Neural Networks

Jaegul Choo (주재걸)
Korea University
https://sites.google.com/site/jaegulchoo/
Slides made by my student, Yunjey Choi

Today's Contents

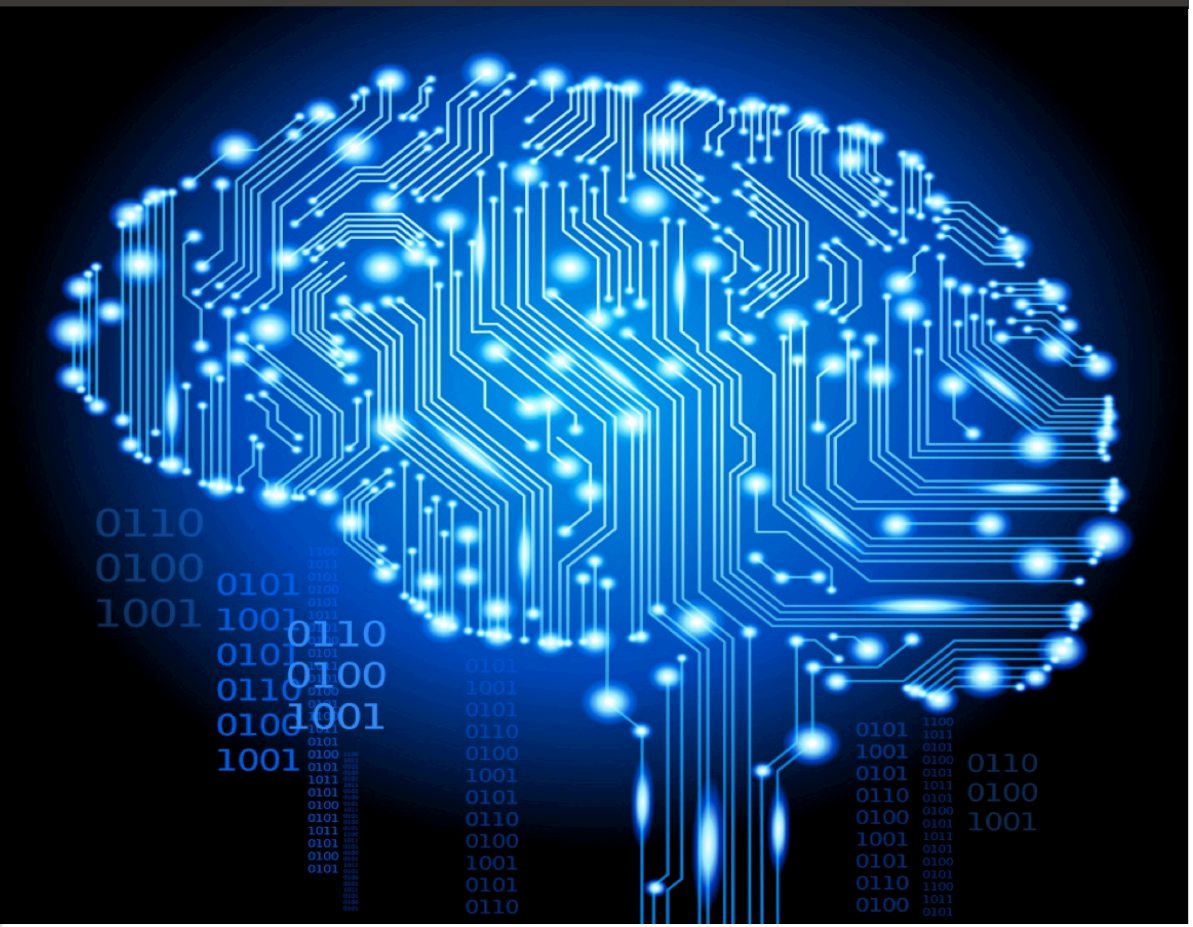
Neural Networks

1. **Deep Learning**
2. **Perceptron**
3. **Neural Network**
4. **Forward Propagation**
5. **Backpropagation**

1. Deep Learning

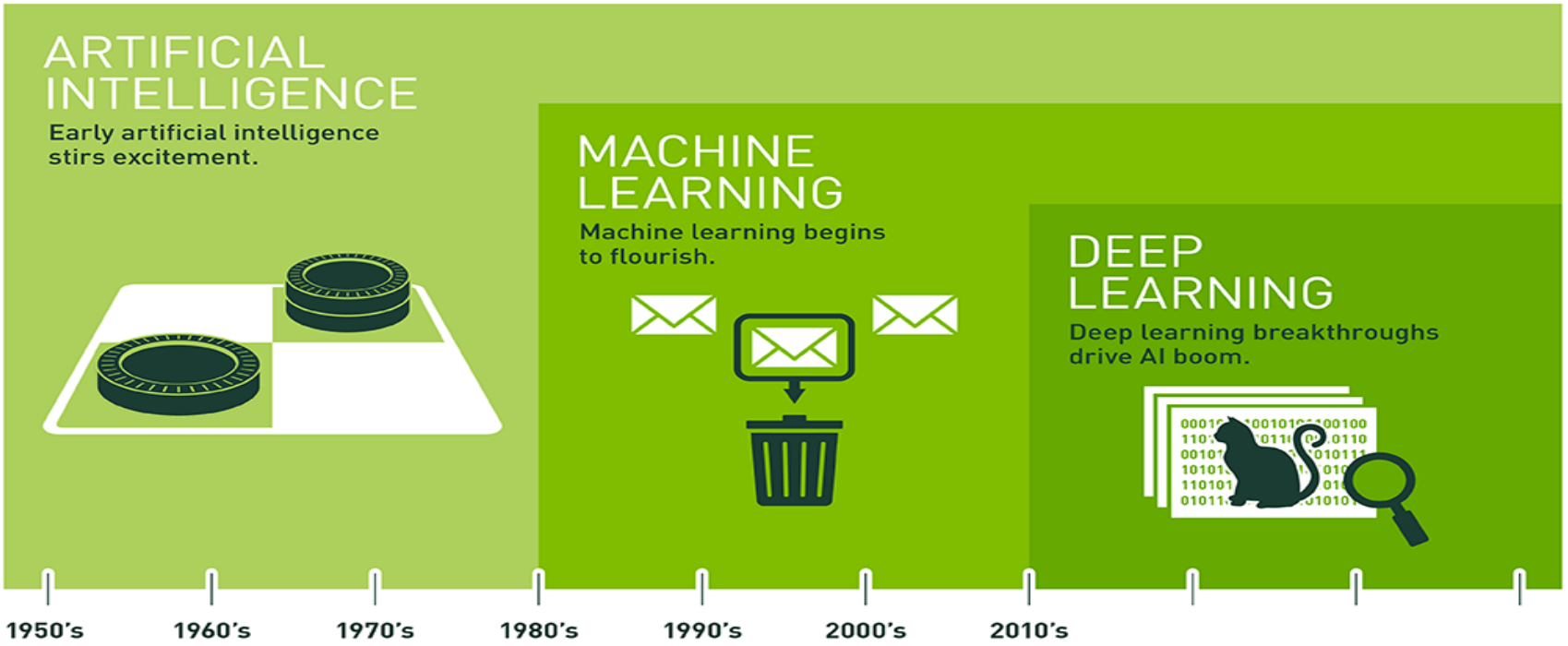
Deep Learning

Deep learning refers to artificial neural networks that are composed of many layers.



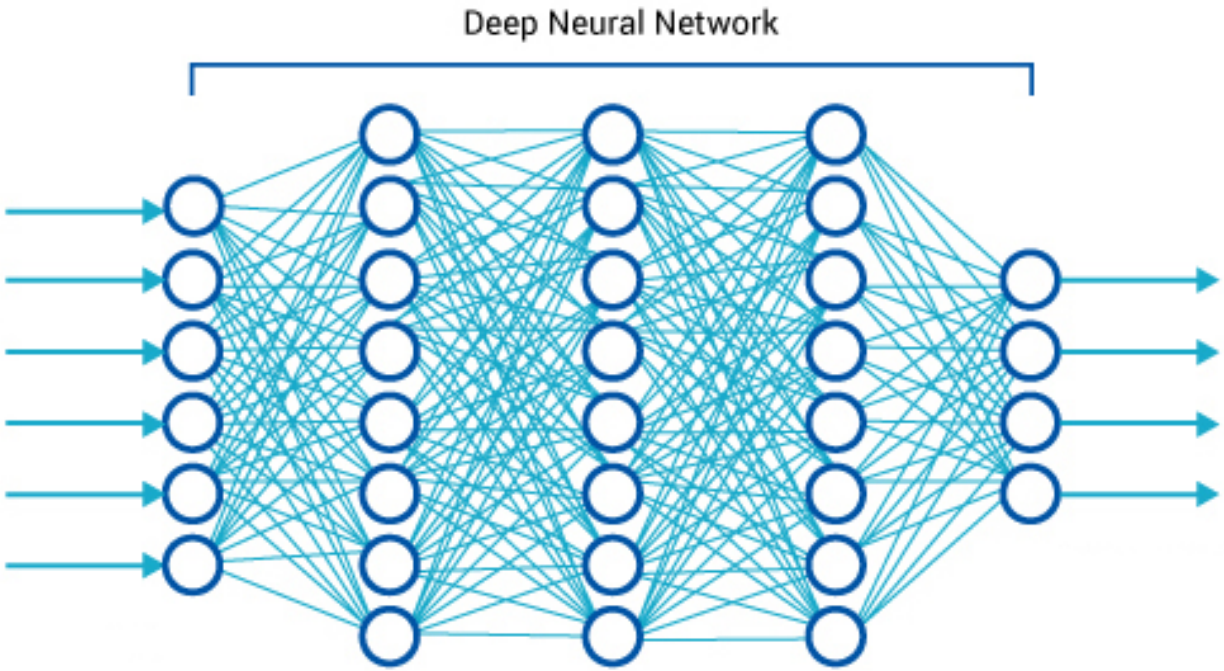
Deep Learning

Deep Learning

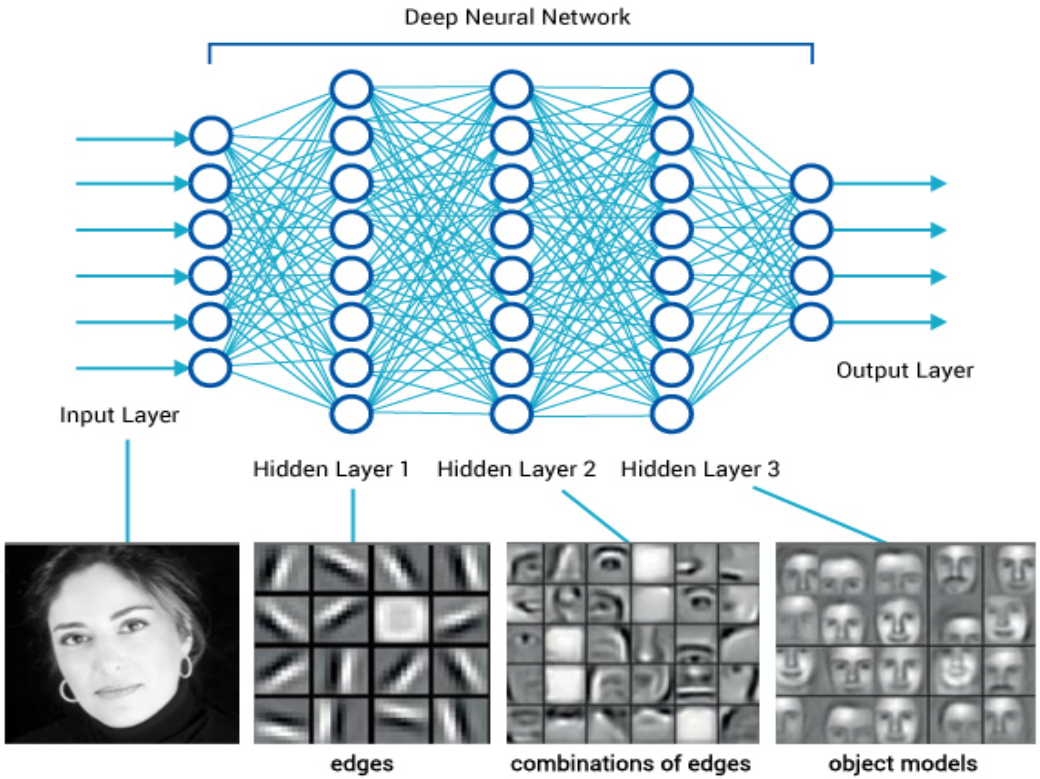


Since an early flush of optimism in the 1950s, smaller subsets of artificial intelligence – first machine learning, then deep learning, a subset of machine learning – have created ever larger disruptions.

