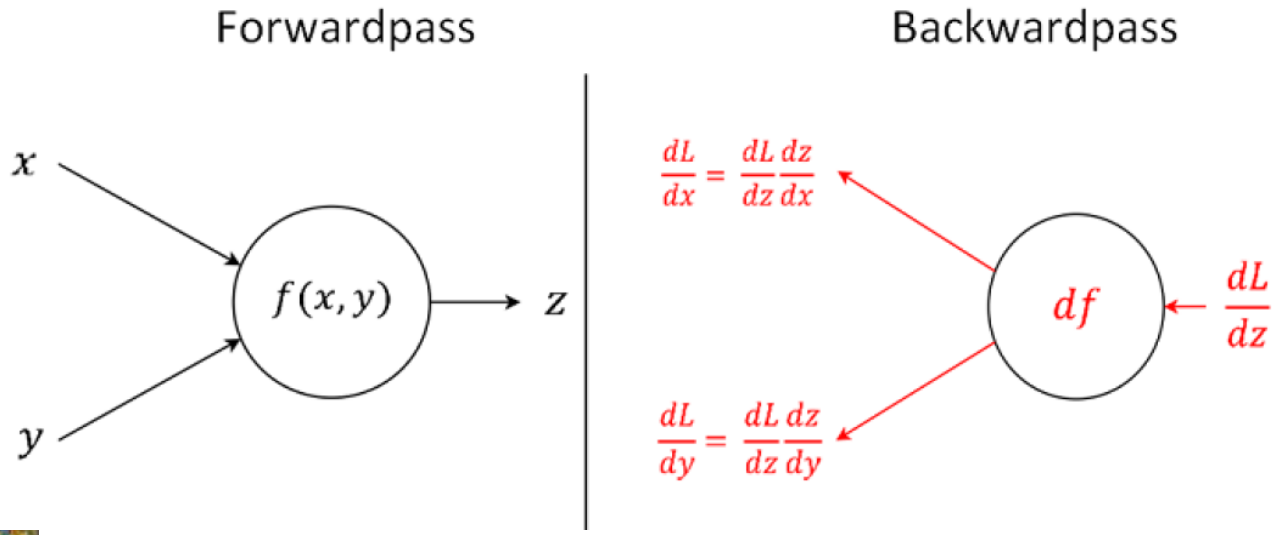
The error backpropagated to the edge $w^{(1)}_{1,3}$ = *effect of error on edge 4* $*$ *effect on edge 3* $*$ *effect on edge 2* $*$ *effect on target edge*



$$\frac{dE}{dw^{(1)}_{1,3}} = \frac{dE}{dw^{(4)}_{2,1}} * \frac{dw^{(4)}_{2,1}}{dw^{(3)}_{2,2}} * \frac{dw^{(3)}_{2,2}}{dw^{(2)}_{3,2}} * \frac{dw^{(2)}_{3,2}}{dw^{(1)}_{1,3}}$$

Backpropagation Summary

- Forward pass is to calculate predicted output
- Backward propagation is to update the weight to `error = ((input * weight) - goal_pred) ** 2`



