Artificial Neural Network Forward Propagation

Dr. Mongkol Ekpanyapong

Hello World in Deep Learning

```
from tensorflow.keras.datasets import mnist
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
test_images.shape
len(test_labels)
from tensorflow import keras
from tensorflow.keras import layers
model = keras.Sequential([
layers.Dense(512, activation="relu"),
layers.Dense(10, activation="softmax")
])
model.compile(optimizer="sgd",
loss="sparse_categorical_crossentropy",
metrics=["accuracy"])
train_images = train_images.reshape((60000, 28 * 28))
train_images = train_images.astype("float32") / 255
test_images = test_images.reshape((10000, 28 * 28))
test_images = test_images.astype("float32") / 255
model.fit(train_images, train_labels, epochs=5, batch_size=128)
```

```
import matplotlib.pyplot as plt
(train_images_ori, train_labels), (test_images_ori, test_labels) = mnist.load_data()

for i in range(10):
    plt.subplot(10,10,i+1)
    plt.imshow(test_images_ori[i])
plt.show()

test_digits = test_images[0:10]
predictions = model.predict(test_digits)
# print("Prediction probability",predictions)

for i in range(10):
    print("Final Prediction ",i, " ",predictions[i].argmax(), " ", test_labels[i])

test_loss, test_acc = model.evaluate(test_images, test_labels)
print(f"test_acc: {test_acc}")
```

Homework

- Perform Image Classification using ANN on CIFAR 10

```
((trainX, trainY), (testX, testY)) = cifar10.load_data()
trainX = trainX.astype("float") / 255.0
testX = testX.astype("float") / 255.0
lb = LabelBinarizer()
trainY = lb.fit_transform(trainY)
testY = lb.transform(testY)
```

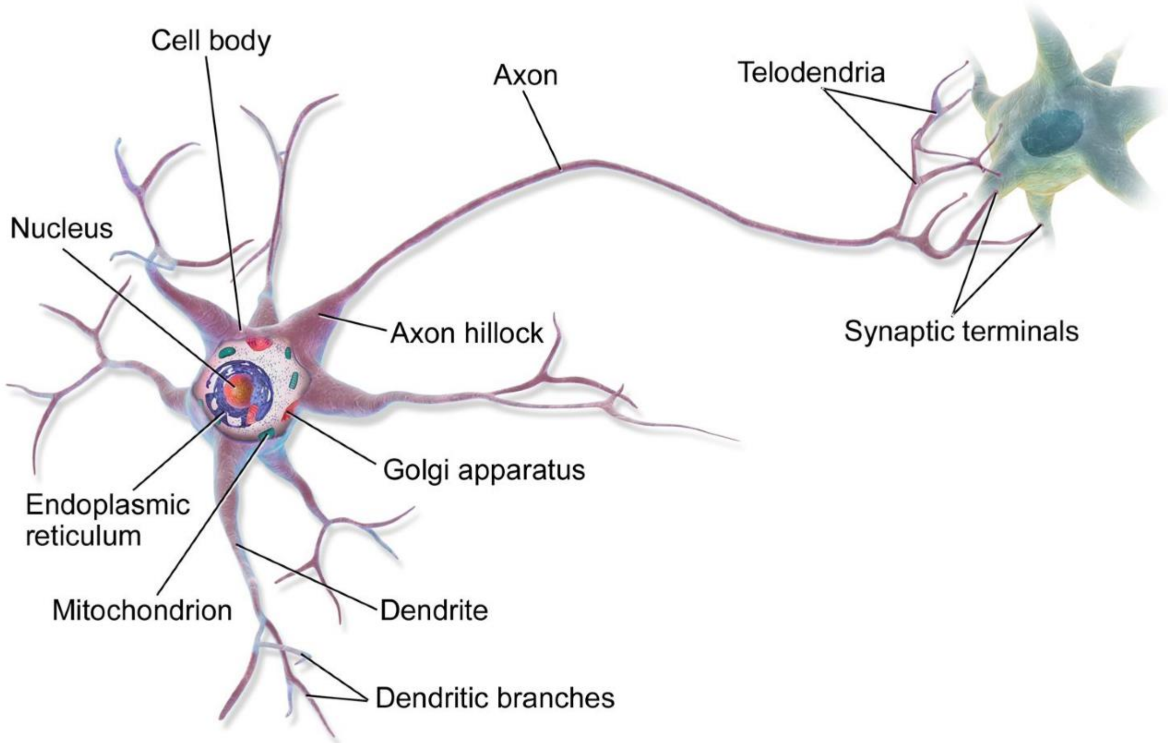
Black Block model

- 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

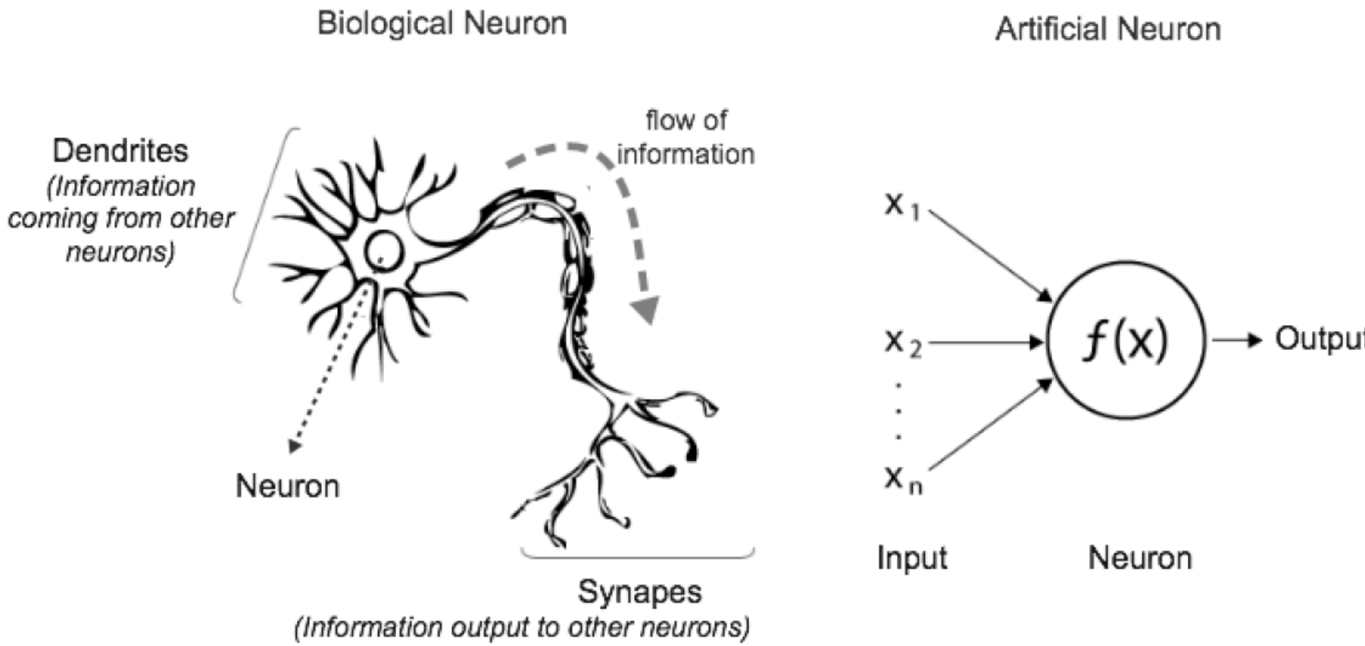
Perceptron Model

- 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

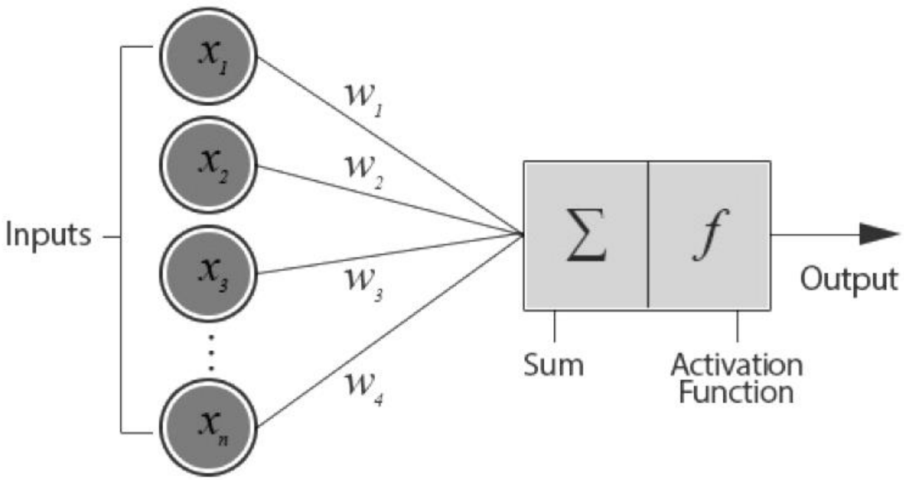
Perceptron Model



Model Comparison



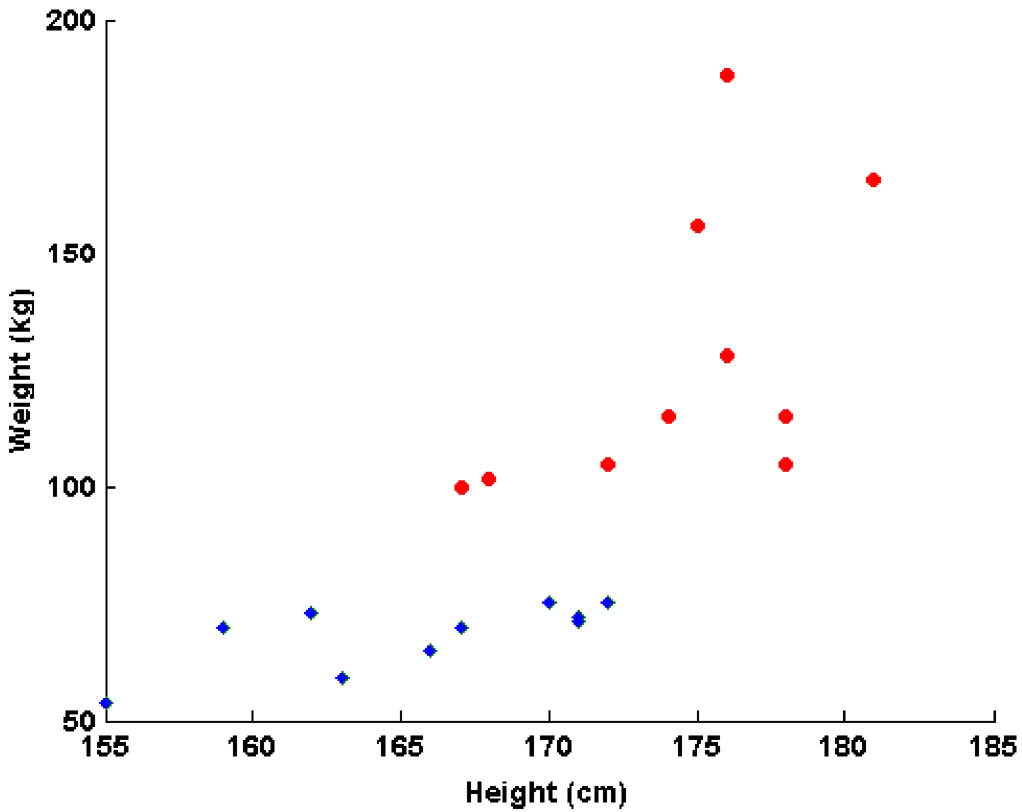
ANN Model



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

Patterns and pattern classes

- Sumo wrestlers and table tennis players



Weighted Sum Function

- Linear combination can be used for weight calculation

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

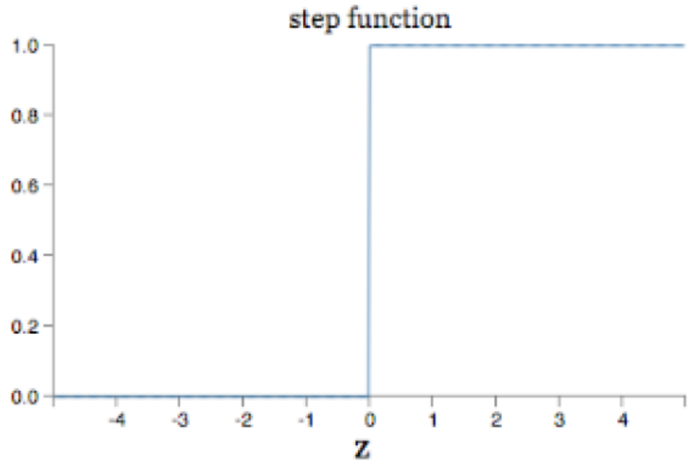
$$z = x_1 \; . w_1 + x_2 . w_2 + x_3 . w_3 + \ldots . + x_n \; . w_n \; + b$$