Deep Learning



Deep Learning

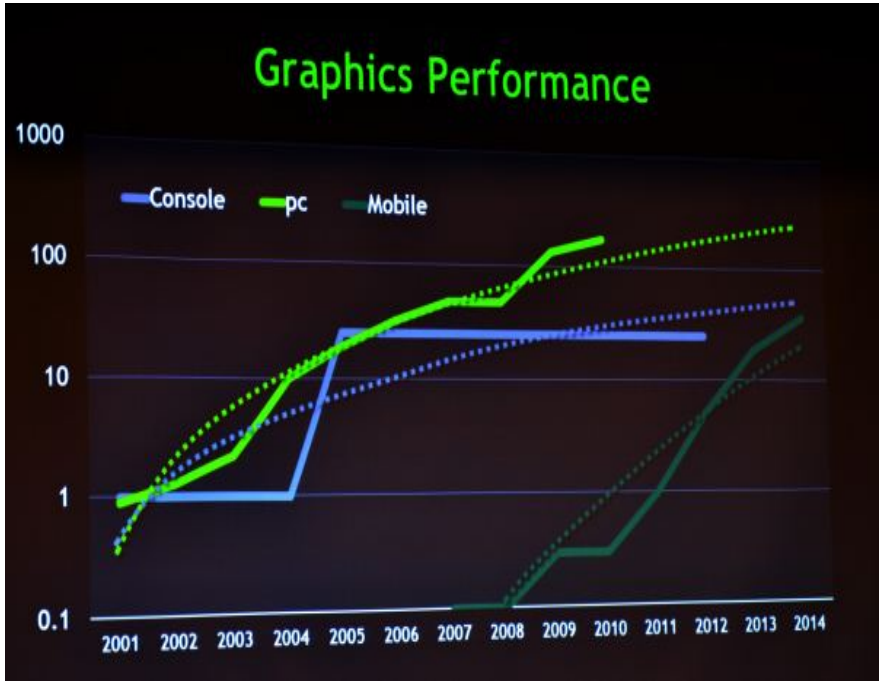


Why is deep learning a growing trend?

- few feature engineering

- state-of-the-art performance

Deep Learning



Deep Learning Heroes

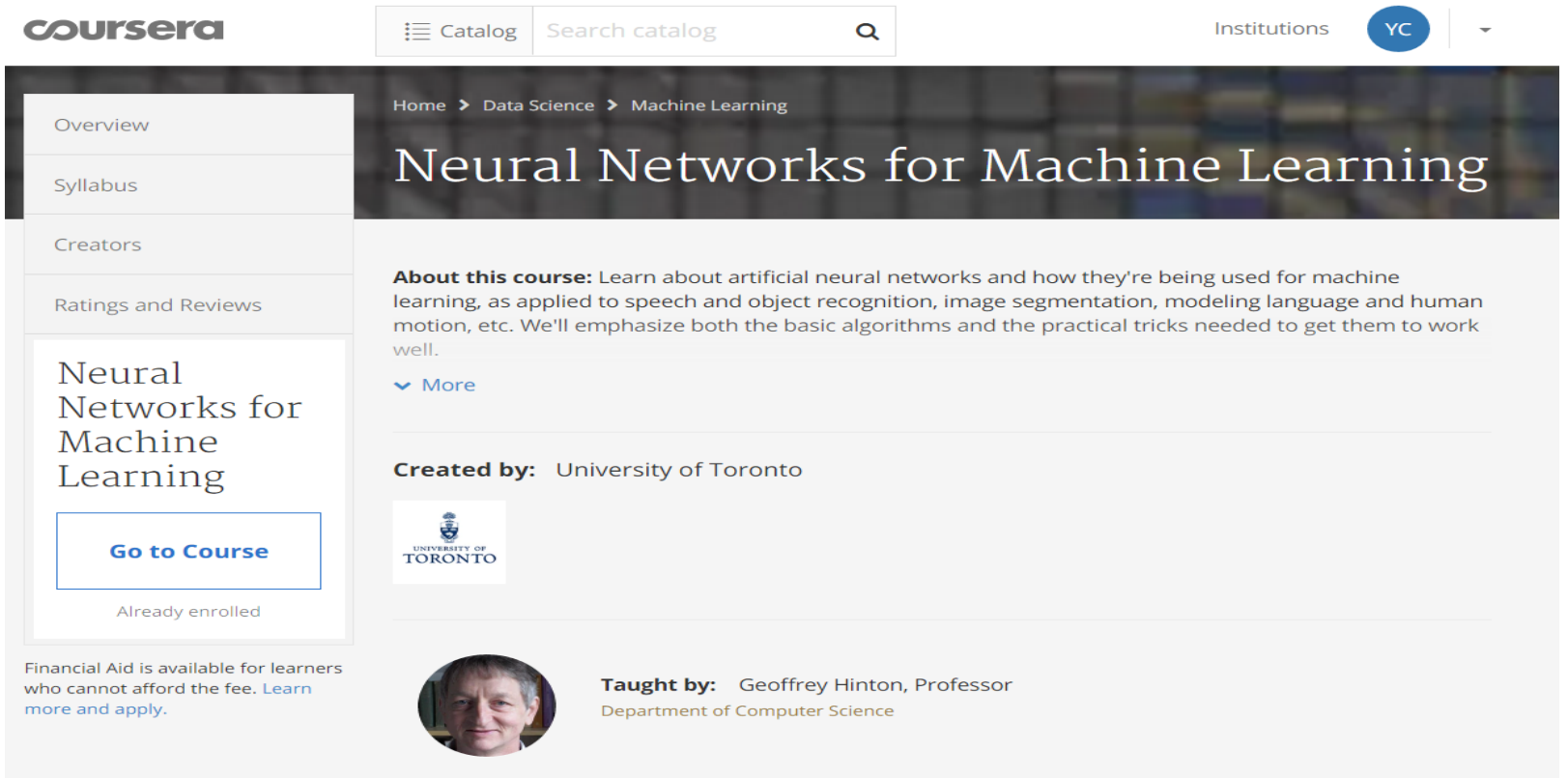
Geoffrey Hinton

Yann LeCun

Andrew Ng

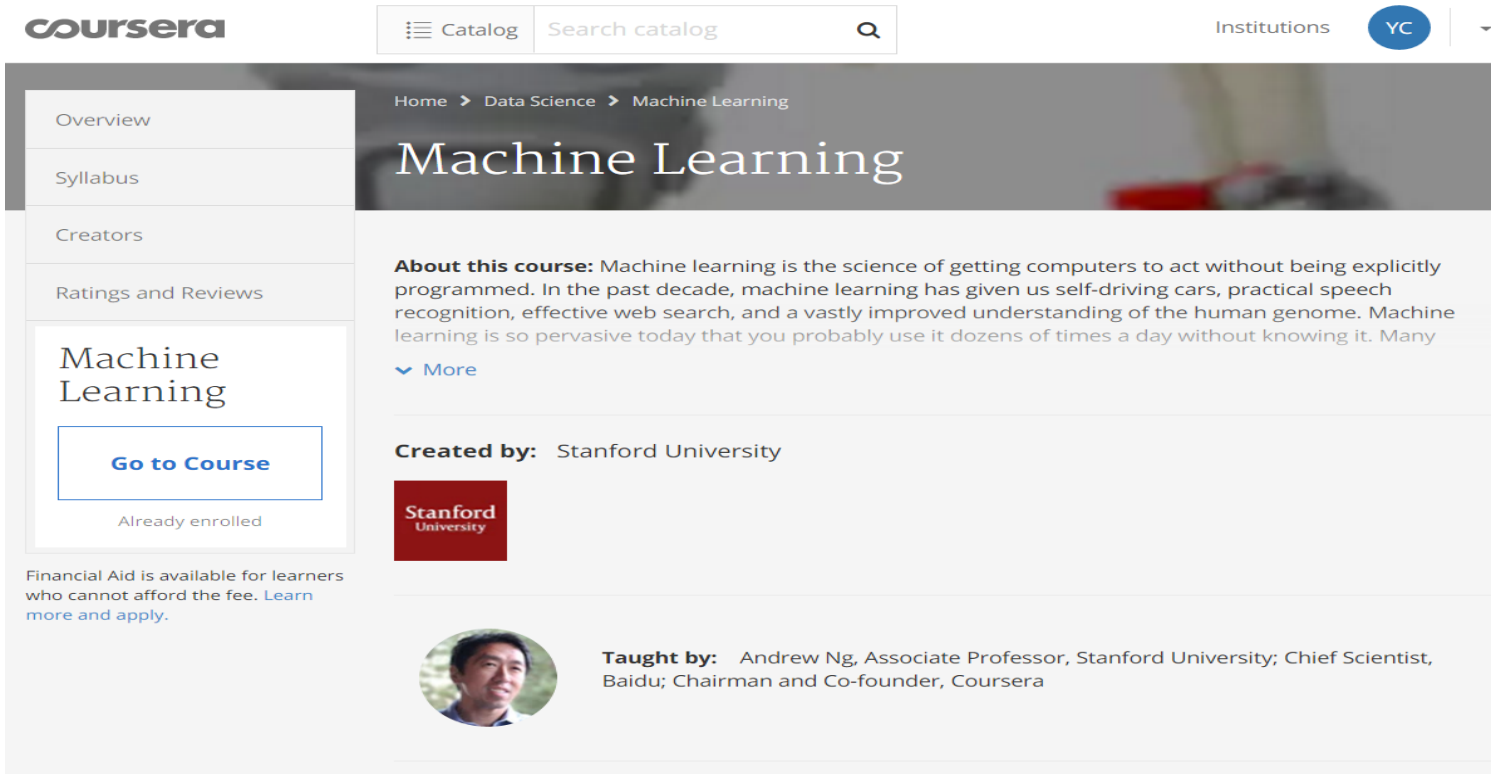
Yoshua Bengio

Lectures for Neural Network



[Neural Networks for Machine Learning - University of Toronto]

Lectures for Neural Network



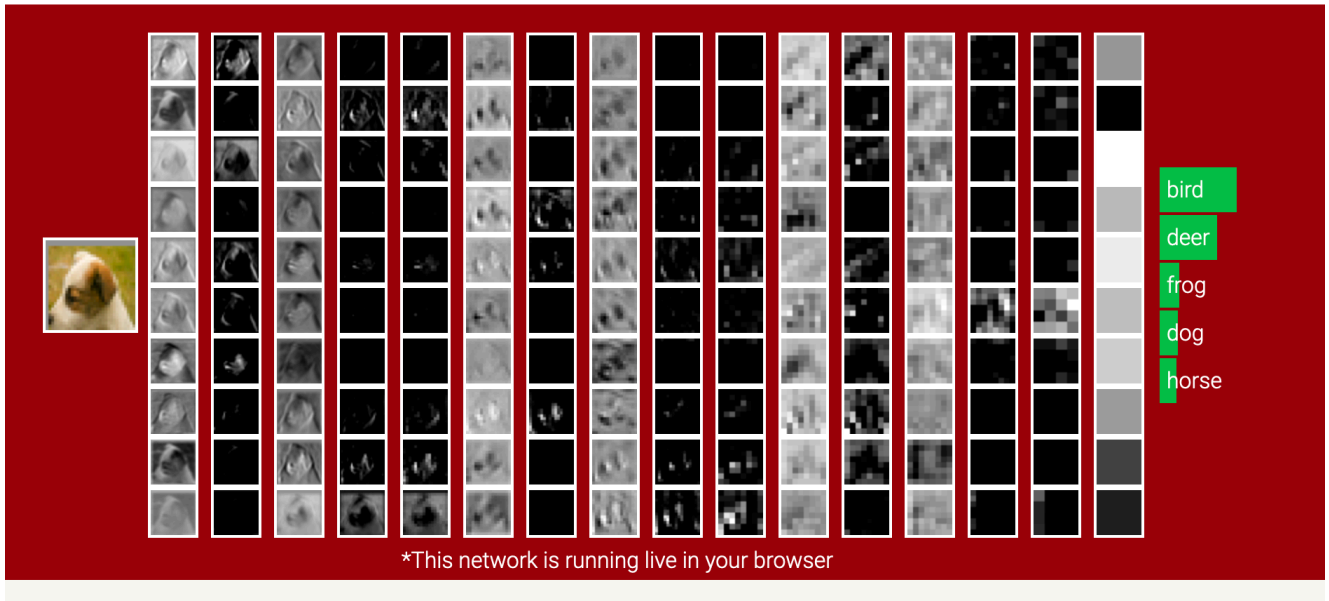
[Machine Learning - Stanford University]