Weight update equation

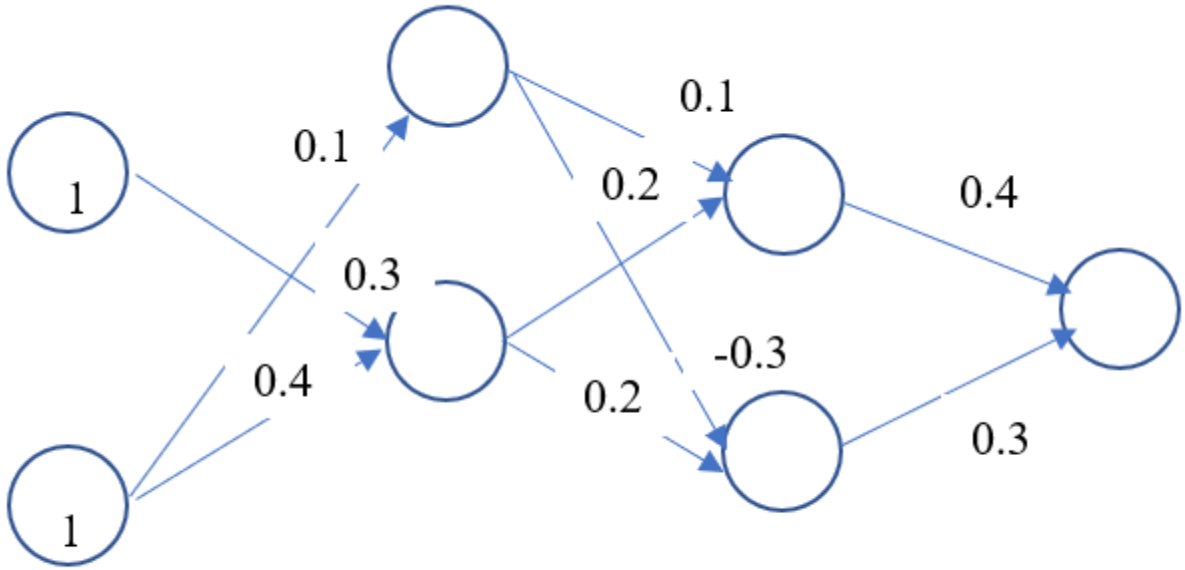
```
error = ((input * weight) - goal_pred) ** 2
```

```
weight = weight - (alpha * derivative)
```

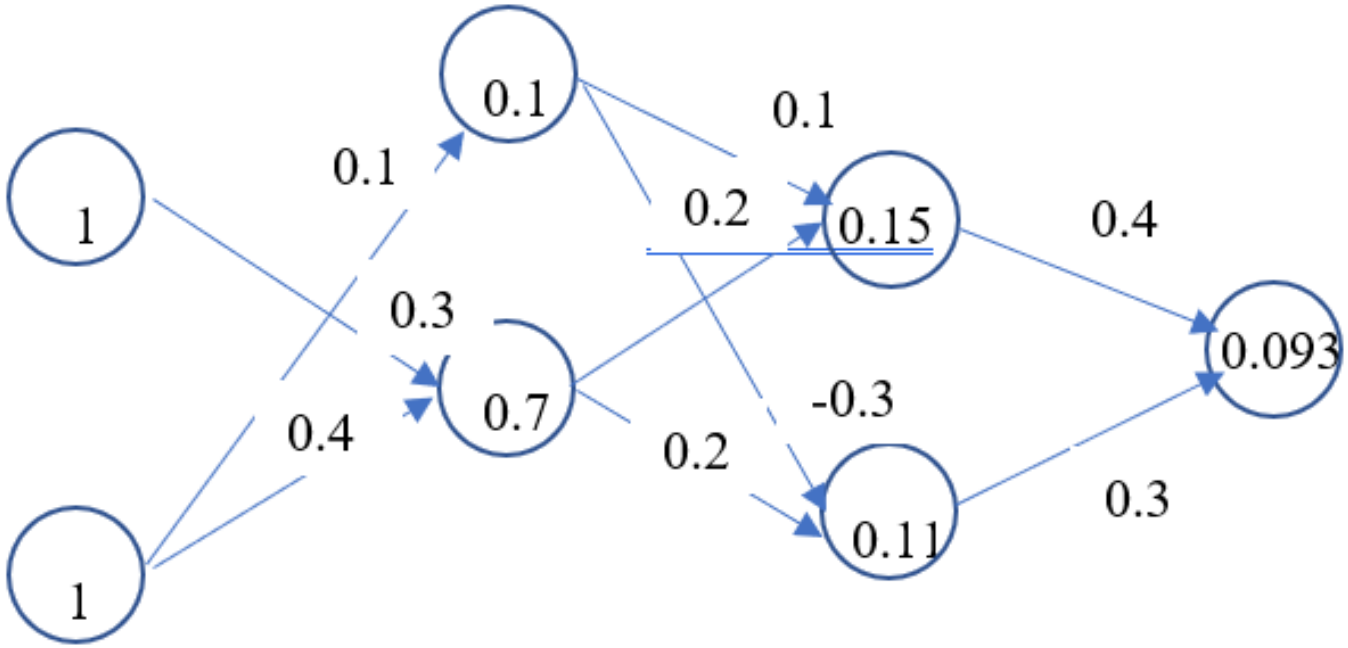
```
weight = weight - (input * (pred - goal_pred) * alpha)
```