- 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



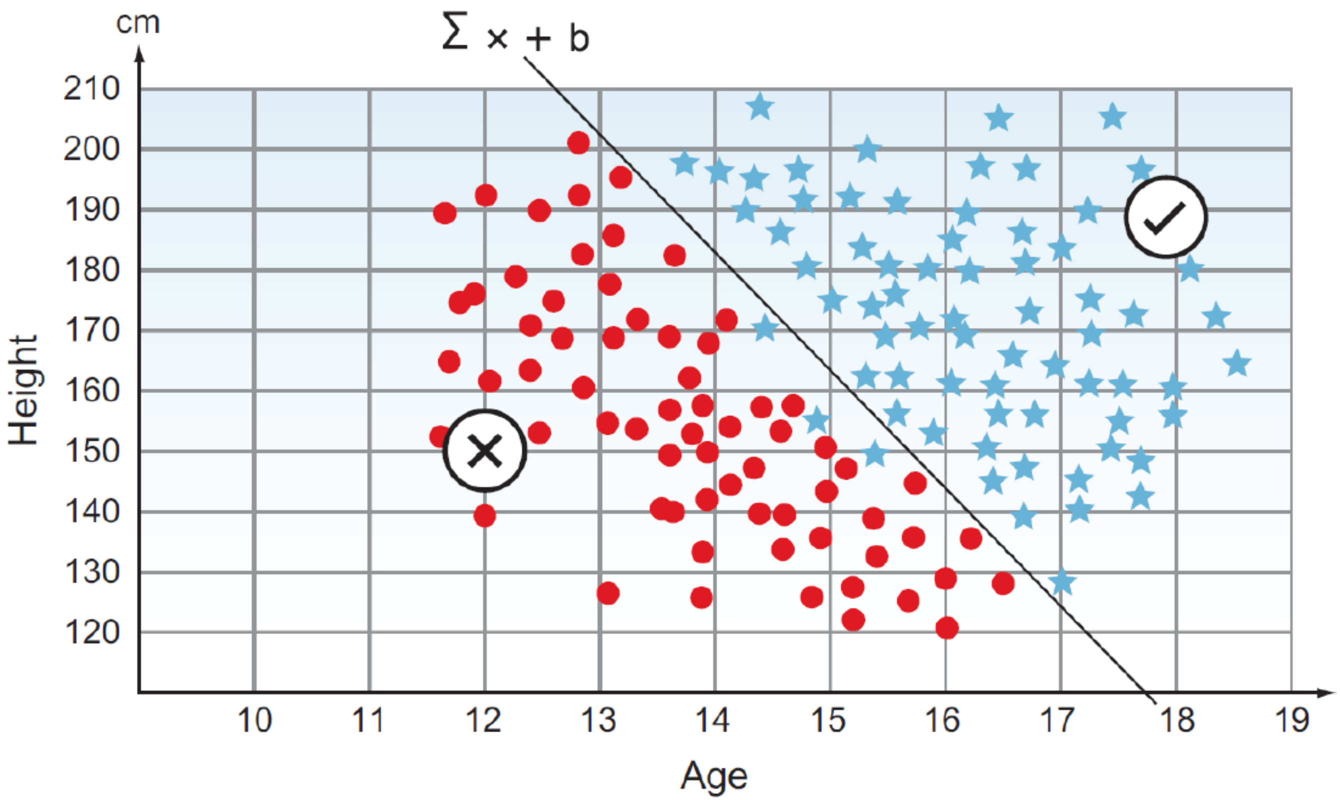
$$\text{output} = \begin{cases} 0 & \text{if } w \cdot x + b \leq 0 \\ 1 & \text{if } w \cdot x + b > 0 \end{cases}$$

```
# 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?

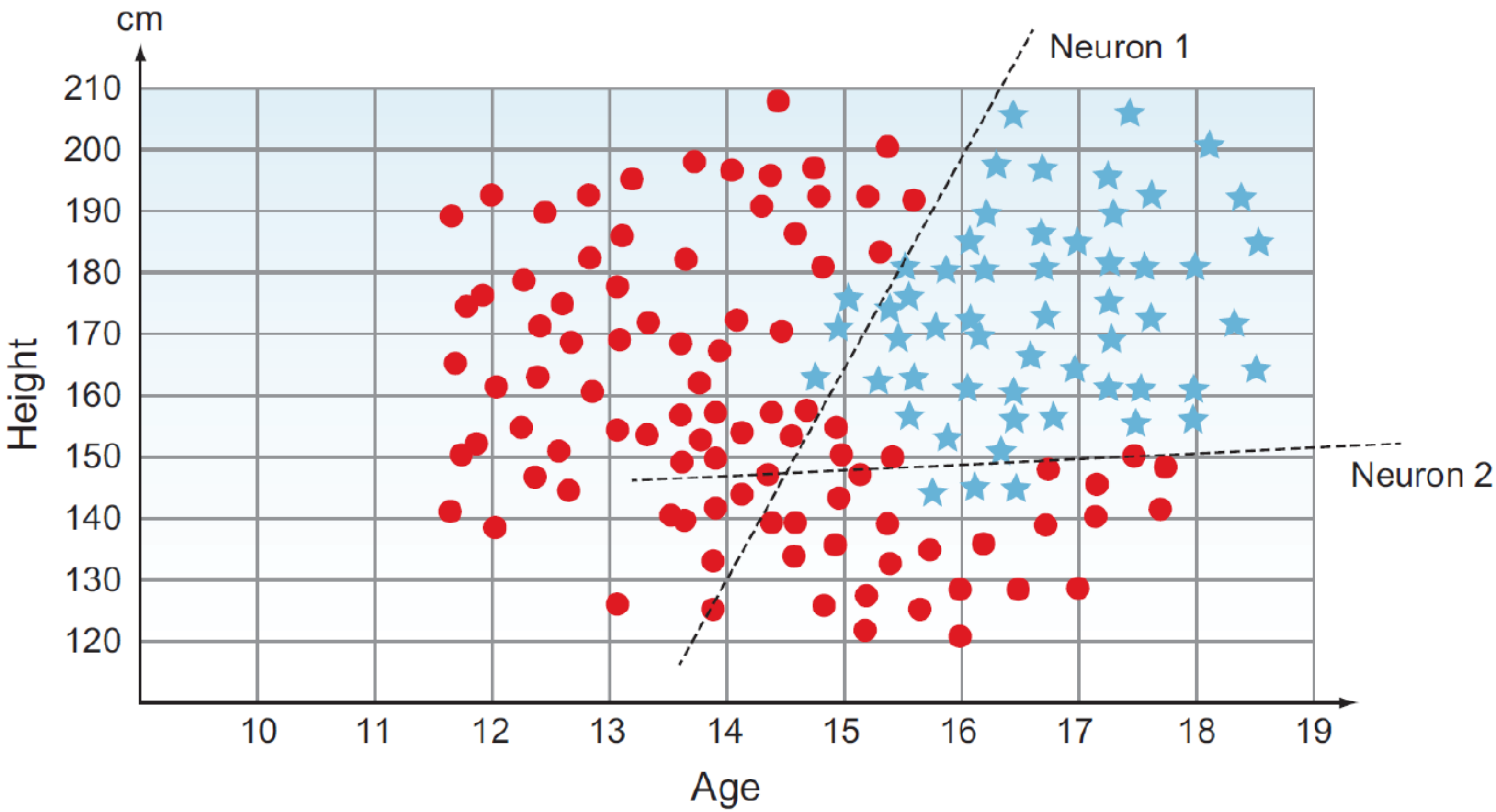
1. The neuron calculates the weighted sum and apply the activation function to make a prediction (feedforward process)
2. It then compares the prediction with the correct label to calculate the error
3. Update the weight: if the prediction is too high, it will adjust the weights to make a lower prediction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

Power of One Neural

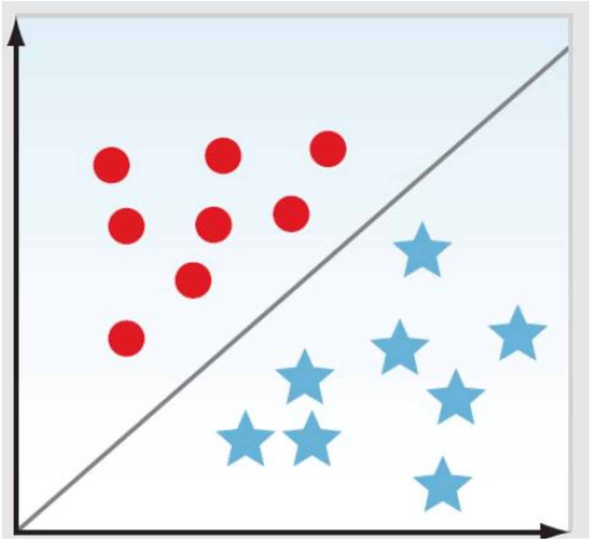


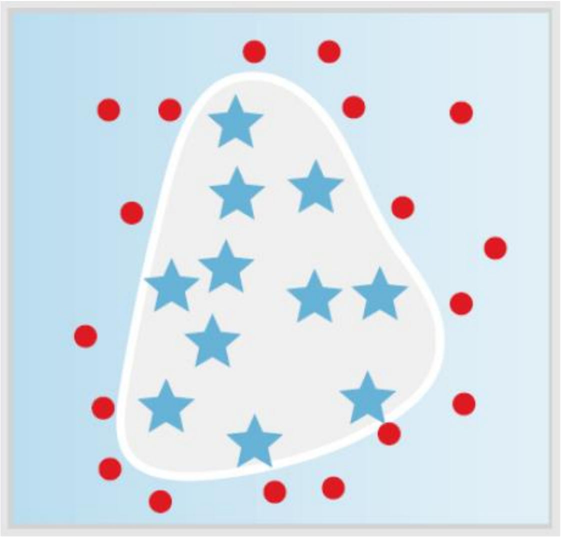
It can handle linearly separable problem

Adding more Neurals?



Linear vs. Non-Linear



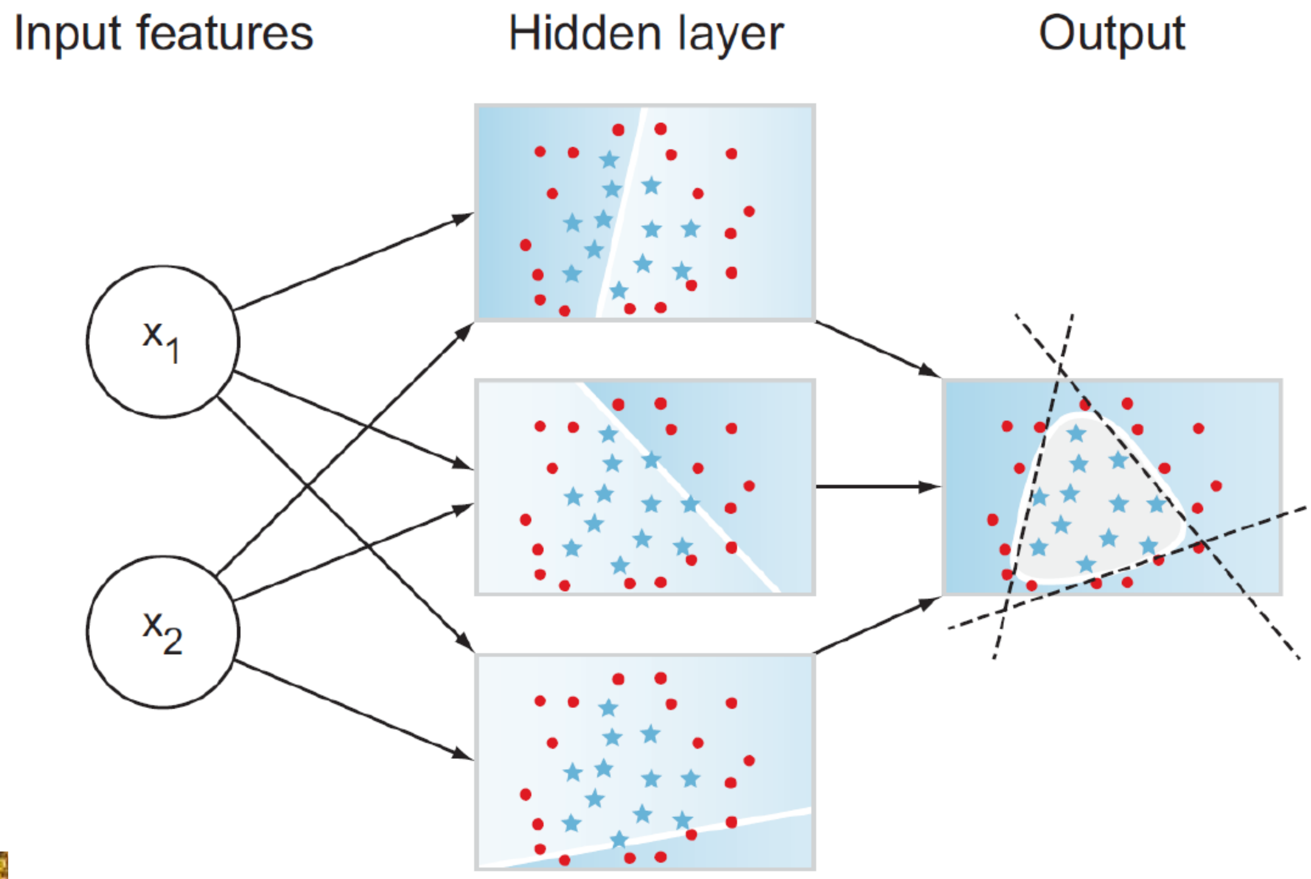


Linear

Non-Linear

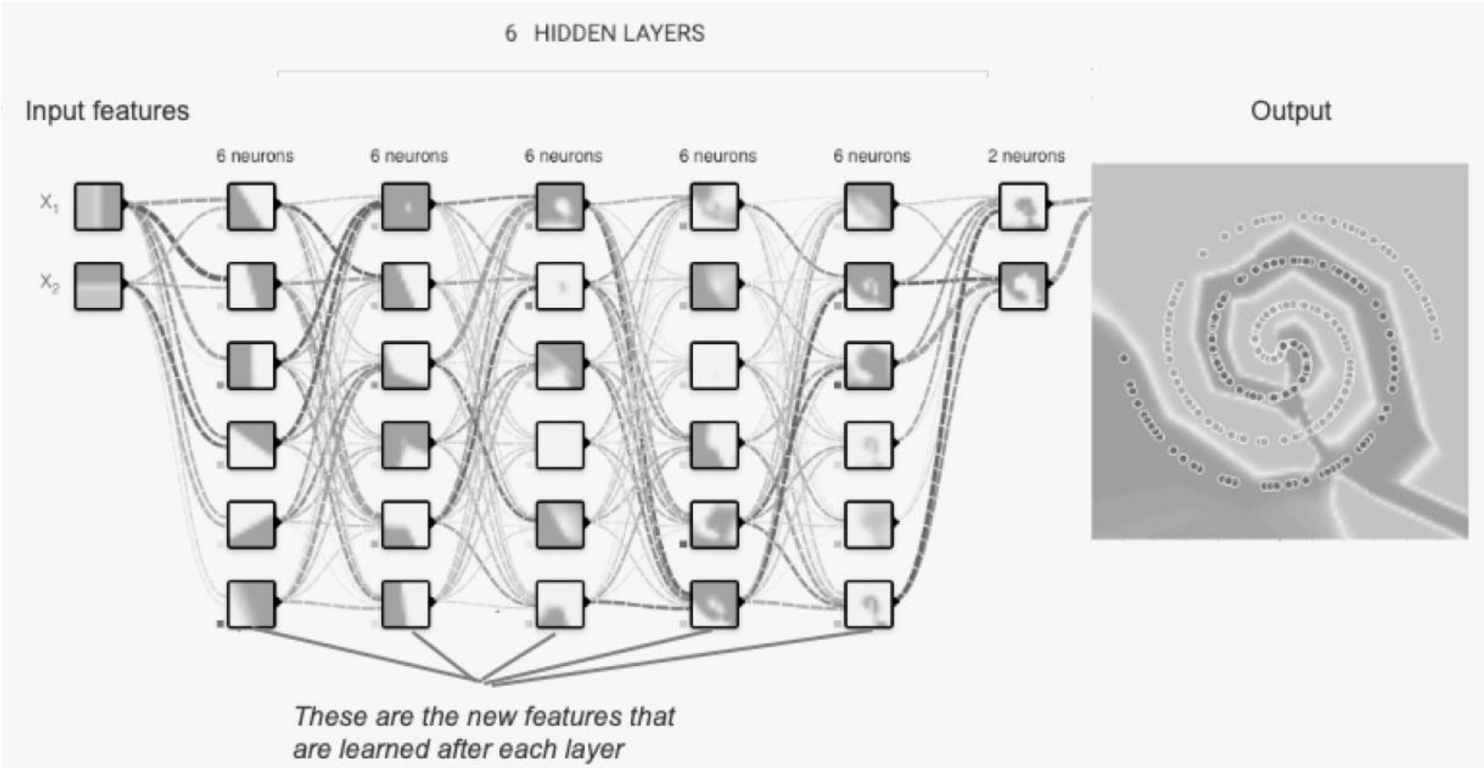
Multi-Layer Perceptron (MLP)

- Introduce the hidden layer



Multi-Layer Perceptron Architecture

- Can handle non-linear problem



Activation Function

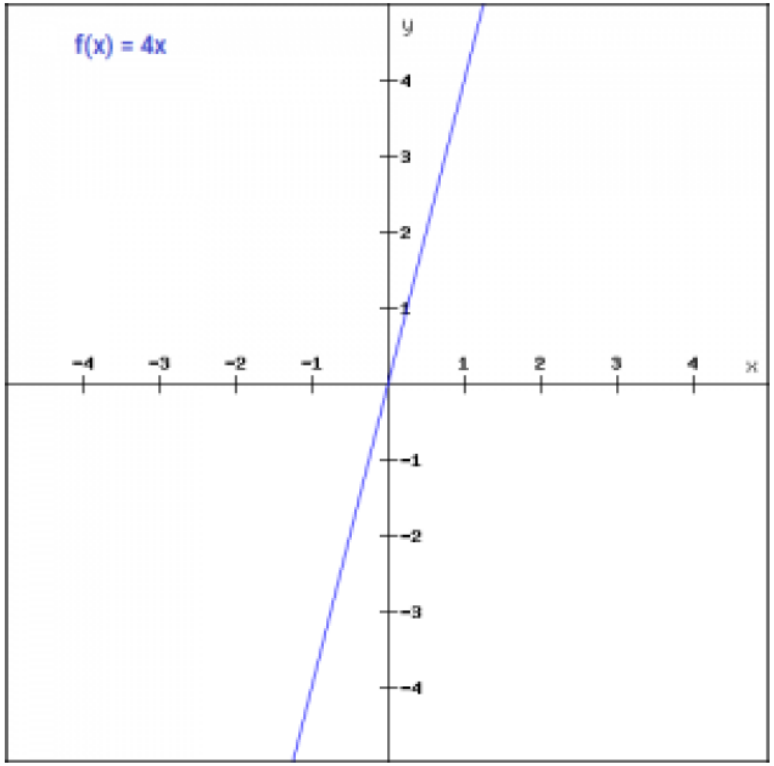
Also referred as transfer function or nonlinearities

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

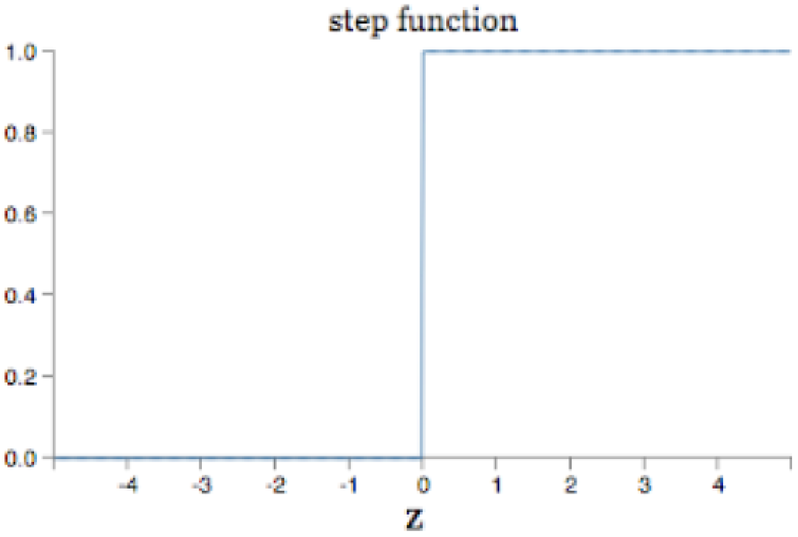


Step Function

Let y is the output

If the input x ≥ 0, y =1

else y = 0

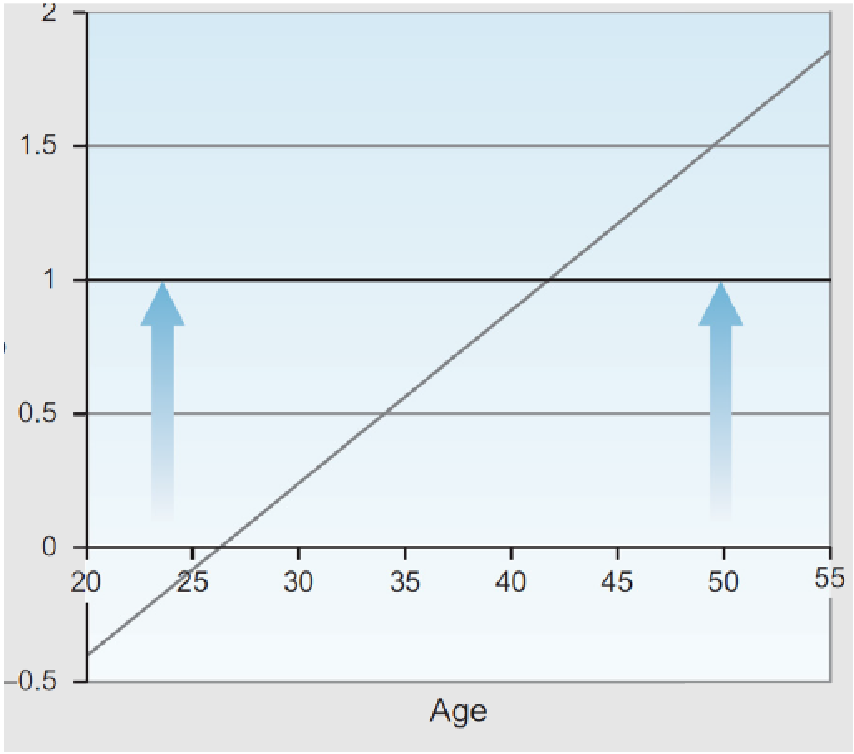


$$\text{output} = \begin{cases} 0 & \text{if } w \cdot x + b \leq 0 \\ 1 & \text{if } w \cdot x + b > 0 \end{cases}$$

Sigmoid Function Explanation

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

Normalized
Number of patients



Sigmoid Function Explanation

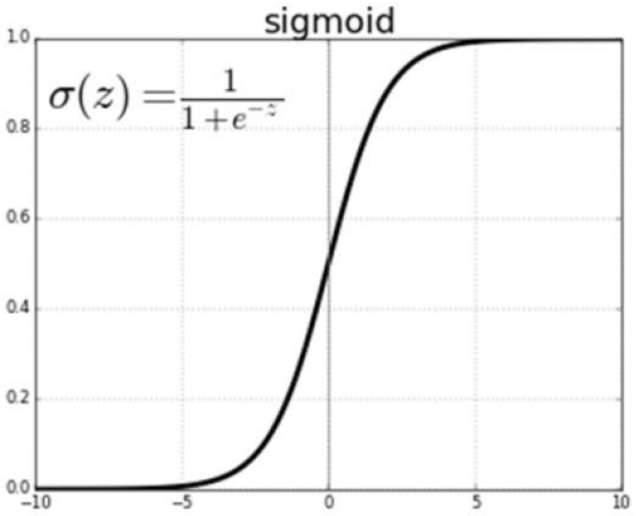
- We don’t want negative probability
- We don’t want 38 and 43 to have the same probability

Exponential function in Sigmoid can help

Sigmoid/Logistic Function

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

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



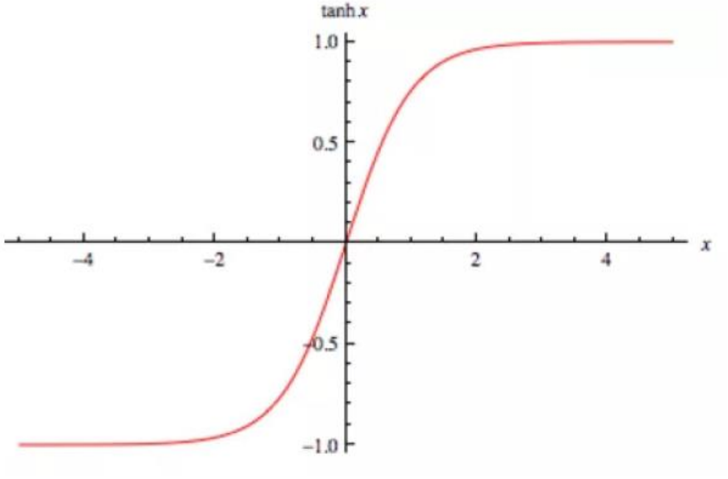