Lectures for Neural Network

CS231n: Convolutional Neural Networks for Visual Recognition

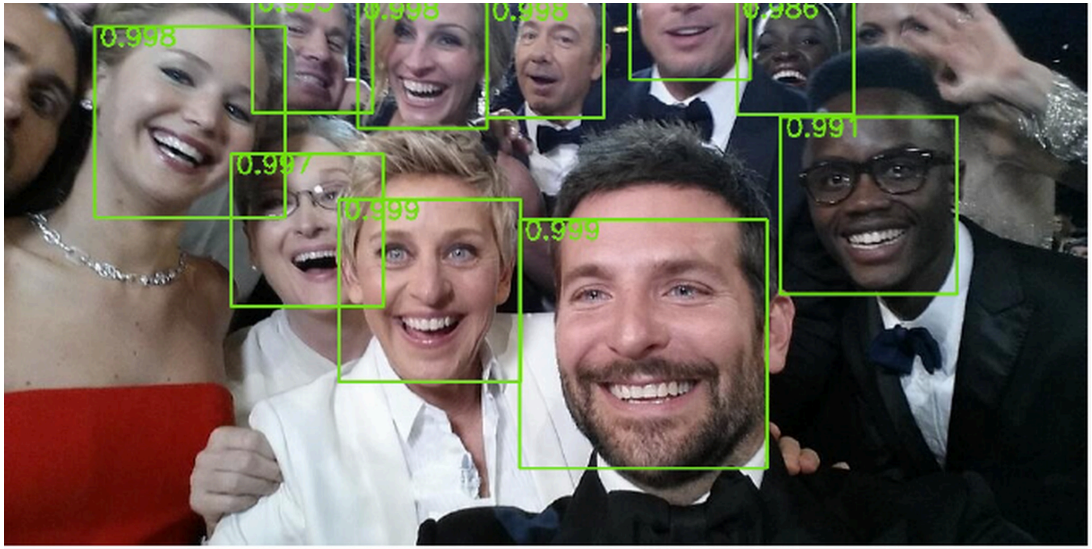
Spring 2019

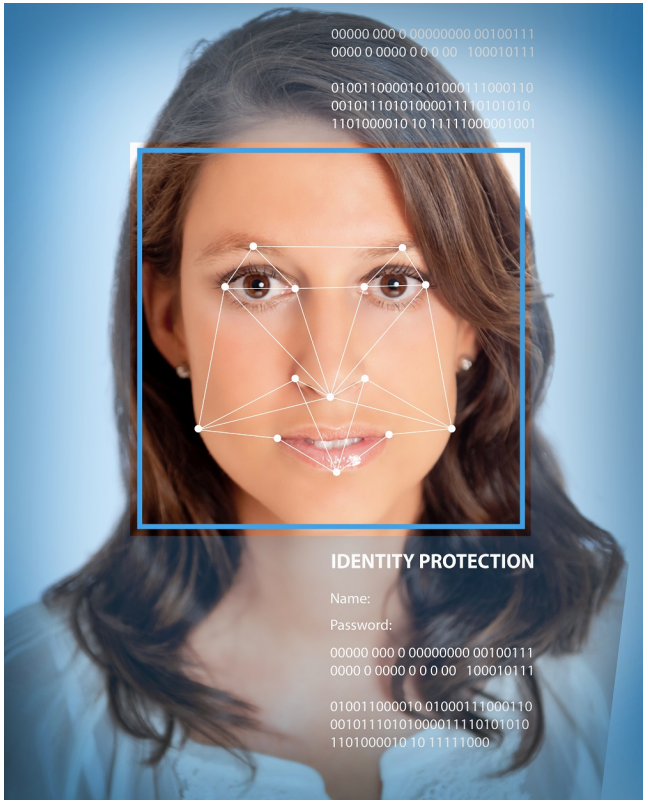
Previous Years: [Winter 2015] [Winter 2016] [Spring 2017] [Spring 2018]



CS231n – Stanford University

Face Detection & Recognition





Object Detection & Recognition

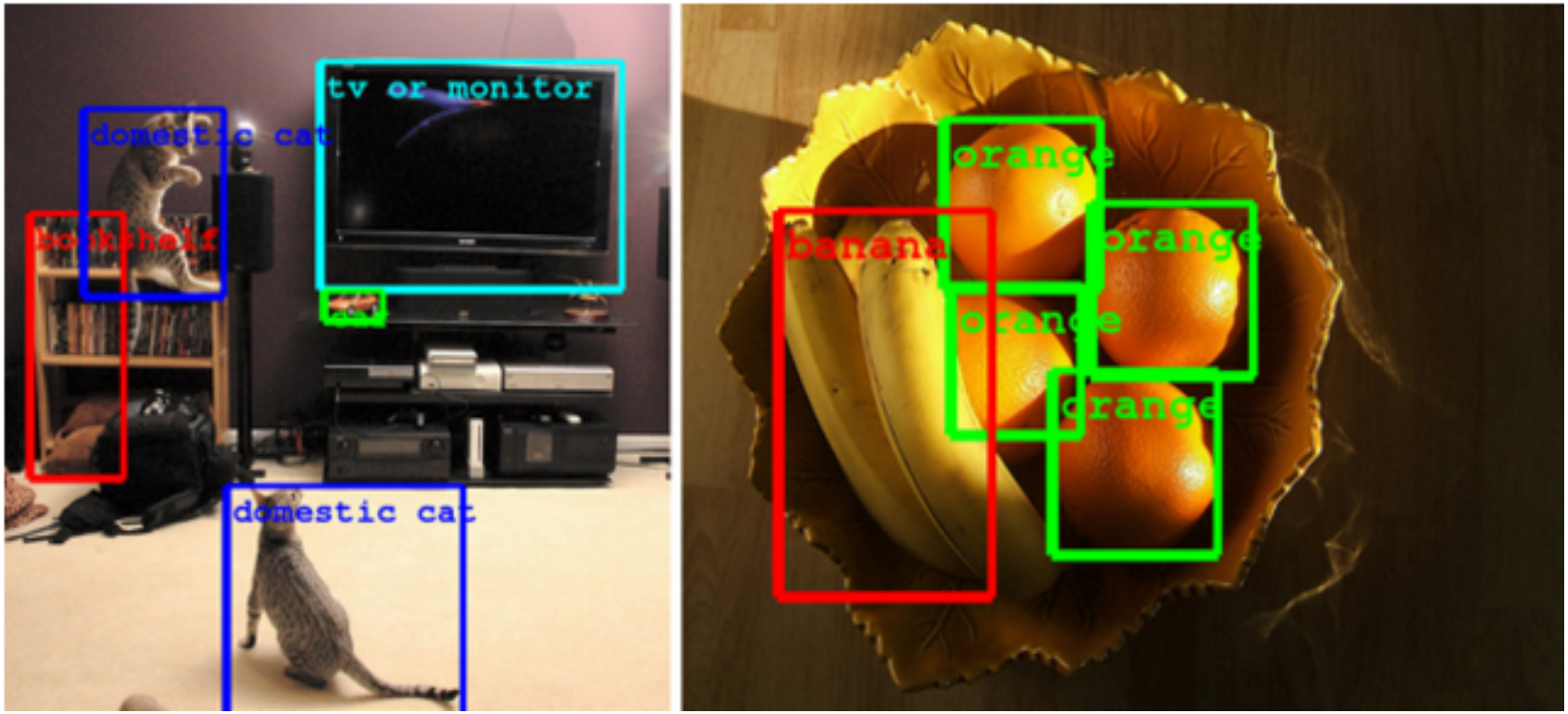
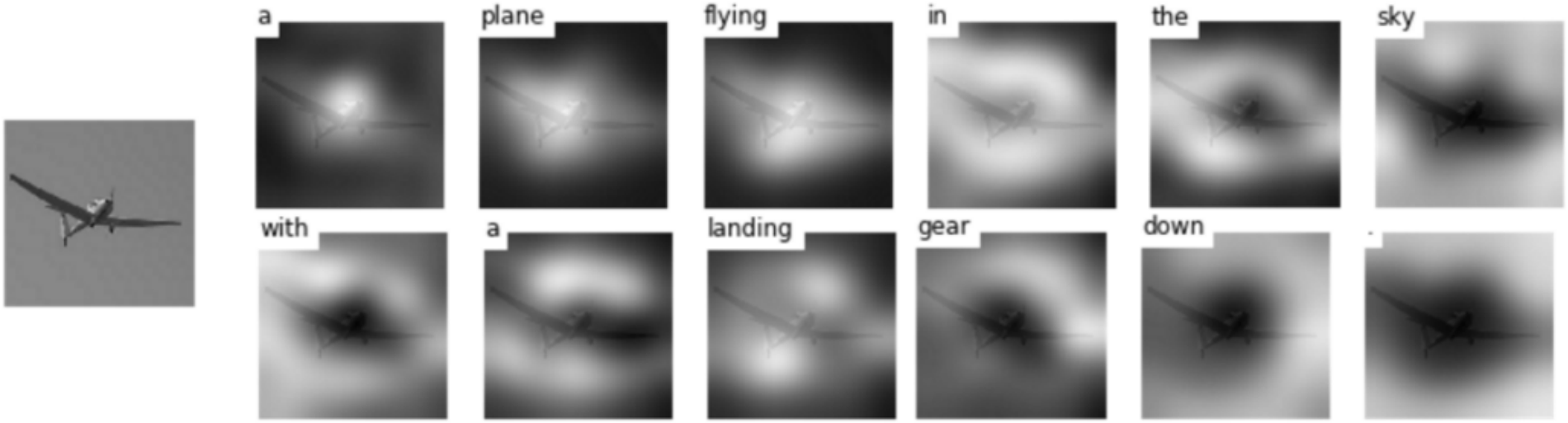
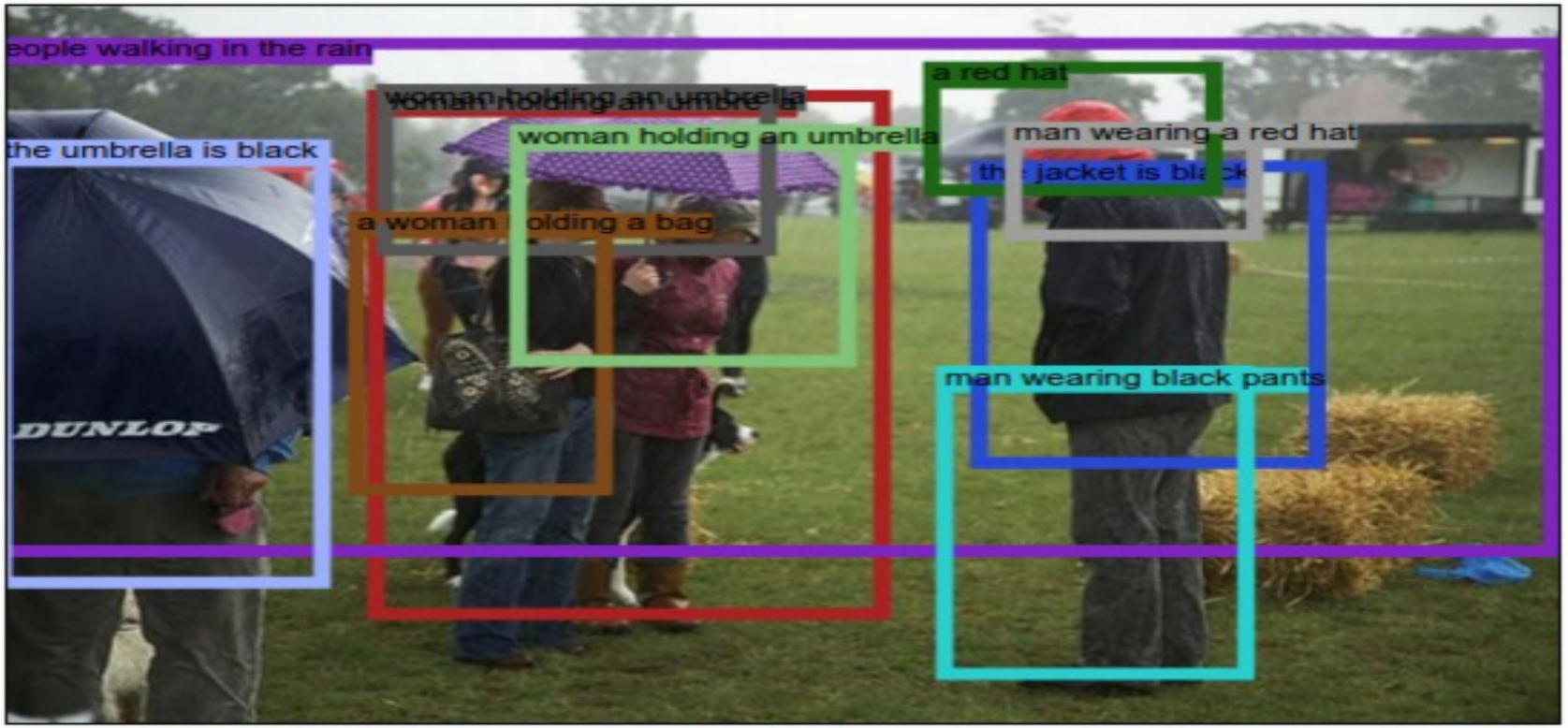