Example

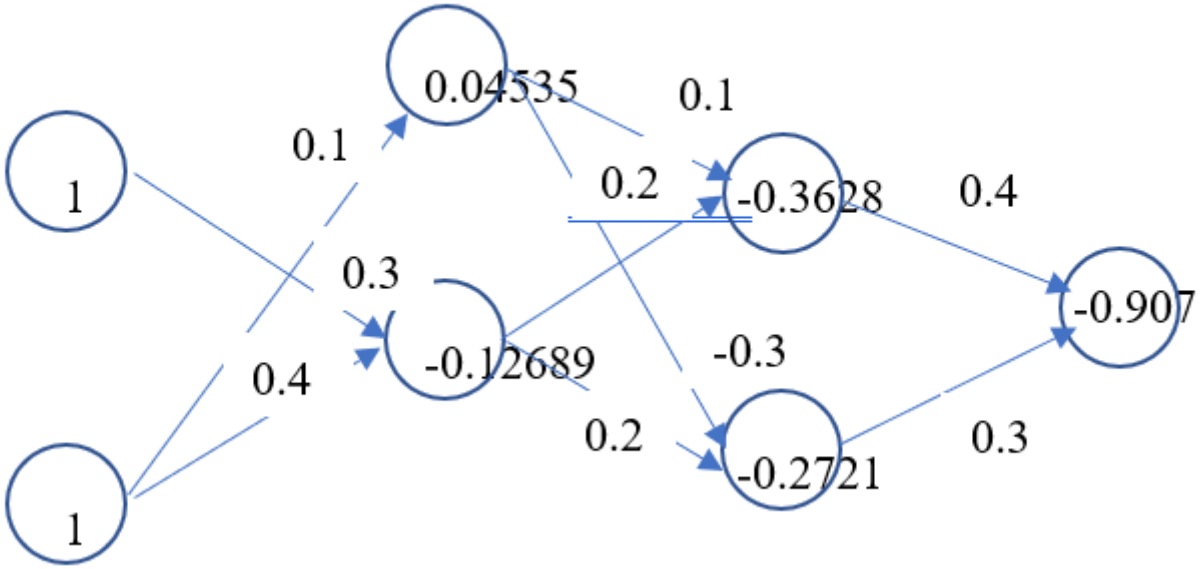
- Compute the feed forward and back propagation for 1 iteration of weight update