```python
# import numpy
import numpy as np

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

Hyperbolic Tangent Function(tanh)

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

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

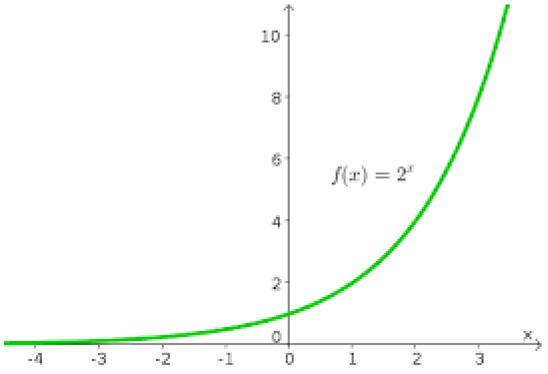


Softmax Function

- Use in multi-class classification

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

Exp function



Softmax Example

- Compute what is the probability?

Logits Scores

2.0

1.0

0.1

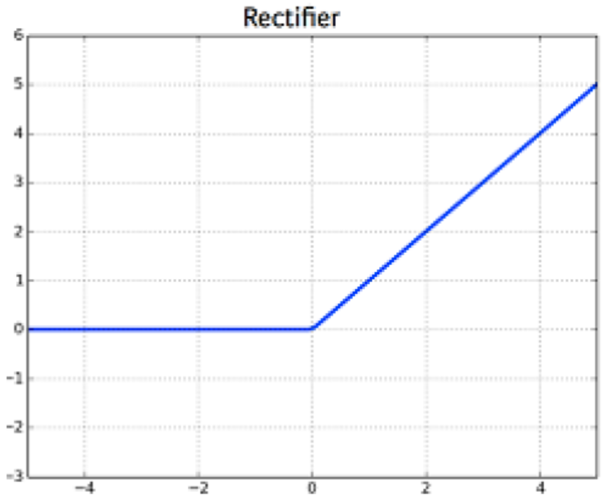
$$softmax(x)_i = \frac{exp(x_i)}{\sum_j exp(x_j))}$$

Answer

Logits Scores		Probabilities
2.0		0.7
1.0	$$\boxed{softmax(x)_i = \frac{exp(x_i)}{\sum_j exp(x_j))}}$$	0.2
0.1		0.1

Rectified Linear Unit (ReLU)

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



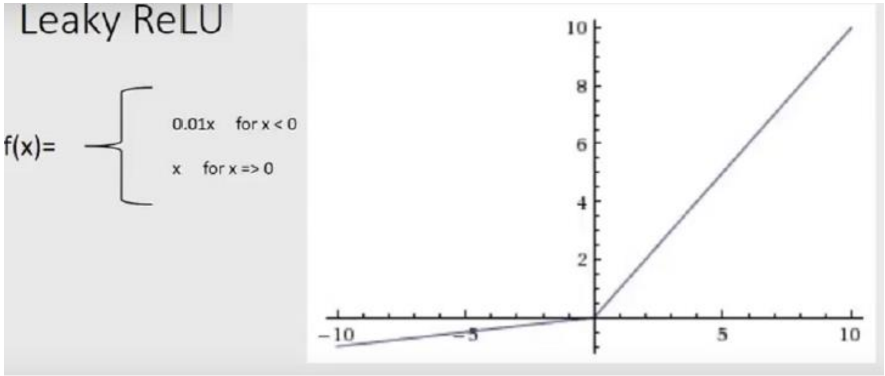
$$RELU(x) = \begin{cases} 0 \ \mathit{if} \ x < 0 \\ x \ \mathit{if} \ x >= 0 \end{cases}$$