Image Captioning



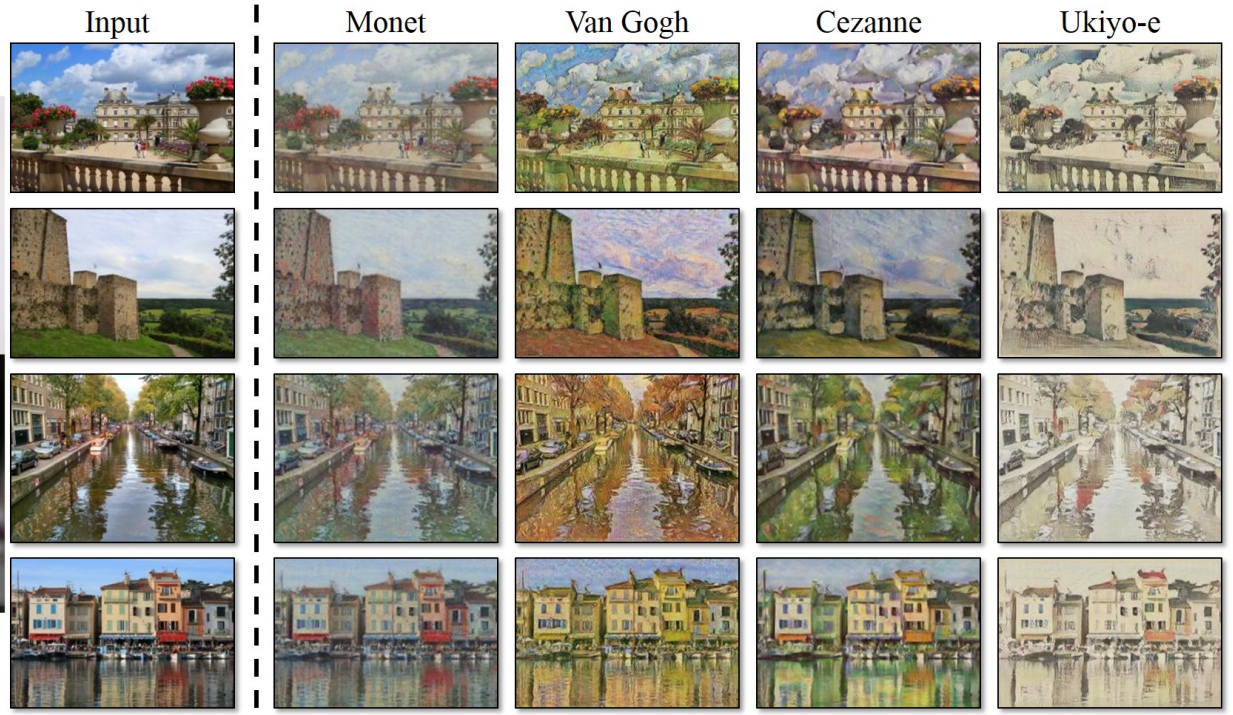
https://github.com/yunjey/show-attend-and-tell

Image Captioning



DenseCap: Fully Convolutional Localization Networks for Dense Captioning

Image Generation/Translation



Deep Fake

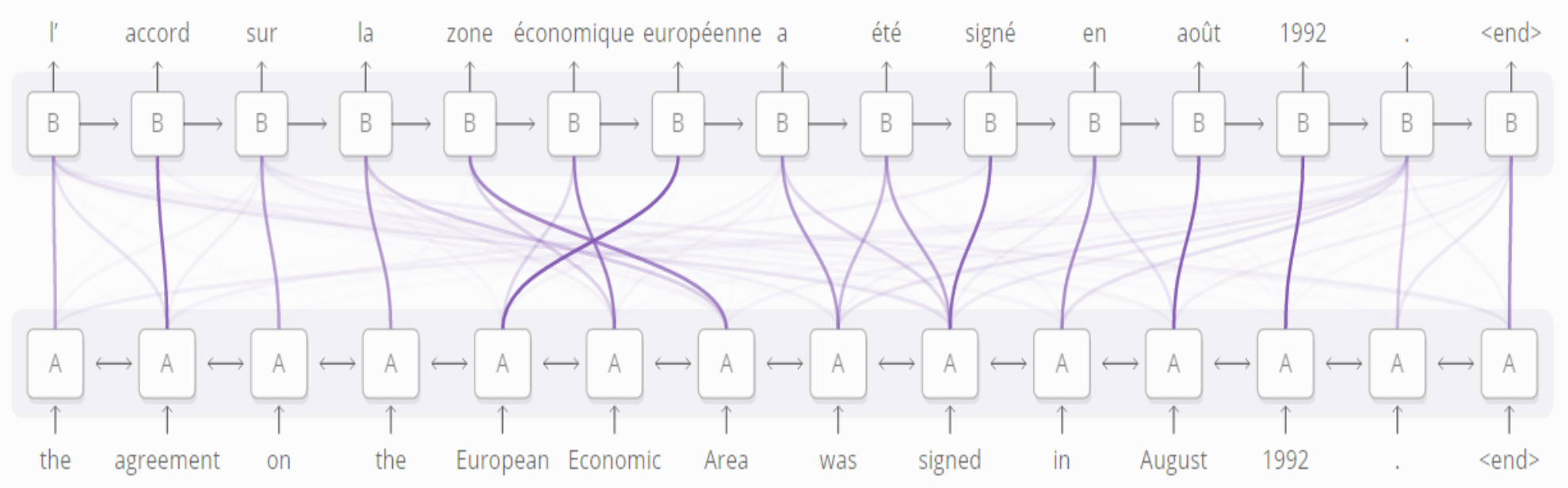
Deepfakes Are Coming. And They're Dangerous.

by Jenna Lifhits | July 20, 2018 06:19 AM

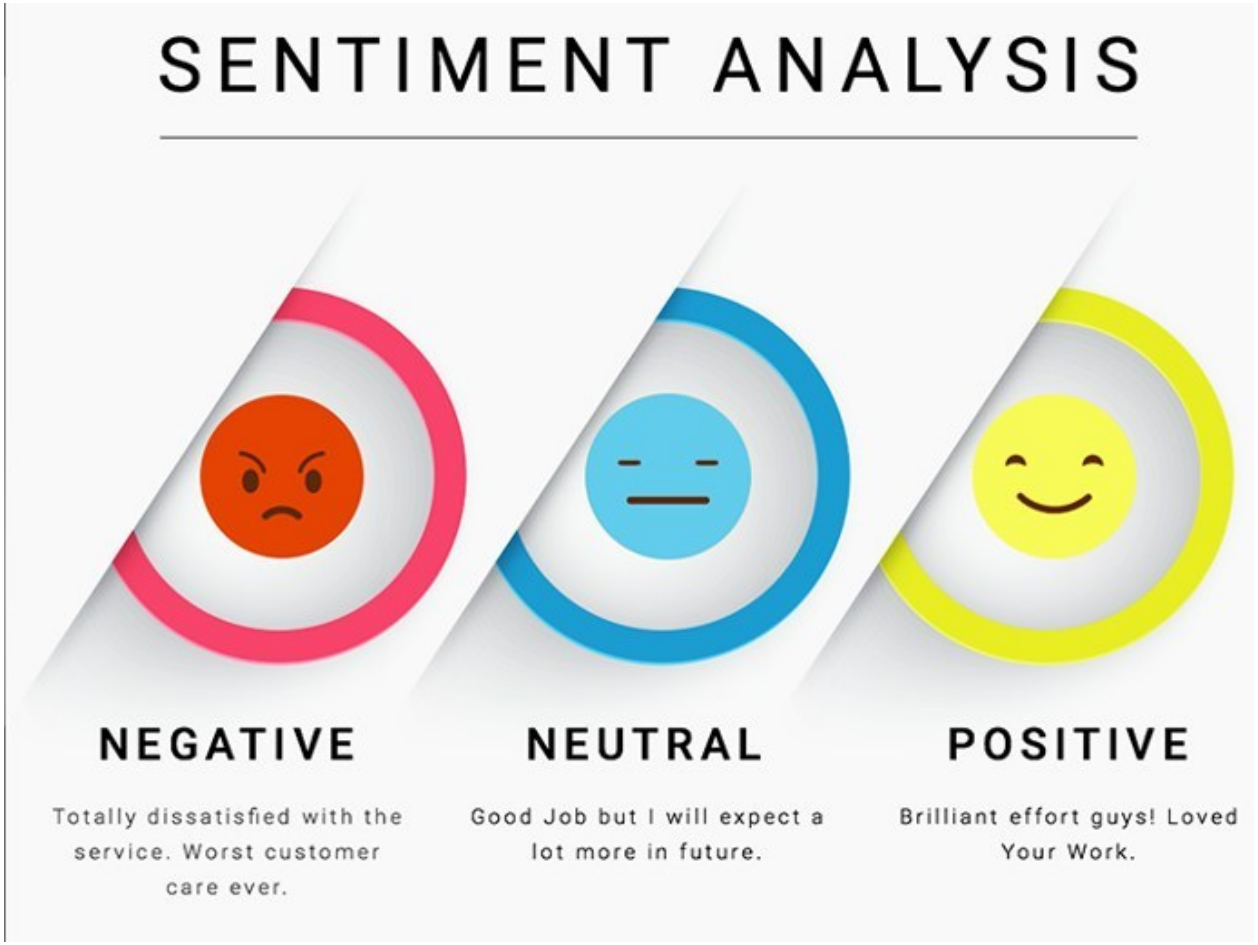
Marco Rubio warns that the United States is not ready for the havoc that impersonation technology can wreak.