Feedforward



Backward Delta



- 1 – 0.093 = 0.907

0.907 * 0.4= - 0.3628

-0.3628*0.1 -0.2721* (-0.3) = 0.04535

-0.3628* 0.2 -0.2721*0.2 = - 0.12689

Weight update

Level 1

0 – 1 * 0.04535 = - 0.04535

0.1 – 1 *0.04535 = 0.05465

0.3 + 1 * 0.12698 = 0.42698

0.4 + 1 *0.12698 = 0.52698

Level 2

0.1 + 0.1* 0.3628 = 0.13628

0.2 + 0.7*0.3628 = 0.45396

-0.3 + 0.1 *0.2721 = -0.2729

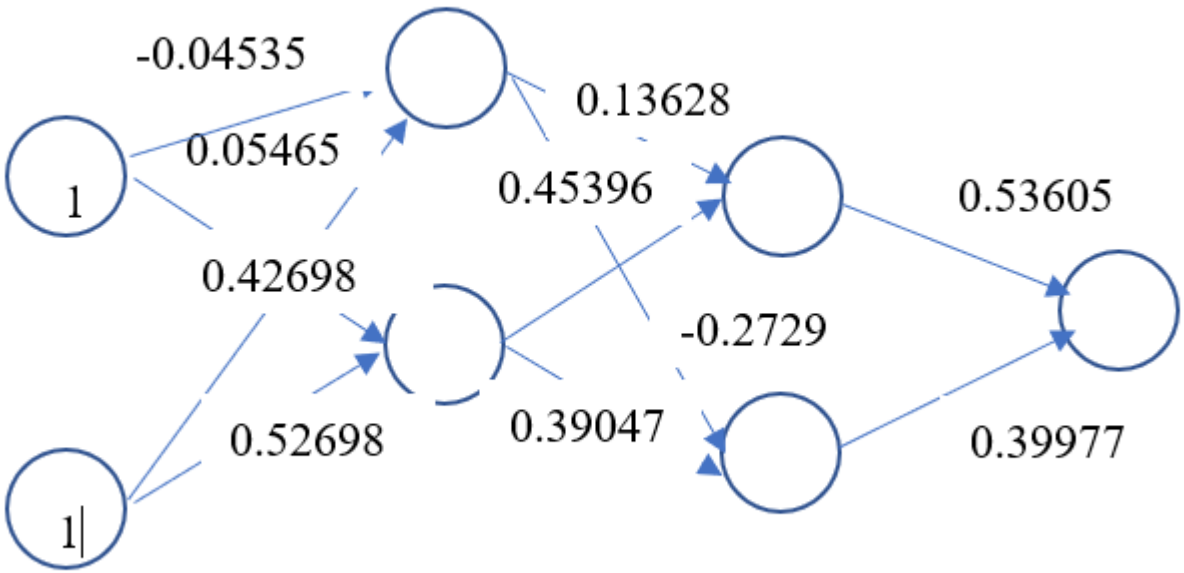
0.2 + 0.7 * 0.2721 = 0.39047

Level 3

0.4 + 0.15*0.907 = 0.53605

0.3 + 0.11 * 0.907 = 0.39977

Backward Weight update



Python program example

```
import numpy as np
np.random.seed(1)
 def relu(x):
    return (x > 0) * x # returns x if x > 0
                       # return 0 otherwise
 def relu2deriv(output):
    return output>0 # returns 1 for input > 0
                    # return 0 otherwise
input1 = np.array( [[ 1, 1 ],
                    [ 1, 1 ]] )
 output1 = np.array([[ 1, 1]]).T
 alpha = 1
hidden_size = 3
print(input1.shape)
```

```
weights_0_1 = np.array( [[ 0, 0.3 ],
                         [ 0.1, 0.4]] )
print(weights_0_1.shape)
weights_1_2 = np.array( [[ 0.1, -0.3 ],
                         [ 0.2, 0.2]] )
print(weights_1_2.shape)
weights_2_3 = np.array( [[ 0.4],[ 0.3 ]] )
print(weights_2_3.shape)
```

```
for iteration in range(1):
    output_error = 0
    for i in range(len(input1)):
        layer_0 = input1[i:i+1]
        print("layer 0",layer_0.shape)
        layer_1 = relu(np.dot(layer_0,weights_0_1))
        layer_2 = relu(np.dot(layer_1,weights_1_2))
        output = np.dot(layer_2,weights_2_3)
        output_error += np.sum((output - output1[i:i+1]) ** 2)
        output_delta =  (output - output1[i:i+1])
        print("delta output:",output_delta)
        layer_2_delta =
output_delta.dot(weights_2_3.T)*relu2deriv(layer_2)
        layer_1_delta =l
ayer_2_delta.dot(weights_1_2.T)*relu2deriv(layer_1)
        weights_2_3 -= alpha * layer_2.T.dot(output_delta)
        weights_1_2 -= alpha * layer_1.T.dot(layer_2_delta)
        weights_0_1 -= alpha * layer_0.T.dot(layer_1_delta)
```

Questions?