Leaky ReLU

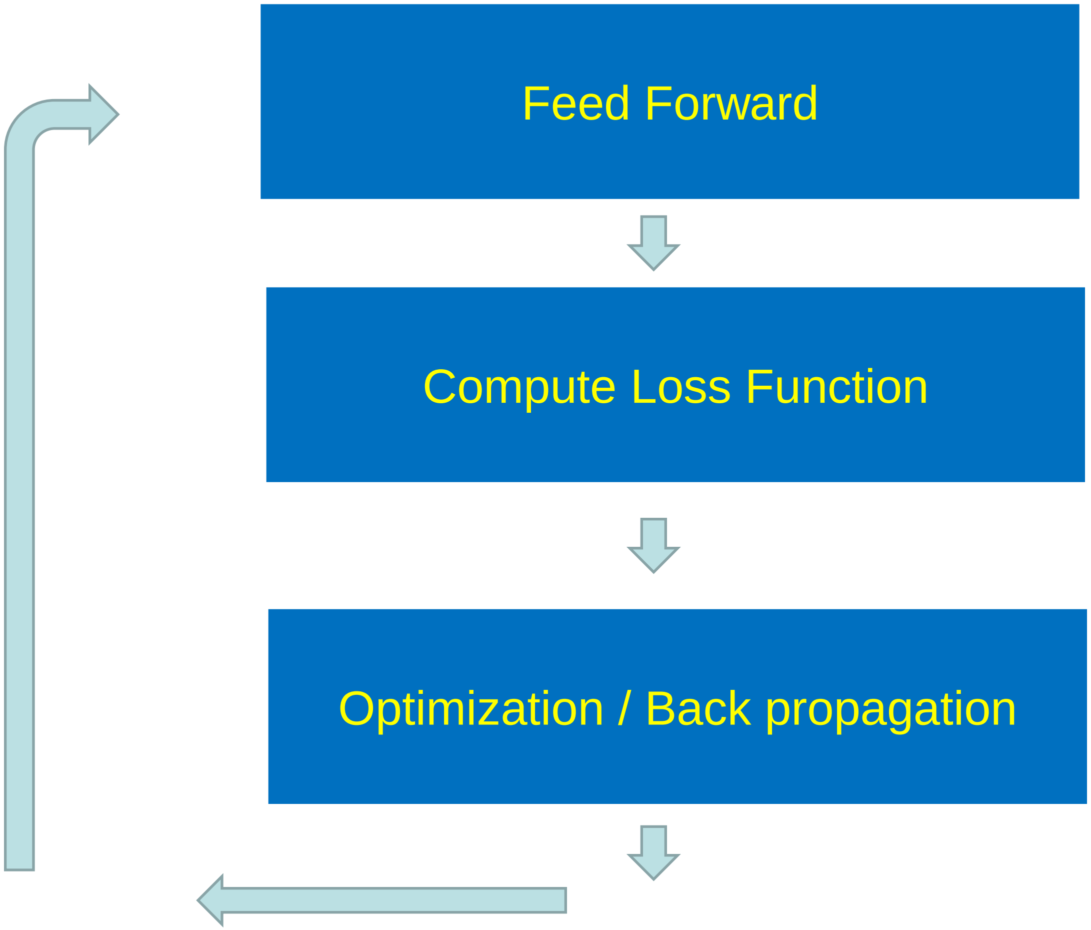
- Provide some negative weight over ReLU

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

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



Neural Network Training

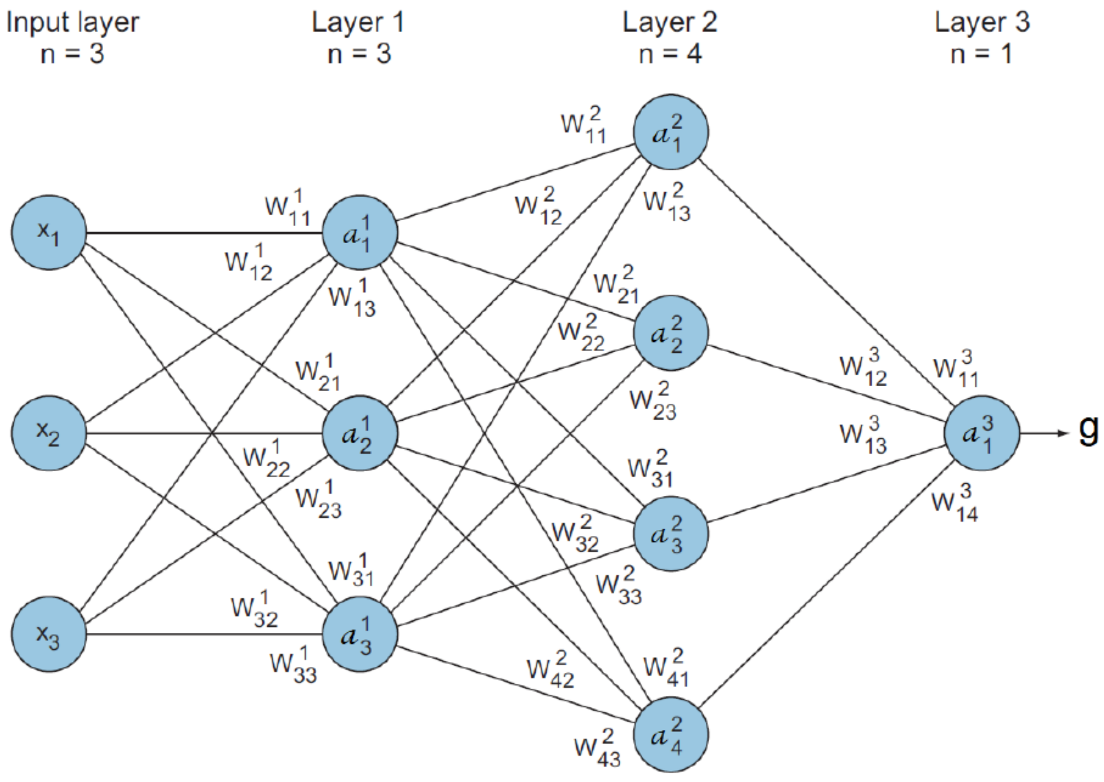


Neural Network Inference

Feed Forward

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)

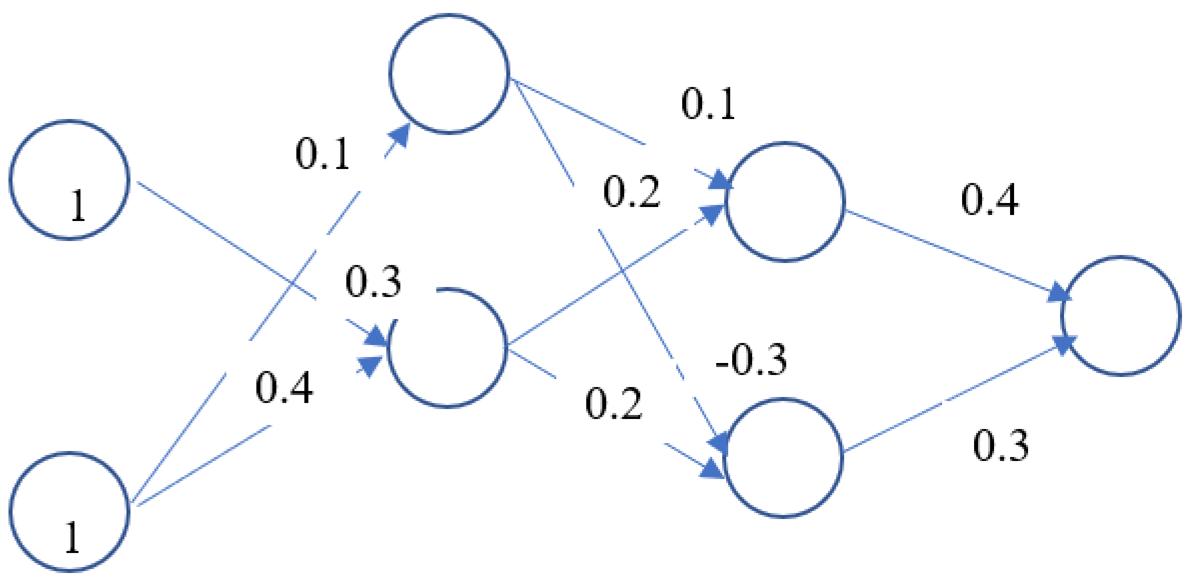
Feedforward calculation

$$\hat{y} = \sigma \underbrace{\overset{W^{(3)}}{\begin{bmatrix} w^{3}_{11} & w^{3}_{12} & w^{3}_{13} & w^{3}_{14} \end{bmatrix}}}_{\text{Layer 3}} \cdot \sigma \underbrace{\overset{W^{(2)}}{\begin{bmatrix} w^{2}_{11} & w^{2}_{12} & w^{2}_{13} \\ w^{2}_{21} & w^{2}_{22} & w^{2}_{23} \\ w^{2}_{31} & w^{2}_{32} & w^{2}_{33} \\ w^{2}_{41} & w^{2}_{42} & w^{2}_{43} \end{bmatrix}}}_{\text{Layer 2}} \cdot \sigma \underbrace{\overset{W^{(1)}}{\begin{bmatrix} w^{1}_{11} & w^{1}_{12} & w^{1}_{13} \\ w^{1}_{21} & w^{1}_{22} & w^{1}_{23} \\ w^{1}_{31} & w^{1}_{32} & w^{1}_{33} \end{bmatrix}}}_{\text{Layer 1}} \underbrace{\begin{bmatrix} x_{1} \\ x_{2} \\ x_{3} \end{bmatrix}}_{\text{Input vector}}$$

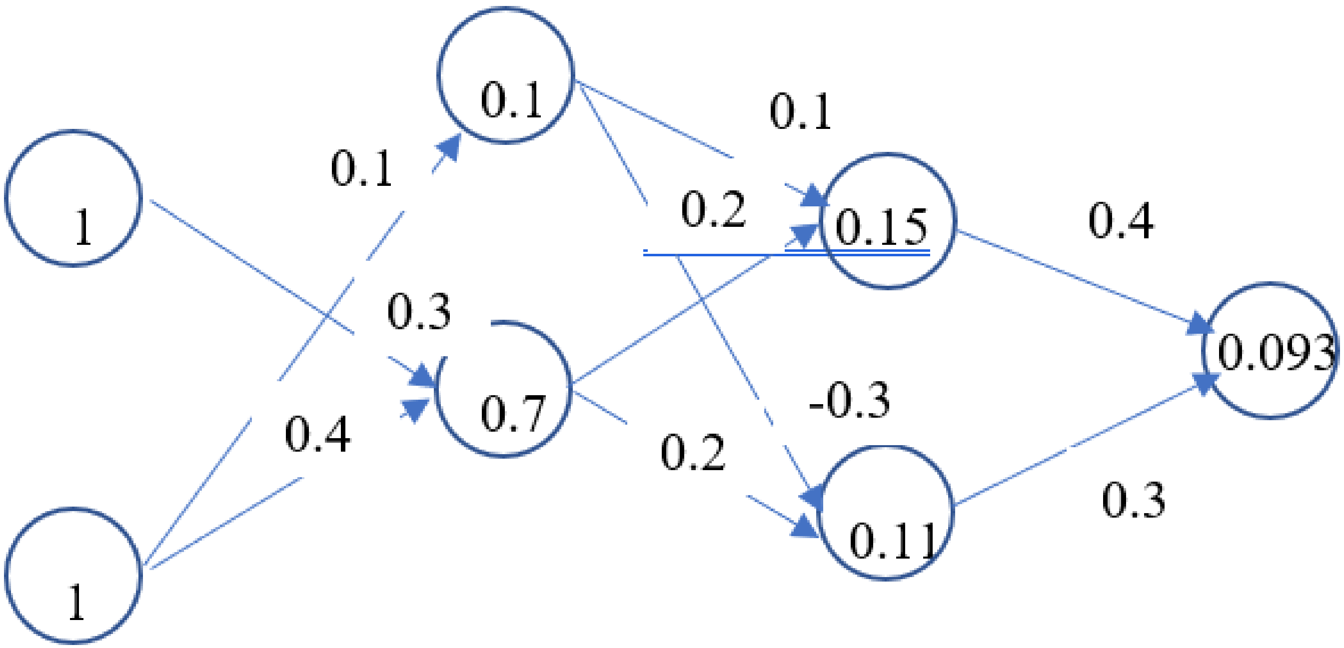
W^{k}_{ij} = weight from node j in layer k-1 to node i in layer k

Feed Forward

- Compute the final prediction output
- The output of next layer is just the weighted sum of the previous input layer
- Use RELU activation function for each output node
- What is the output?