https://www.youtube.com/watch?v=tISBgFxY3SE

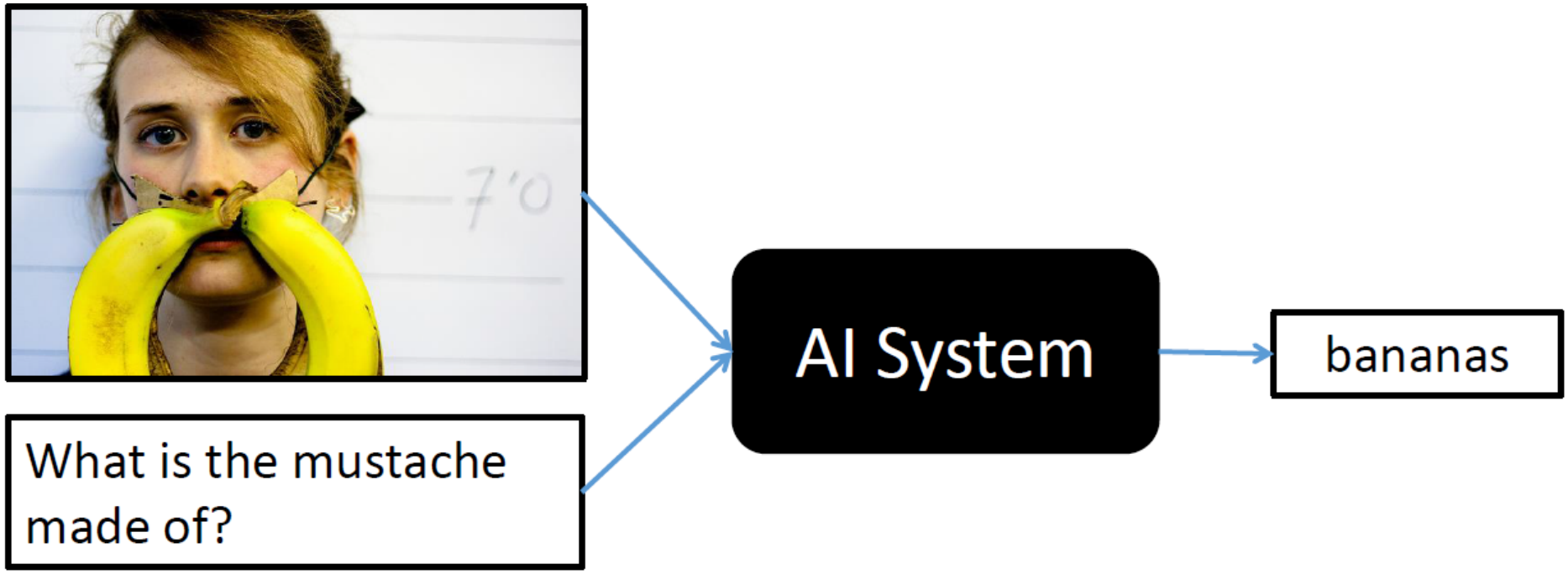
Machine Translation



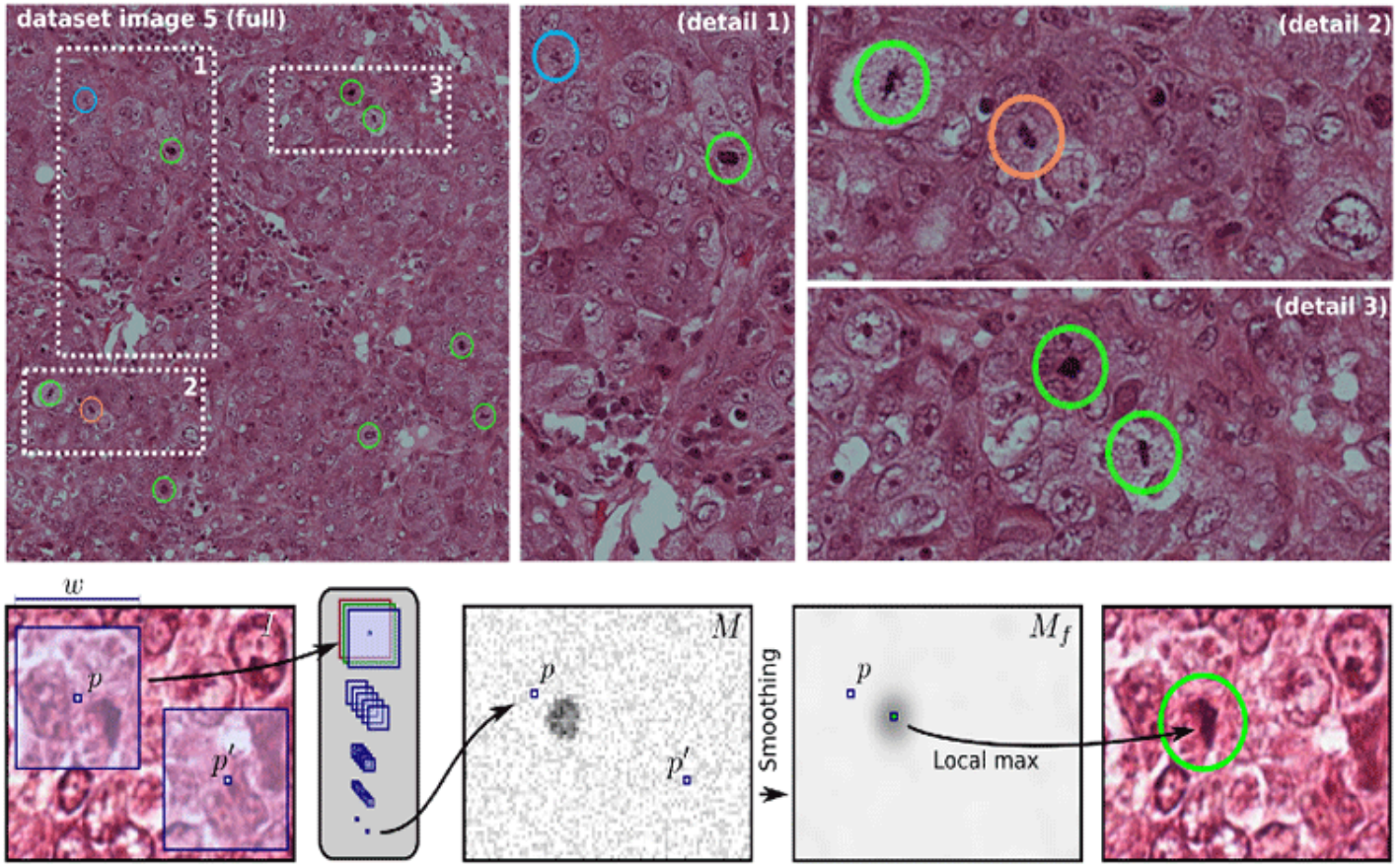
Sentiment Analysis



Question Answering

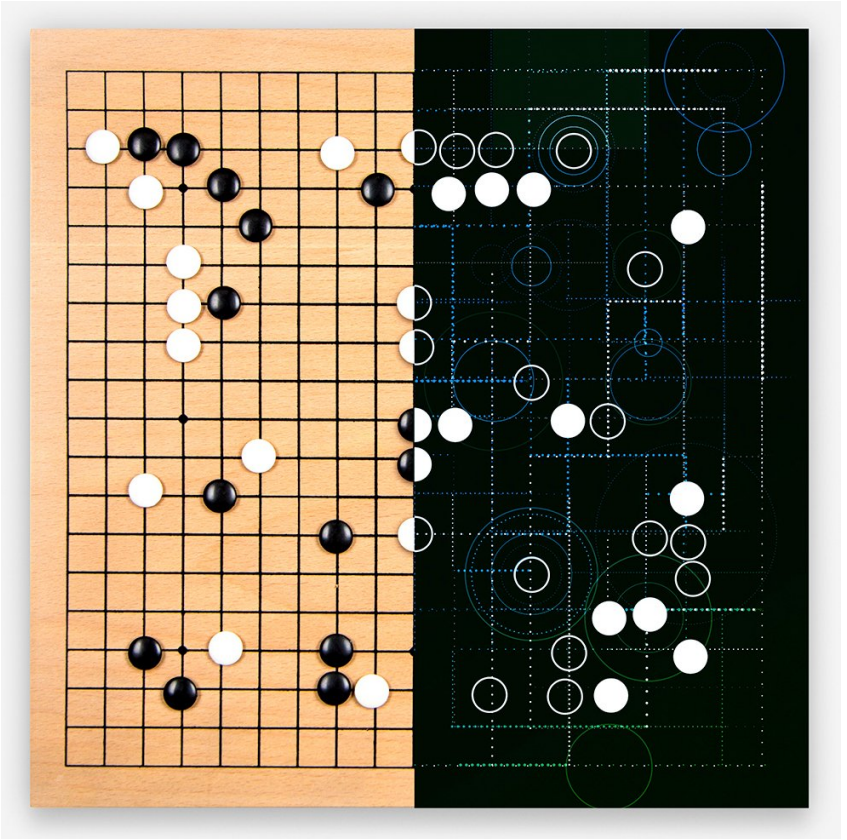


Medical Analysis



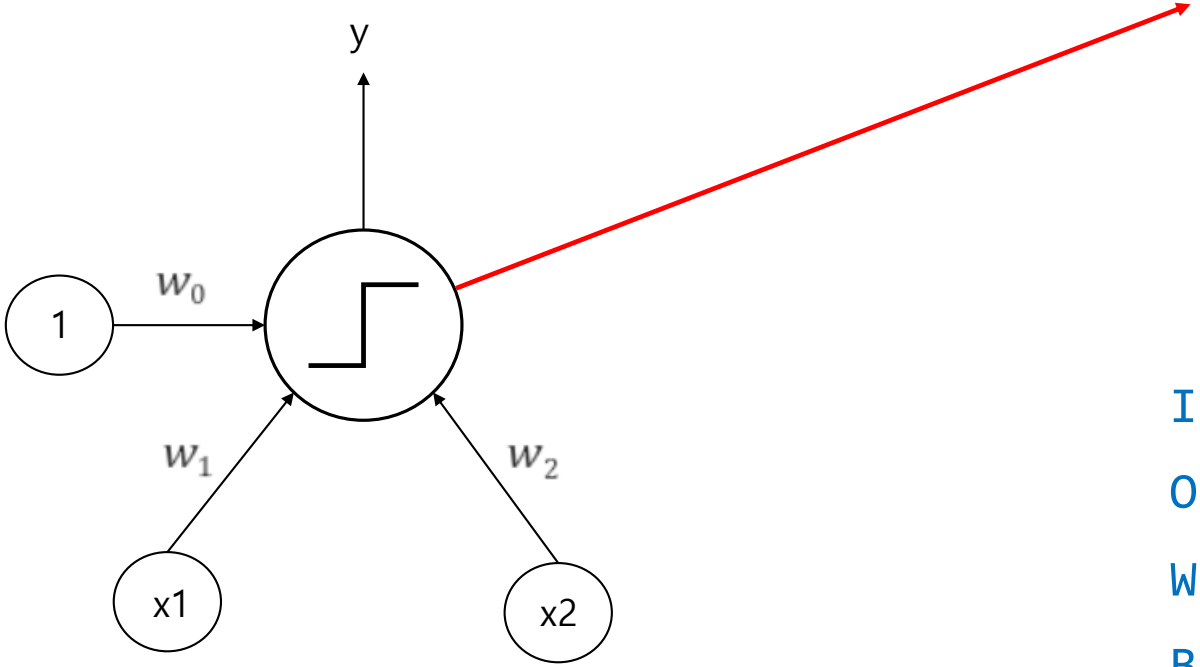
Recommender System

Game AI

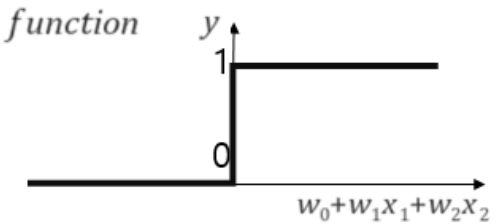


2. Perceptron

Single Layer Perceptron



hard thresholding function



$$y = \begin{cases} 1 & w_0+w_1x_1+w_2x_2 \geq 0 \\ 0 & otherwise \end{cases}$$

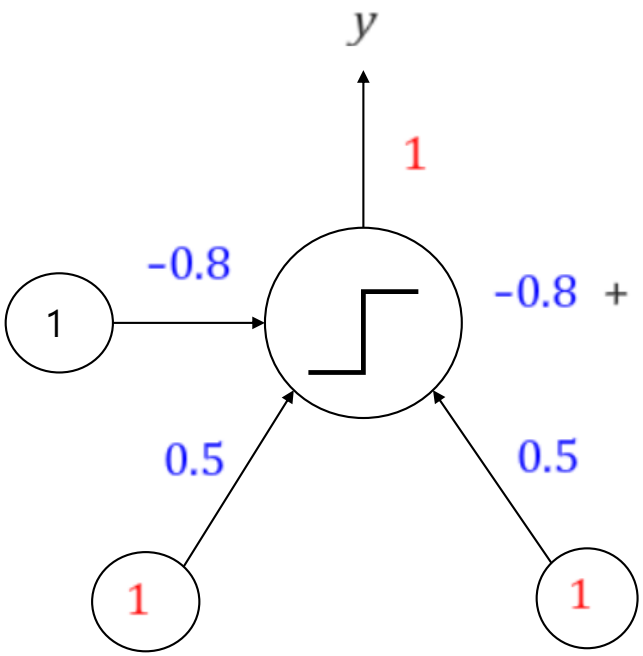
Inputs: x1, x2

Output: y

Weights: w0, w1, w2

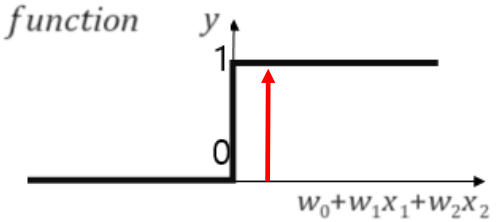
Bias: 1

Single Layer Perceptron



$-0.8 + 0.5\times1 + 0.5\times1 = 0.2$

hard thresholding function

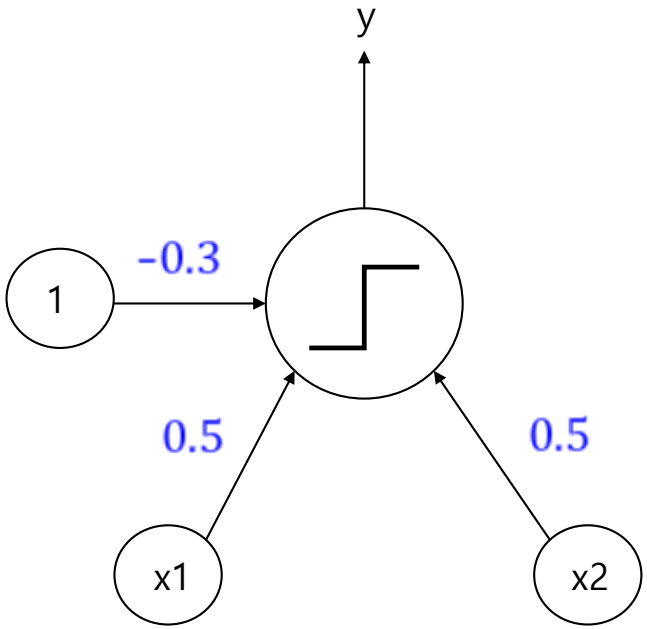


$$y = \begin{cases} 1 & w_0+w_1x_1+w_2x_2 \geq 0 \\ 0 & otherwise \end{cases}$$

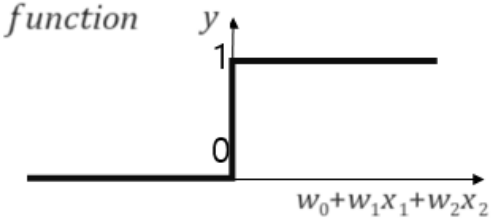
AND Gate

x_1	x_2	y
0	0	0
0	1	0
1	0	0
1	1	1

Single Layer Perceptron



hard thresholding function



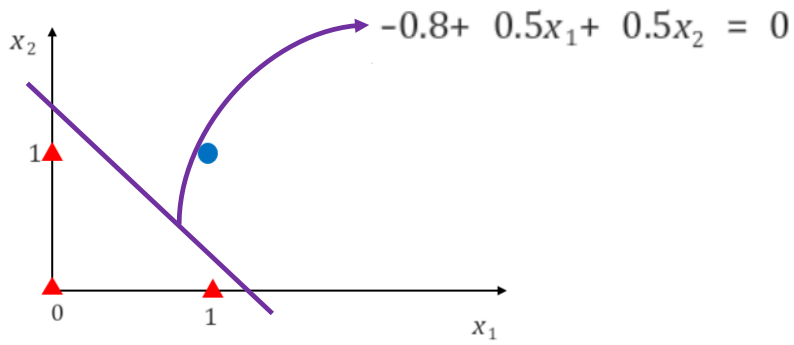
$$y = \begin{cases} 1 & w_0+w_1x_1+w_2x_2 \geq 0 \\ 0 & otherwise \end{cases}$$

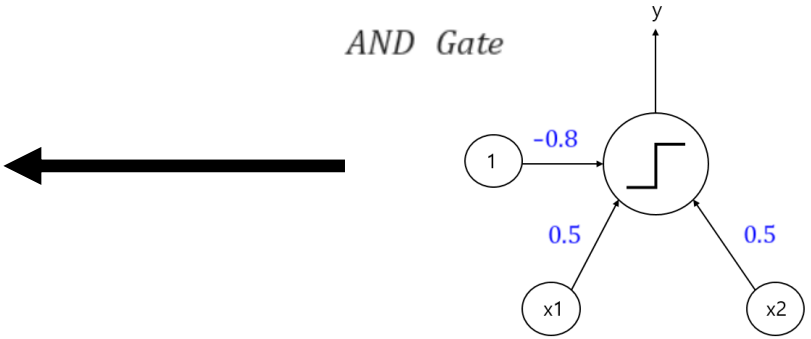
OR Gate

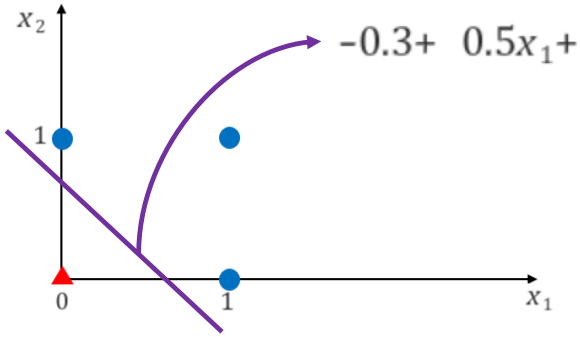
x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	1

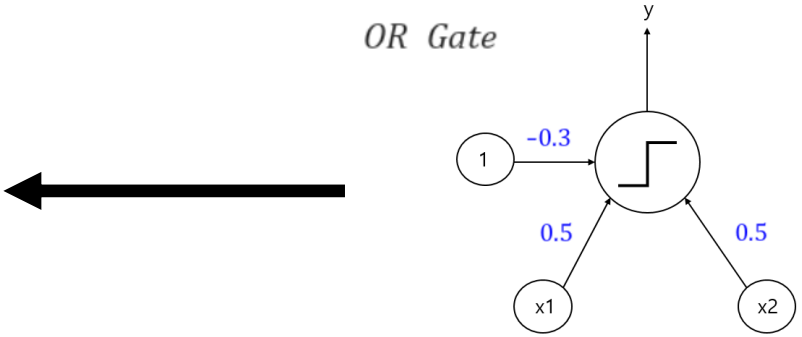
Single Layer Perceptron

● : 1 ▲ : 0





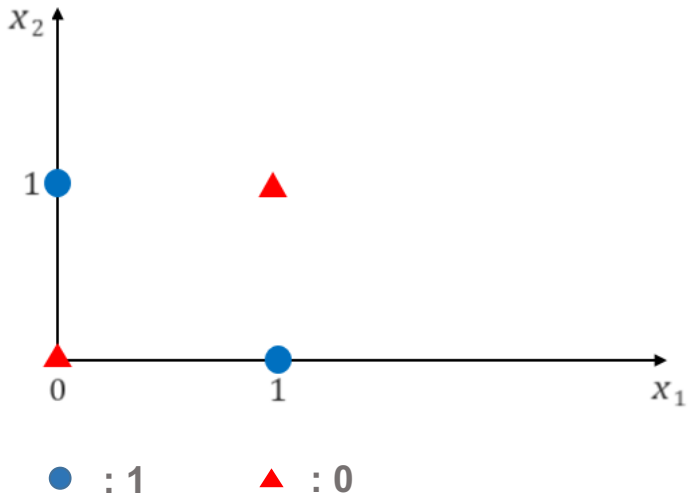




XOR Gate

Is it possible to solve XOR problem using a single layer perceptron?

No. Single layer perceptron can only solve linear problem. XOR problem is non-linear.