Output



Error Function

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

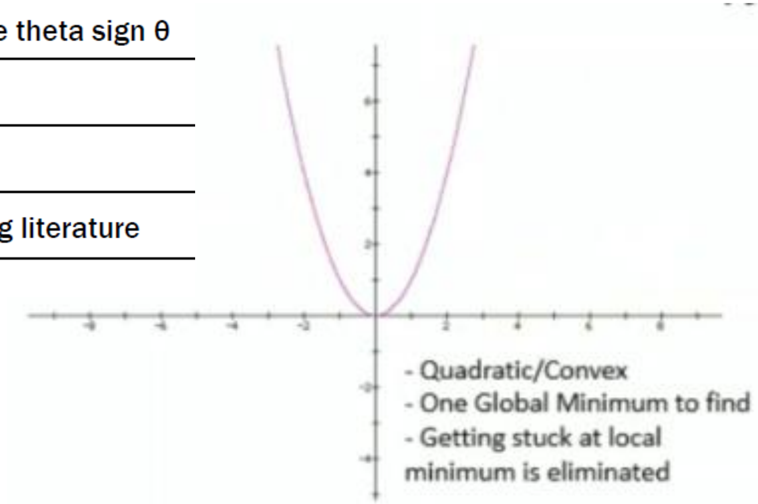
- The error should always be positive (to avoid the error to cancel each other)

Mean Square Error (L2 Loss)

- 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
y_i	The correct output (the label)
$(\hat{y}_i - y_i)$	Usually called the residual



Example

No.	Predicted	Actual	Error	Squared Error
1	48	60	-12	144
2	51	53	-2	4
3	57	60	-3	9

$$1/3 * (144 + 4 + 9) = 157/3 = 52.3$$

Variations

- Half of the Mean Squared Error

$$MSE = \frac{1}{2n}\sum_{i=1}^{n}(\hat{Y}_i - Y_i)^2$$

- Root Mean Squared Error

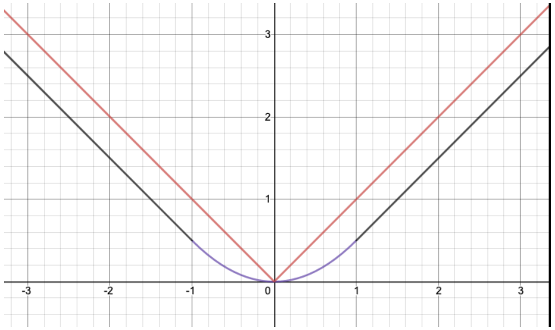
$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(\hat{Y}_i - Y_i)^2}$$

Mean Absolute Error (L1 Loss)

- Mean Absolute Error

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

- It is not continuous function

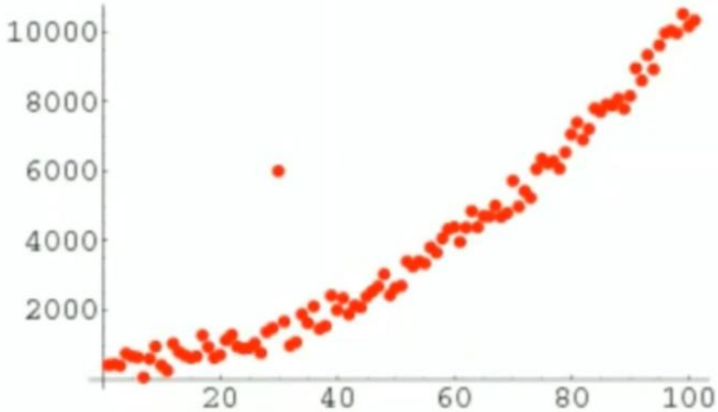


Example

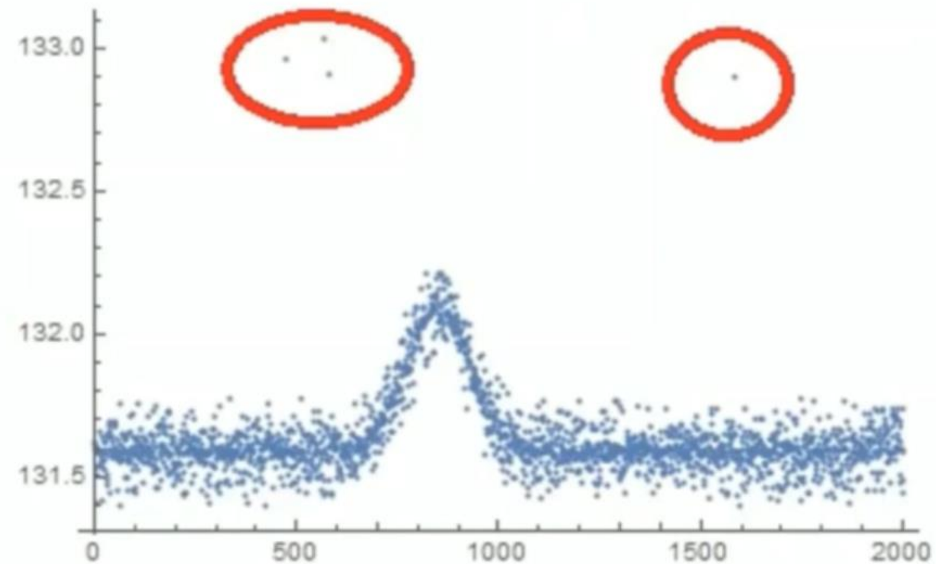
No.	Predicted	Actual
1	48	60
2	51	53
3	57	60

$$\text{Loss} = (|48-60| + |51-53| + |57-60|)/3$$
$$= (12 + 2 + 3) / 3$$
$$= 5.66$$

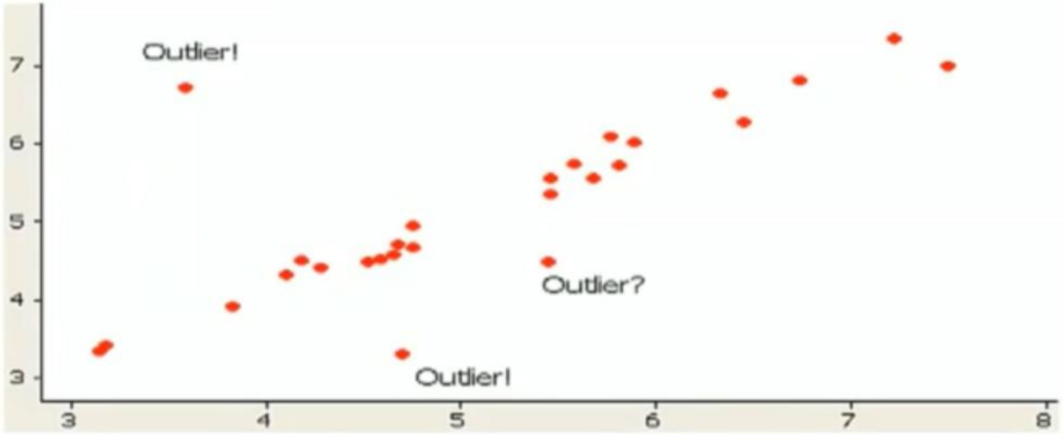
Outliers



10000
8000
6000
4000
2000
20
40
60
80
100



133.0
132.5
132.0
131.5
0
500
1000
1500
2000



Outlier!
Outlier?
Outlier!
7
6
5
4
3
3
4
5
6
7
8

When to use MSE or MAE Loss?

- MSE is more sensitive to outliers since it is computed as the square of error

If the absolute error is 10, the L2 loss will be 100

- If you want to lower outliers with the expense of higher average error, you can use MSE

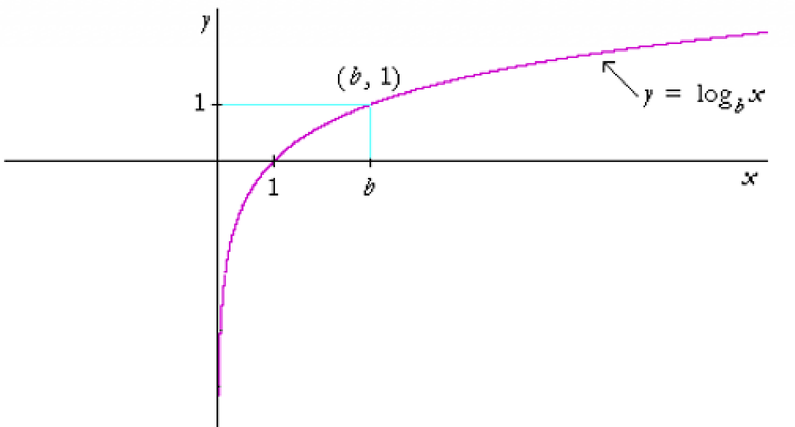
Cross Enthropy

- It is commonly used in classification problem

$$E\,(p_i,b) = -\Sigma^{m}_{i=1}\, y_i \log_b(p_i)$$

Use natural log function

y is the ground truth and p is the probability of prediction



Example

- The real probability: (natural log)

P(cat)	P(dog)	P(fish)
0.0	1.0	0.0

- The first prediction:

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

- The error function:

$$E = -(0.0 * \ln(0.2) + 1.0 * \ln(0.3) + 0.0 * \ln(0.5)) = 1.2$$

Example

- The second prediction

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

- The error function

$$E = -(0.0 * \ln(0.3) + 1.0 * \ln(0.5) + 0.0 * \ln(0.2)) = 0.6$$

Calculate the Cross Entropy

Class	Predicted Probabilities	Ground Truth
0	0.3	0
1	0.6	1
2	0.1	0

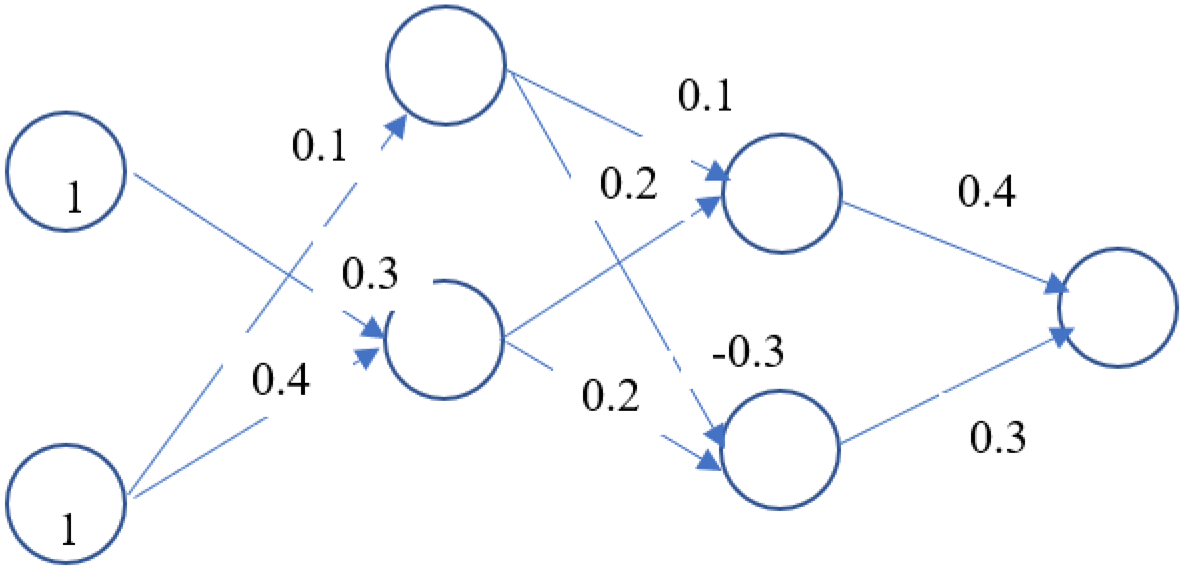
Answer

L = - (0x ln 0.3 + 1.0 x ln 0.6 + 0 x ln 0.1)

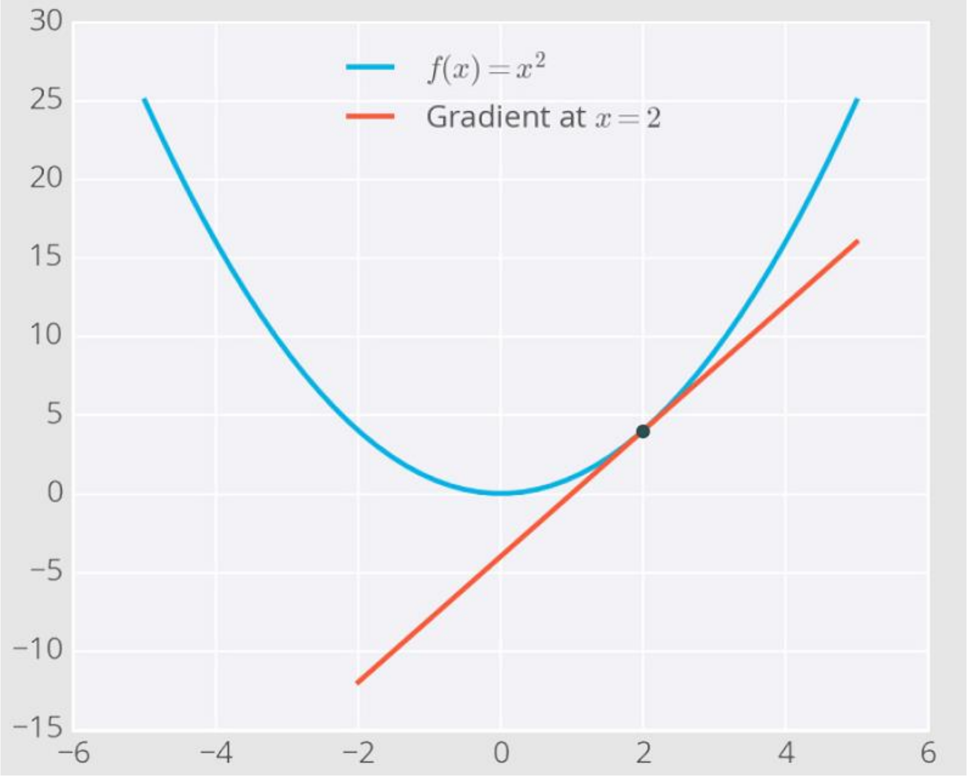
L = 0.510

Feed Forward

- Compute the final prediction output
- The output of next layer is just the weighted sum of the previous input layer



Partial Derivative



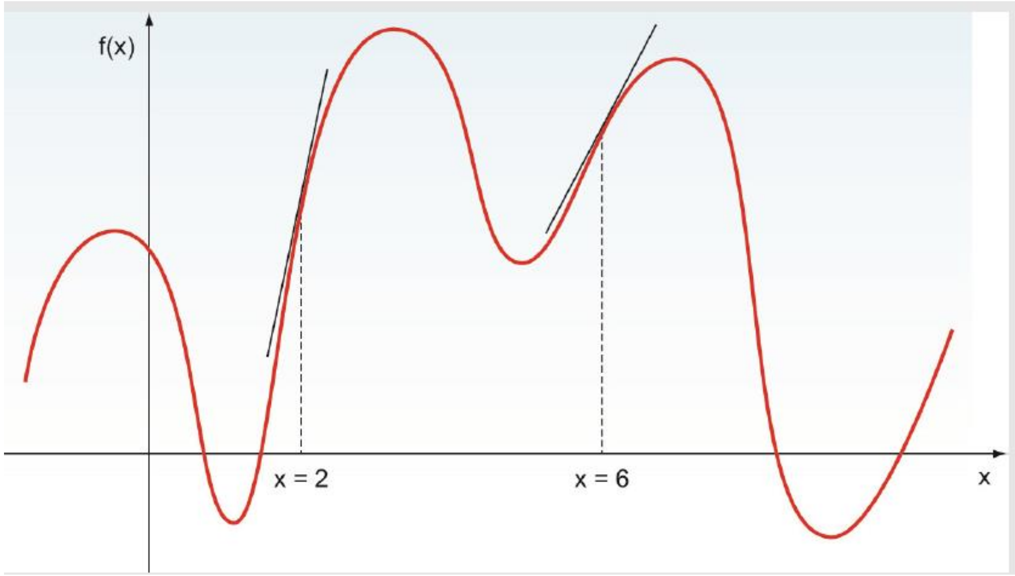
Derivative Rule

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: what is f'(x)

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

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



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

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

Backpropagation

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

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

Derivative of Error with respect to weight

Old weight

$$W_{new} = W_{old} - \alpha \left(\frac{\partial Error}{\partial W_x}\right)$$

New weight

Learning rate

Backpropagation

- It is based on the chain rule

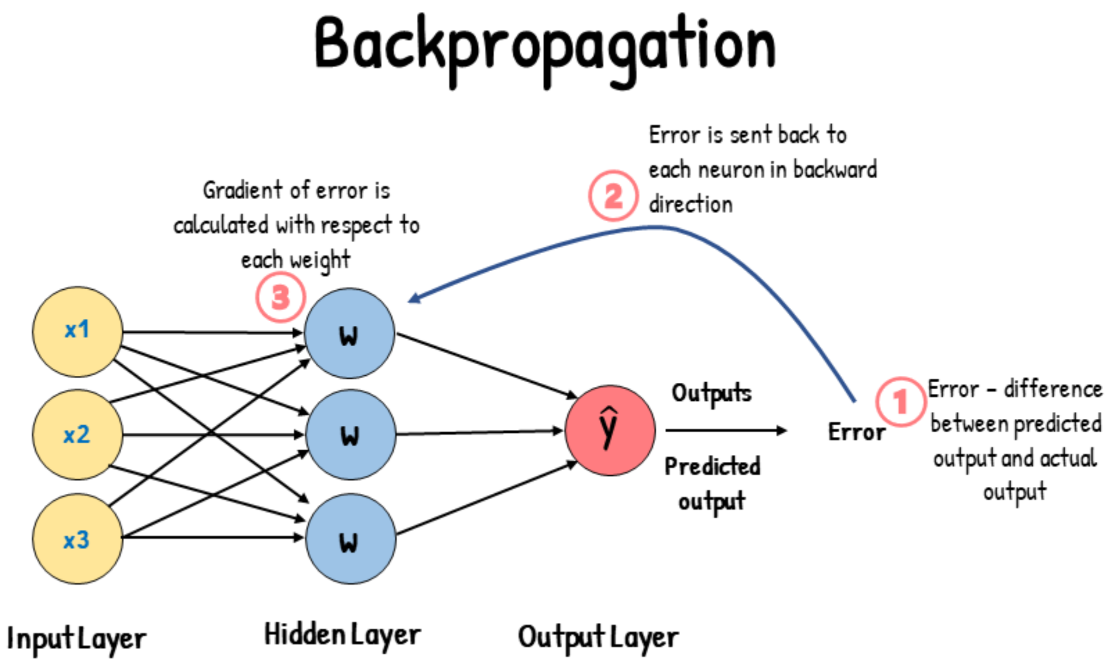


Image from Analytics Vidhya

Chain Rule in Derivatives

- Chain Rule

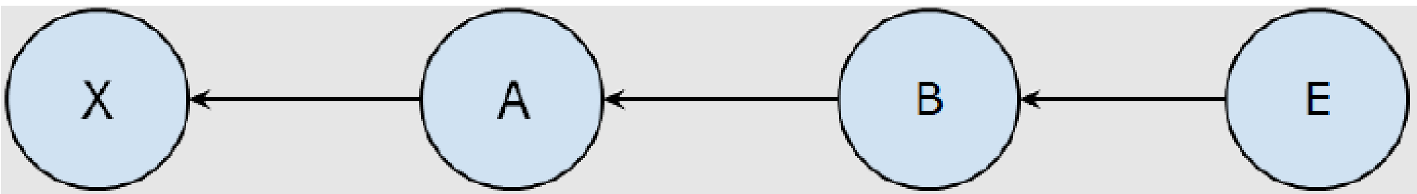
Chain Rule: $\frac{d}{dx}f(g(x)) = f'(g(x))g'(x)$

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

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

Example of Chain Rule

- We want to calculate dE/dx



$$\frac{dE}{dx} = \frac{dE}{dB} * \frac{dB}{dA} * \frac{dA}{dx}$$

Derivative function

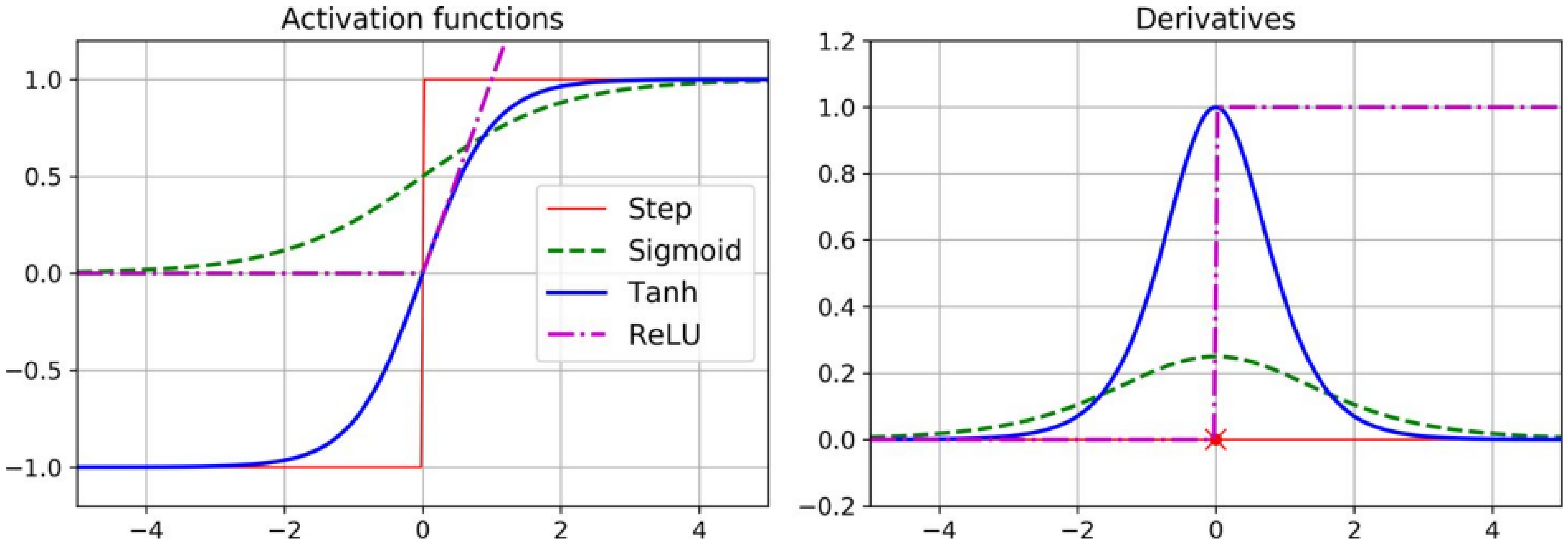


Figure 10-8. Activation functions and their derivatives

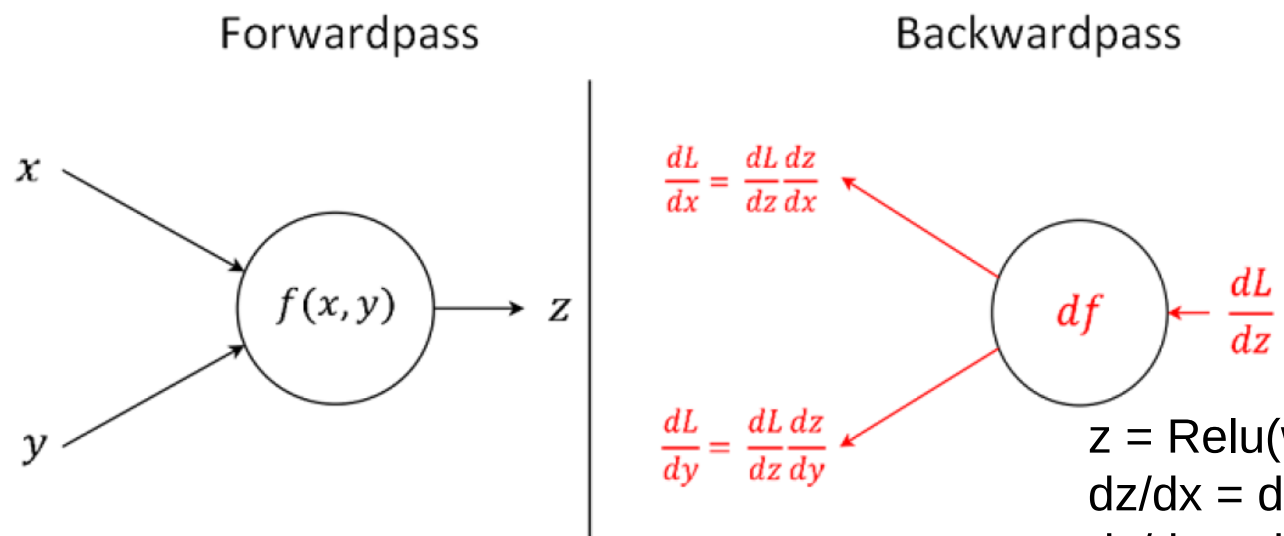
Python

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 Summary

- Forward pass is to calculate predicted output
- Backward propagation is to update the weight to minimize the error

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



$z = Relu(w_1x + w_2y+b)$

$dz/dx = dRelu(x) *w_1$

$dz/dy = dRelu(y) * w_2$

Weight update techniques

- This is also applied on Convolution Neural Network

$$\begin{bmatrix} 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 \end{bmatrix} * \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 2 & 1 & -1 \\ -1 & 1 & 3 \\ 2 & 1 & 1 \end{bmatrix}$$

Input Image — Filter or Kernel — Output or Feature Map

Weight update equation

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

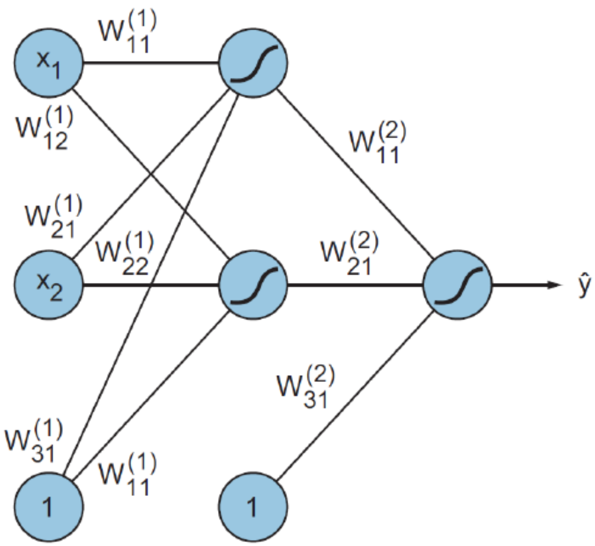
```
weight = weight - (alpha * derivative)
```

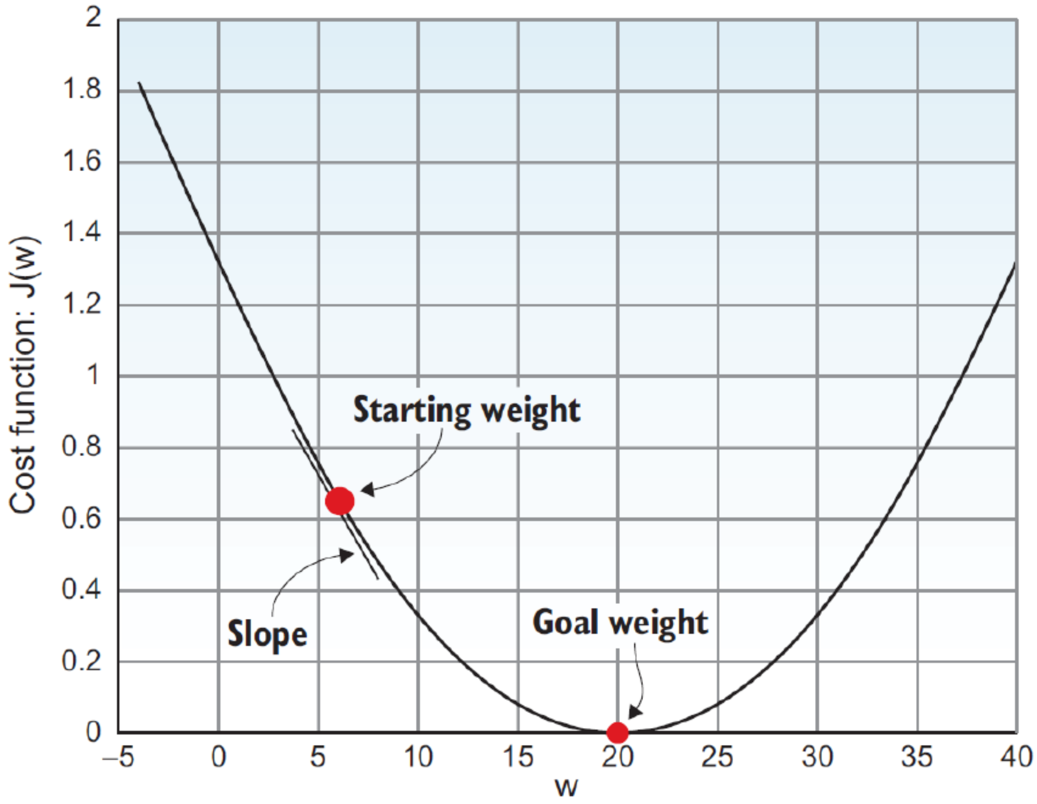
```
weight = weight - (input * (pred - goal_pred)*)alpha
```