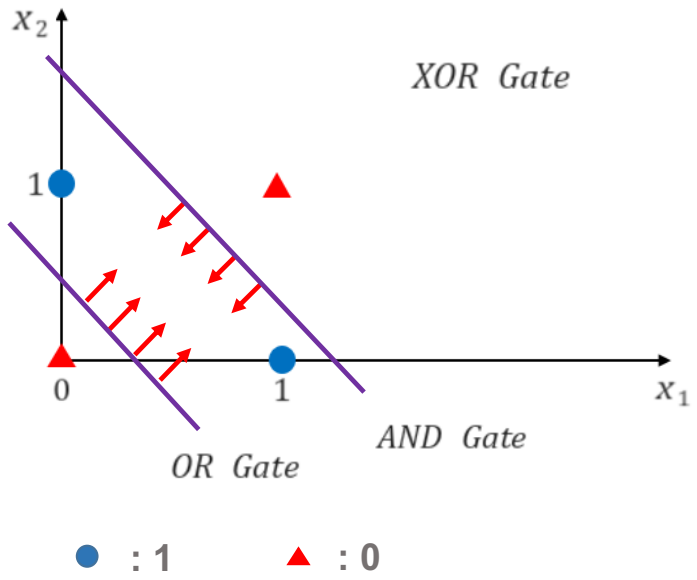


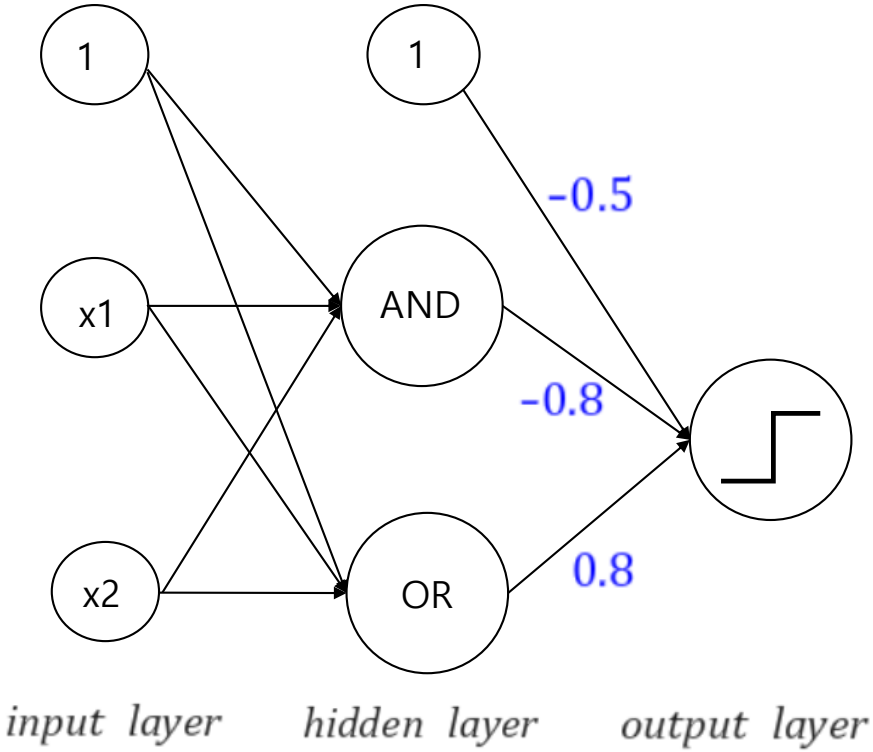
XOR Gate

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

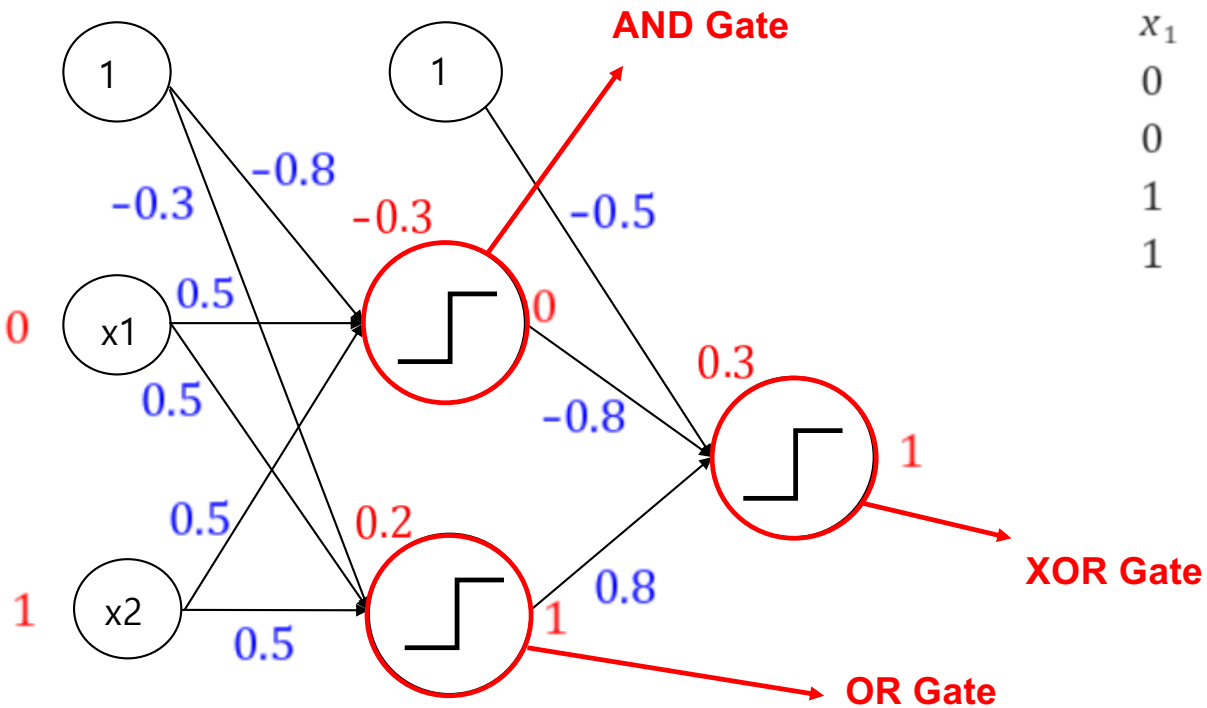
Multi Layer Perceptron

But if we use 2 single layer perceptron, we can solve XOR problem. This model is called multi layer perceptron.





Multi Layer Perceptron



OR Gate

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	1

AND Gate

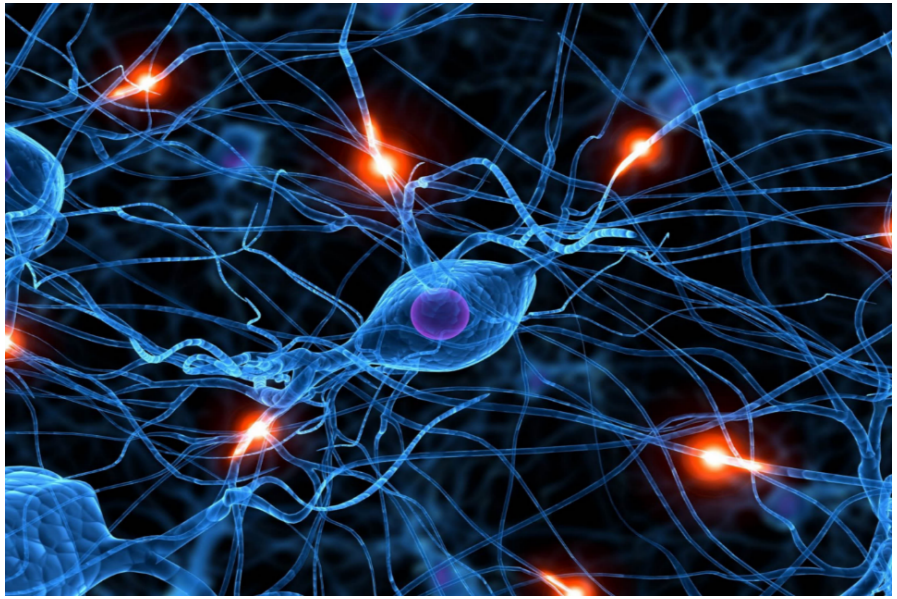
x_1	x_2	y
0	0	0
0	1	0
1	0	0
1	1	1

XOR Gate

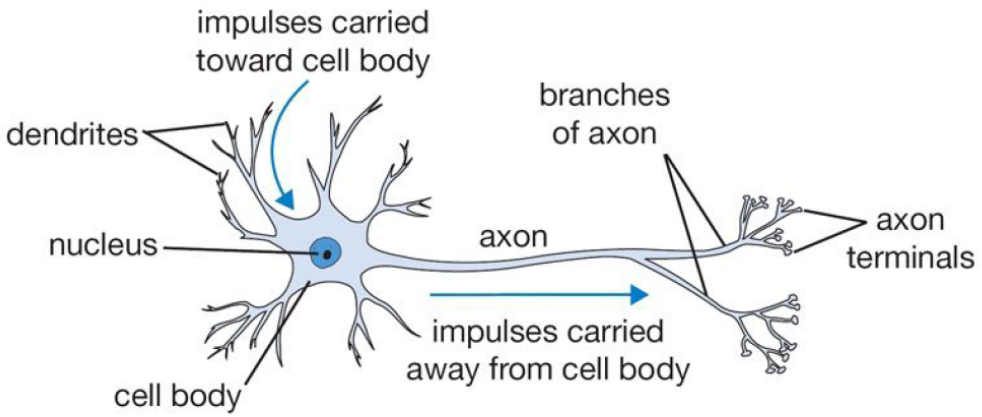
x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