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

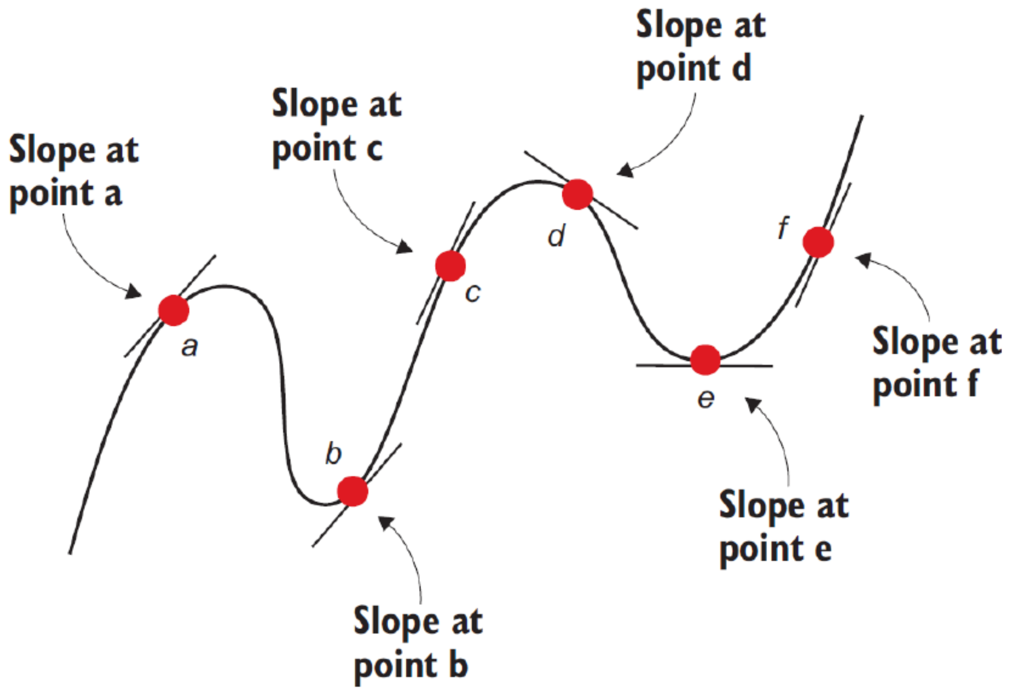
Weight Value





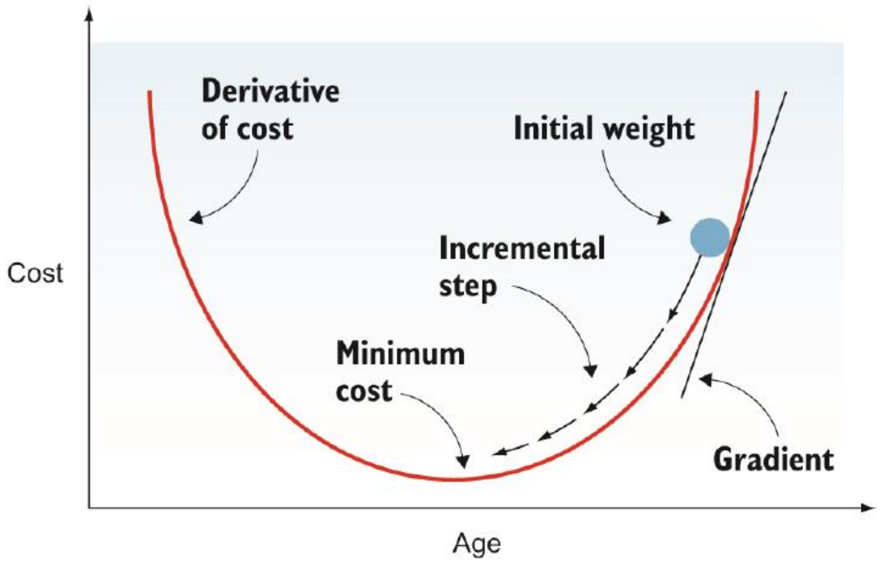
Batch Gradient Descent

- 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

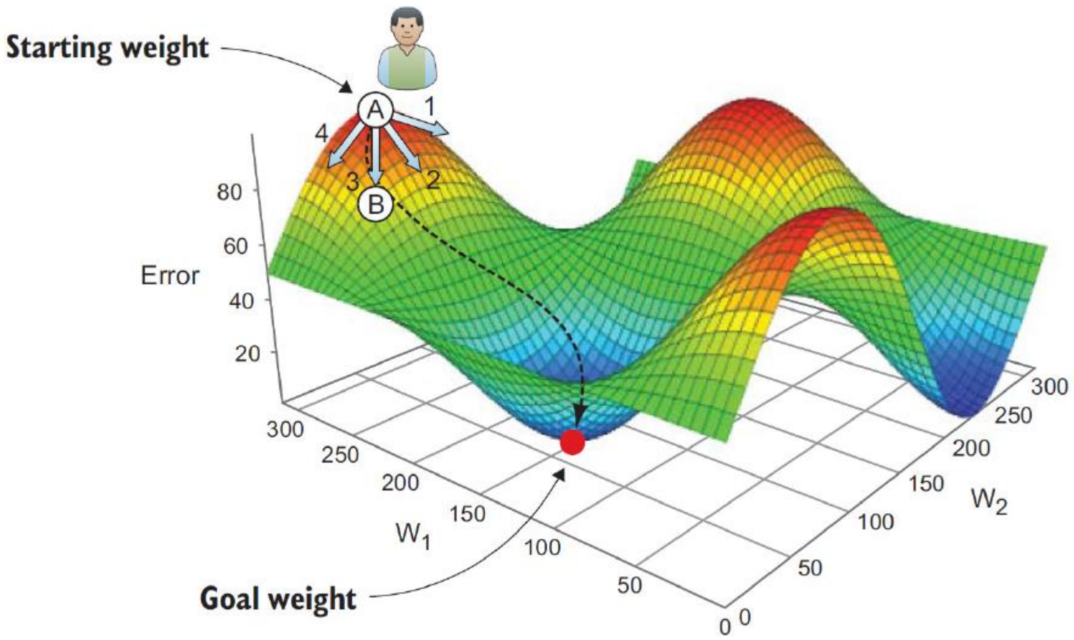


How does gradient descent work?

- The step direction (gradient)

- The step size (learning rate)

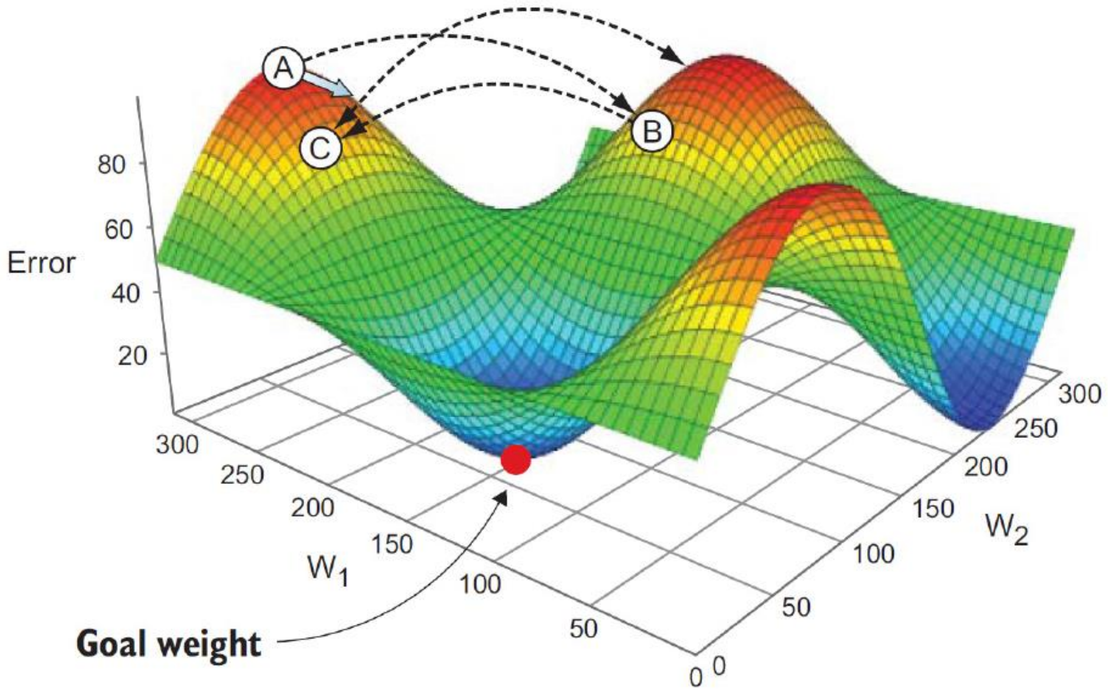
Example



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 /2
```

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

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

Standard/Batch Gradient Descent (BGD)

- It uses the entire training set to update the weight

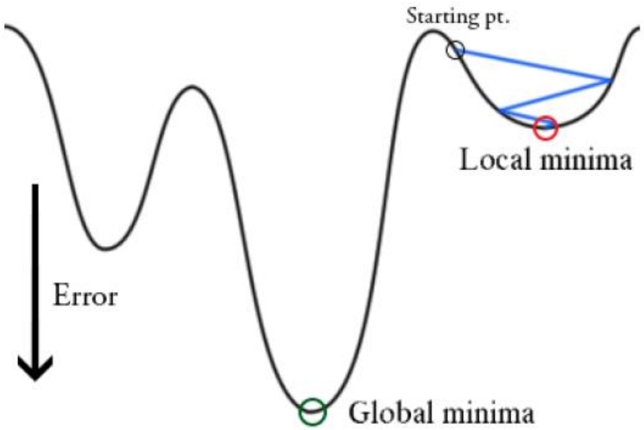
- The error function

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

- N is the total amount of data in training set
- Sometimes, we also use 2N

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)

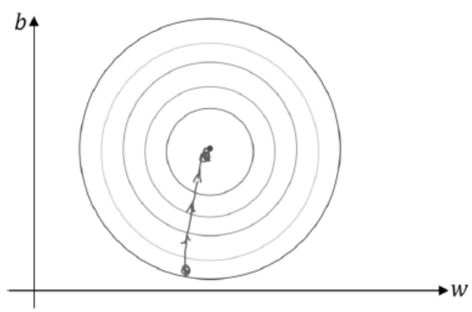
- SGD is the most used optimization algorithms for machine learning

- 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

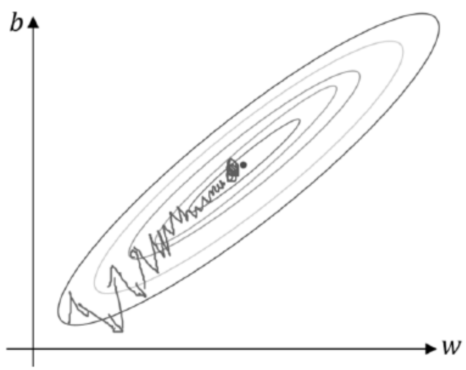
1) Take ALL the data
2) Compute the gradient
3) Update the weights and take a step down
4) Repeat for n number of epochs (iterations)



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)



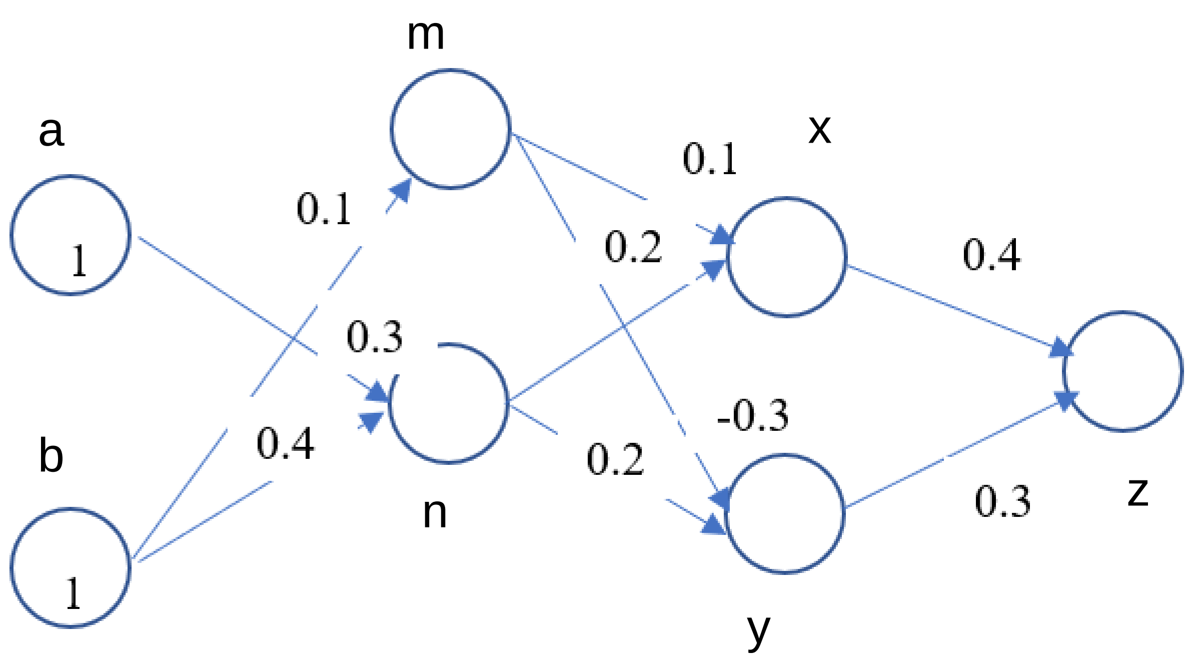
Top View of the error mountain

Mini-batch Gradient Descent (MN-GD)

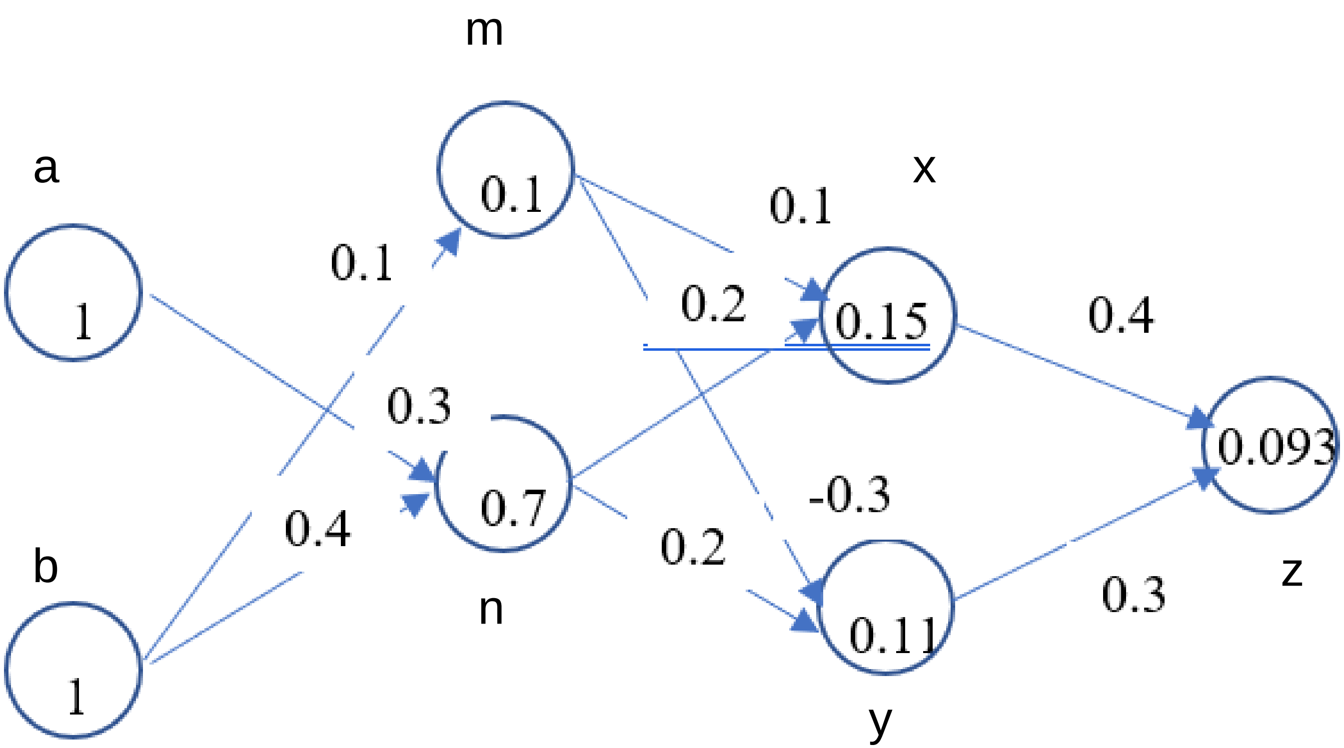
- The compromise between Batch GD and Stochastic GD
- Group of training instead of a single instance
- It is faster comparing with BGD
- It reduces small error from SGD

Example

- Compute the feed forward and back propagation for 1 iteration of weight update. Assume the MSE and that the ground truth output is 1 and alpha (learning rate = 1).

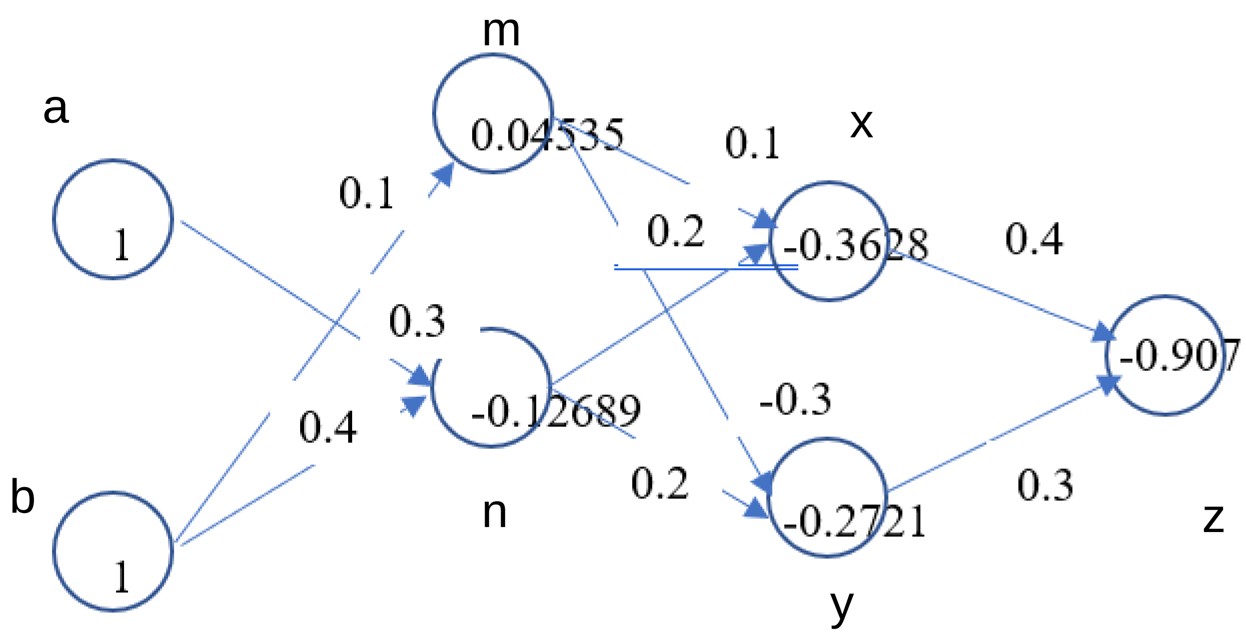


Feedforward



All node value has positive results/ relu2dev are all 1.

Backward Delta



Error = E = (0.093-1)^2 = 0.822

dE/dz = 0.093-1 = -0.907

dE/dx = dE/dz * dz/dx = -0.907 * 0.4= - 0.3628   (z = 0.4 x + 0.3 y)

dE/dy = dE/dz * dz/dy = -0.907 * 0.3= - 0.2721

dE/dm = dE/dz * dz/dx * dx/dm + dE/dz * dz/dy * dy/dm = -0.3628*0.1 -0.2721*(-0.3) = 0.04535

dE/dn = dE/dz * dz/dx * dx/dn + dE/dz * dz/dy * dy/dn = -0.3628* 0.2 - 0.2721*0.2 = - 0.12689

Delta_weight is computed by just multiply back from the edge weight backward

Weight update

Level 1

$w_{am} = w_{am} - \alpha\ dE/dm = 0 – 1 * 0.04535 = - 0.04535$

$w_{bm} = w_{bm} - \alpha\ dE/dm = 0.1 – 1 *0.04535 = 0.05465$

$w_{an} = w_{an} - \alpha\ dE/dn = 0.3 + 1 * 0.12698 = 0.42698$

$w_{bn} = w_{bn} - \alpha\ dE/dn = 0.4 + 1 *0.12698 = 0.52698$

Level 2

$w_{mx} = w_{mx} - \alpha\ dE/dx = 0.1 + 0.1* 0.3628 = 0.13628$

$w_{nx} = w_{mx} - \alpha\ dE/dx = 0.2 + 0.7*0.3628 = 0.45396$

$w_{my} = w_{my} - \alpha\ dE/dy = -0.3 + 0.1 *0.2721 = -0.2729$

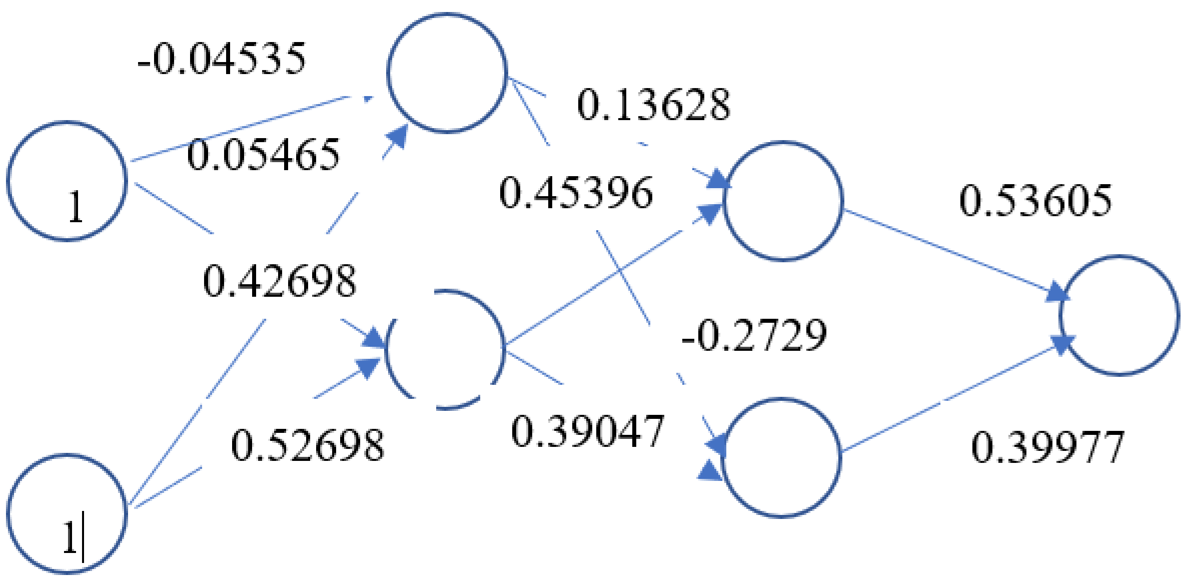
$w_{ny} = w_{ny} - \alpha\ dE/dy = 0.2 + 0.7 * 0.2721 = 0.39047$

Level 3

$w_{xz} = w_{xz} - \alpha\ dE/dz = 0.4 + 0.15*0.907 = 0.53605$

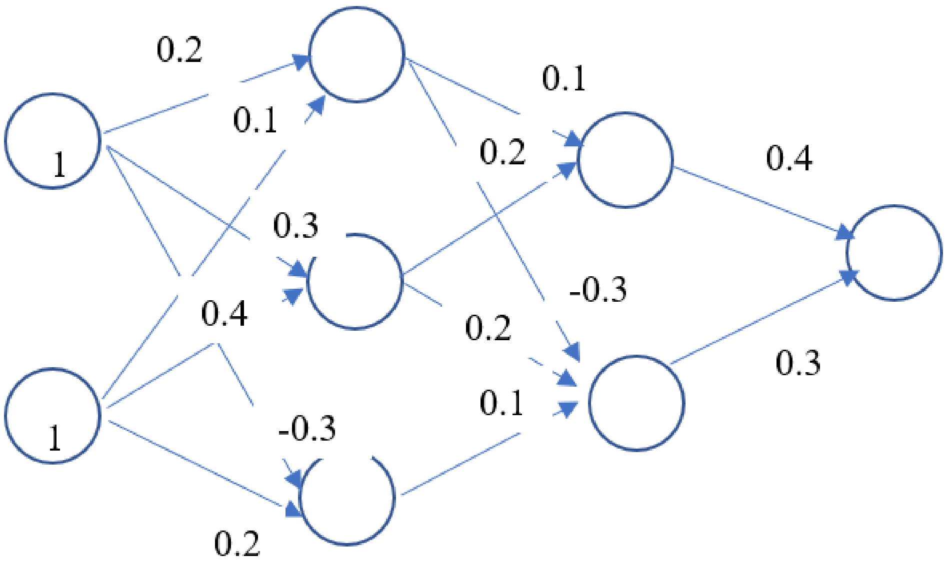
$w_{yz} = w_{yz} - \alpha\ dE/dz = 0.3 + 0.11 * 0.907 = 0.39977$

Backward Weight update



Try this

- Please compute feed forward and back propagation for 1 iteration



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 ]] )
 output1 = np.array([[ 1]]).T
 alpha = 1
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 = 
layer_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)
```

Question?

If we double the number of layers in a neural network, how much more time (approximately) do we expect the backpropagation algorithm to take?

Answer

Twice

Question?

If we double the number of neurons in each layer of a neural network, how much more time (approximately) do we expect the backpropagation algorithm to take?

Answer

Four times

Website to test ANN Model

https://playground.tensorflow.org/#activation=tanh&batchSize=10&dataset=circle&regDataset=reg-plane&learningRate=0.03&regularizationRate=0&noise=0&networkShape=4,2&seed=0.53408&showTestData=false&discretize=false&percTrainData=50&x=true&y=true&xTimesY=false&xSquared=false&ySquared=false&cosX=false&sinX=false&cosY=false&sinY=false&collectStats=false&problem=classification&initZero=false&hideText=false

Question?

Homework

- Write a similar Python Program for the following network