3. Neural Network

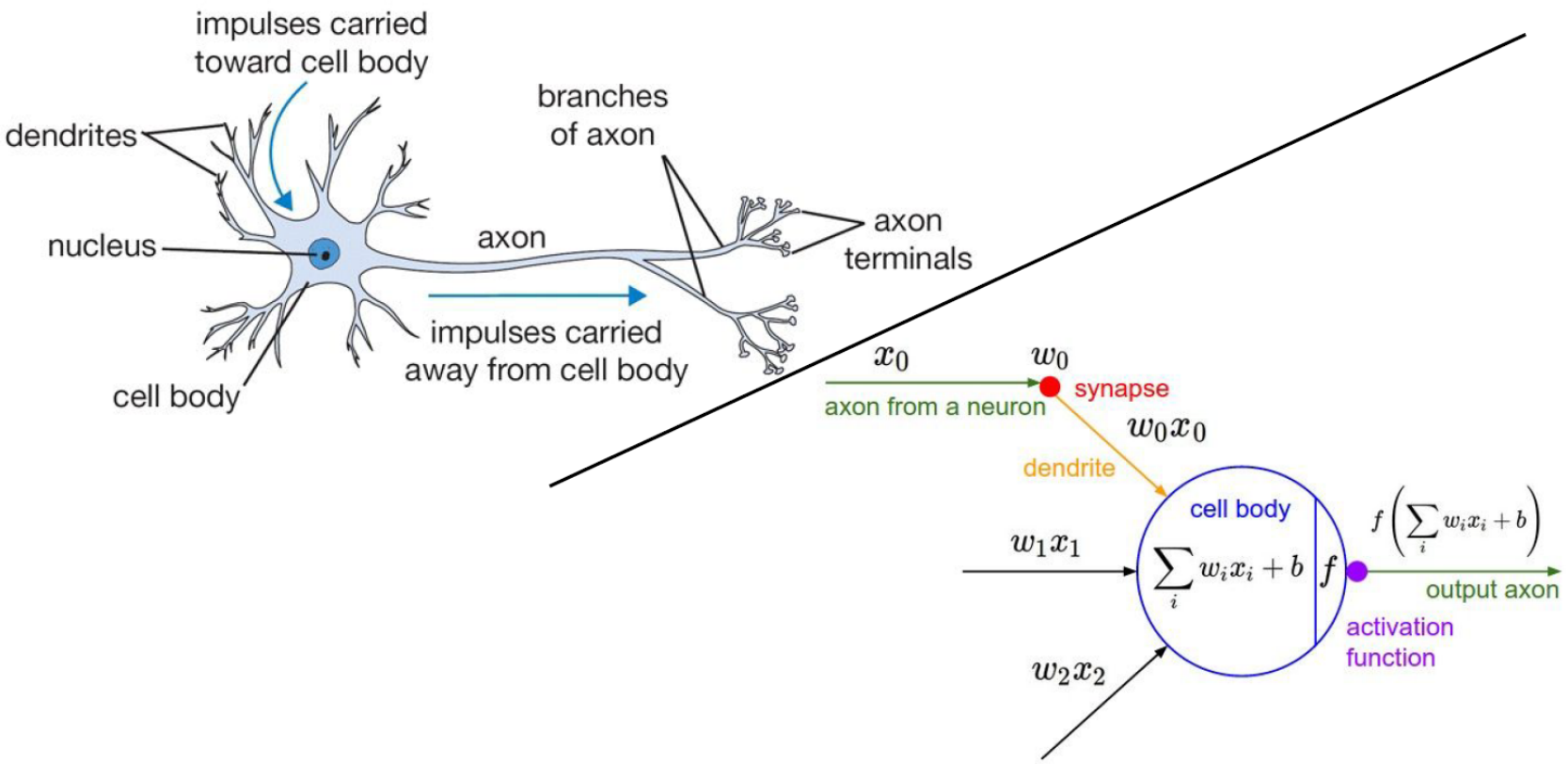
Human Brain and Neural Network



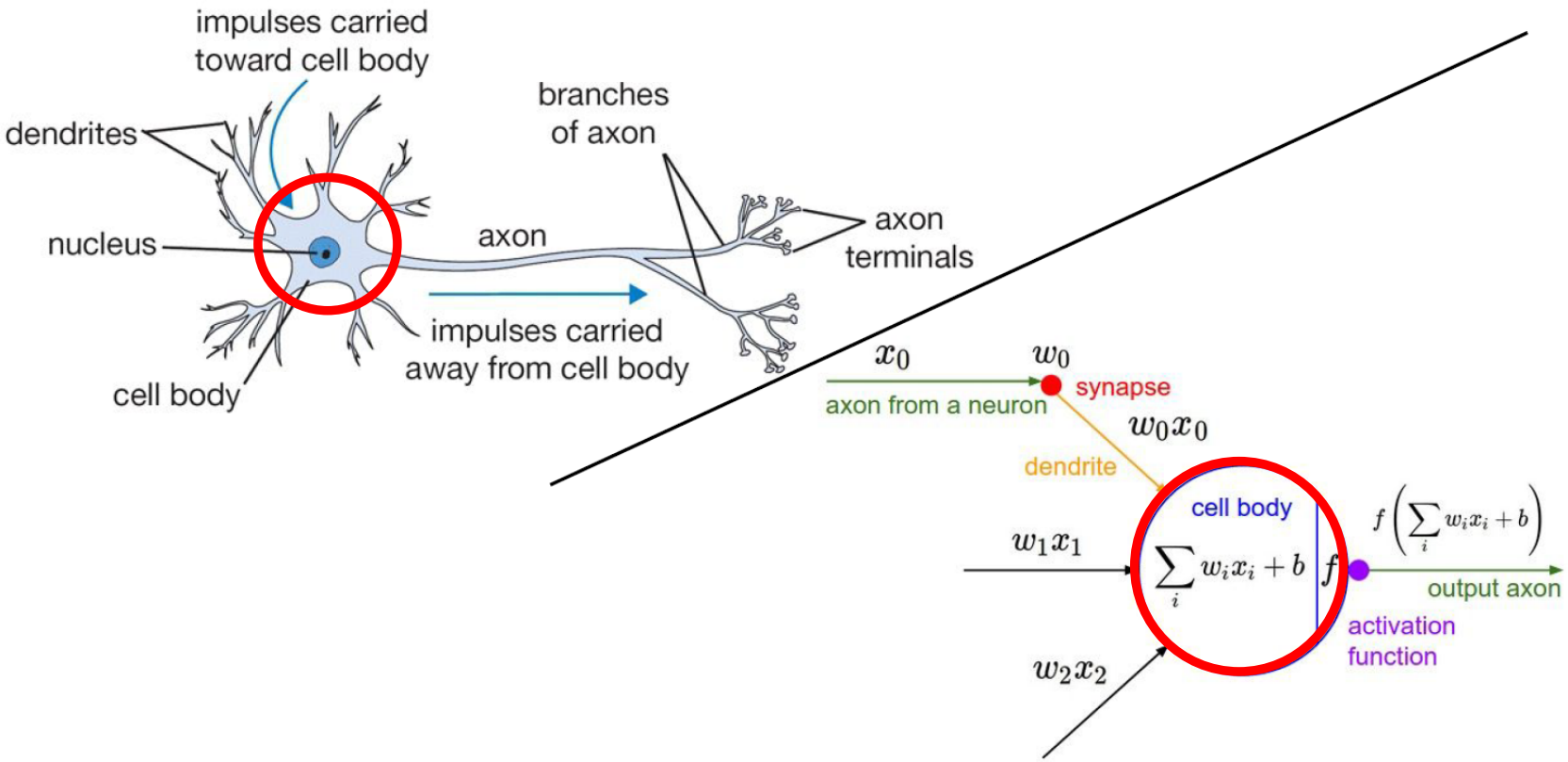
Human Brain and Neural Network



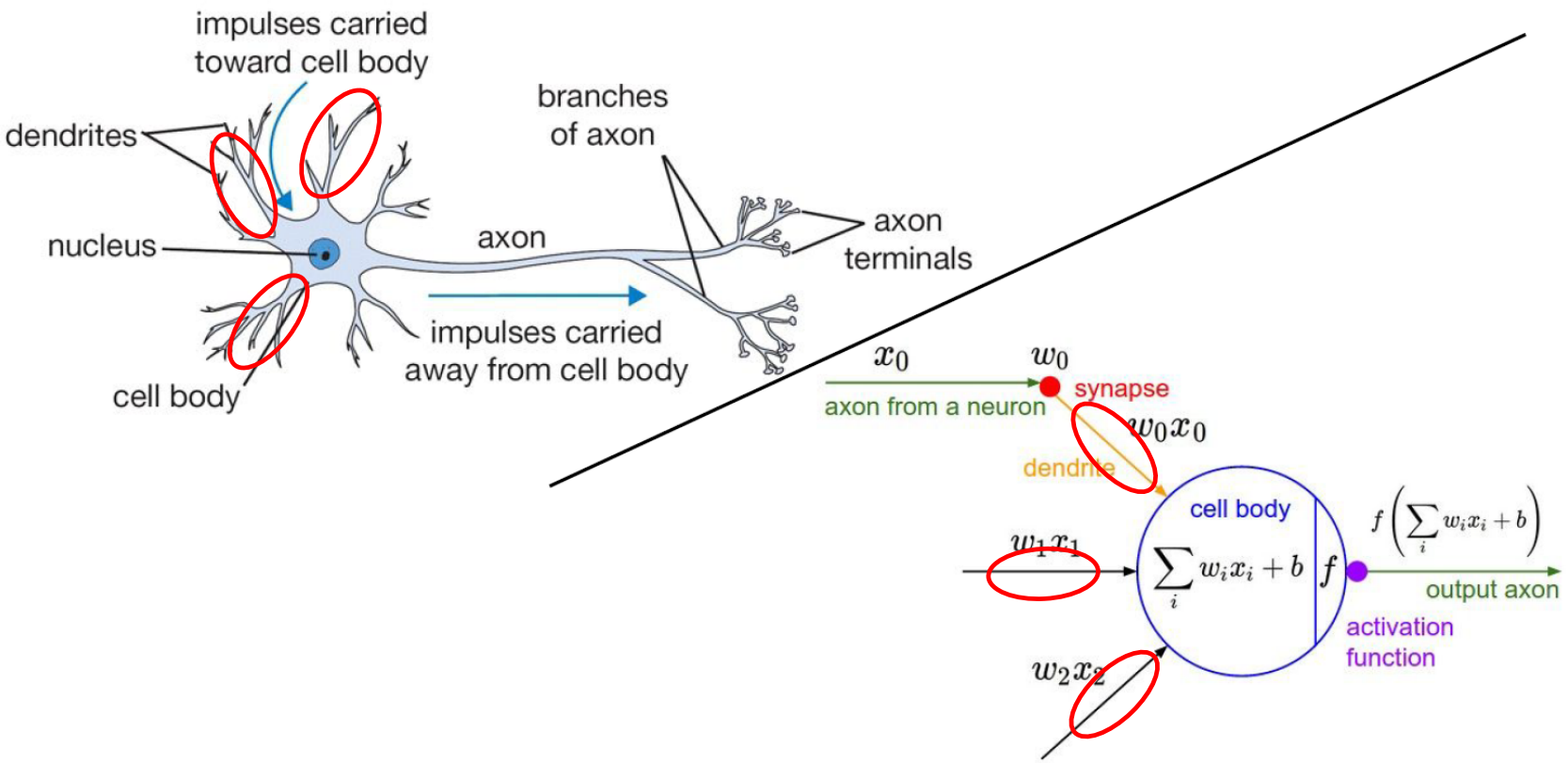
Human Brain and Neural Network



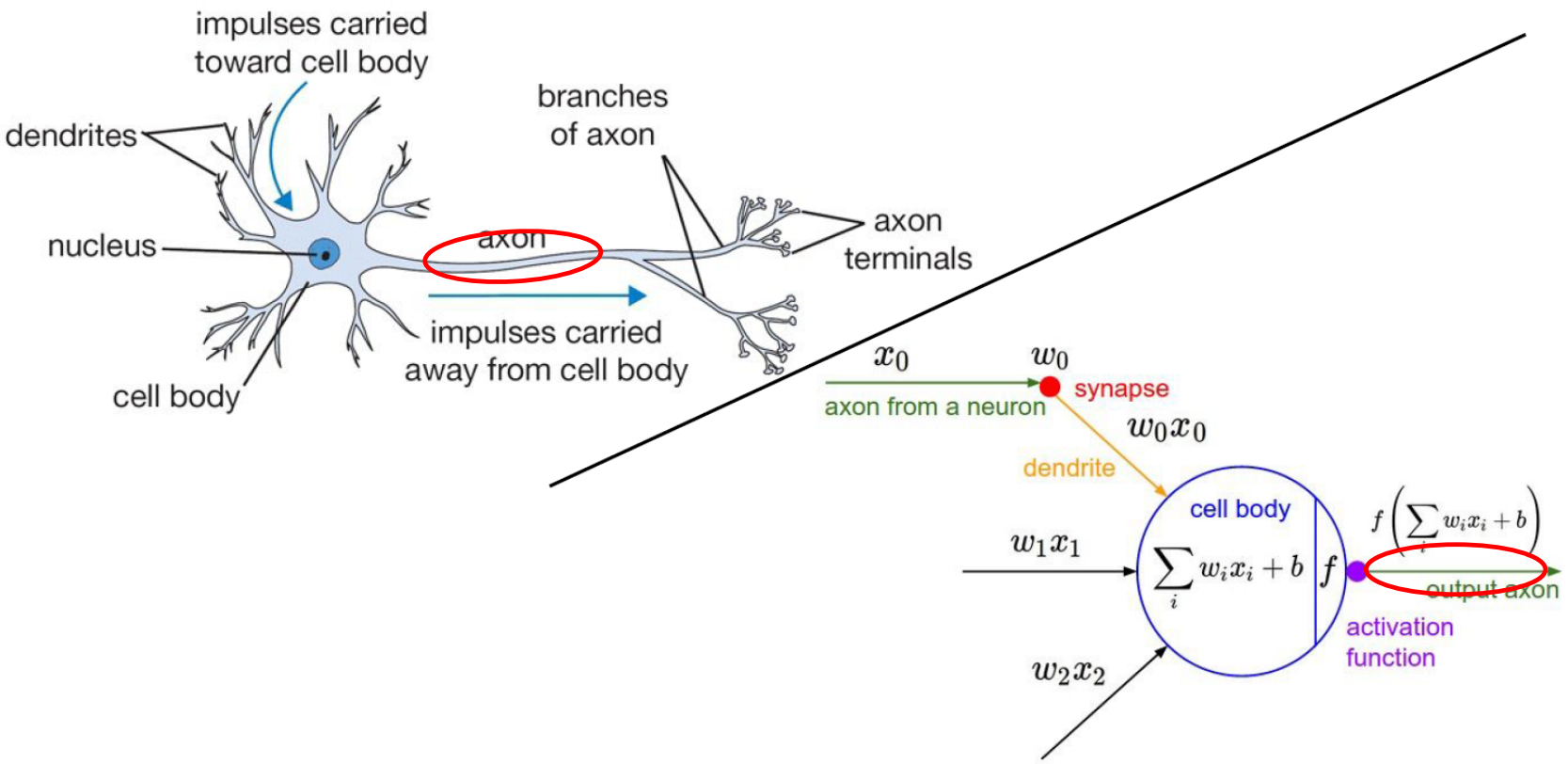
Human Brain and Neural Network



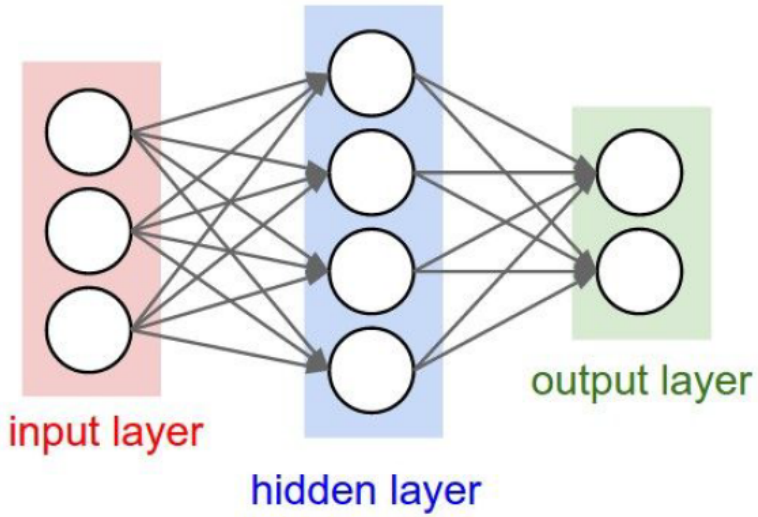
Human Brain and Neural Network



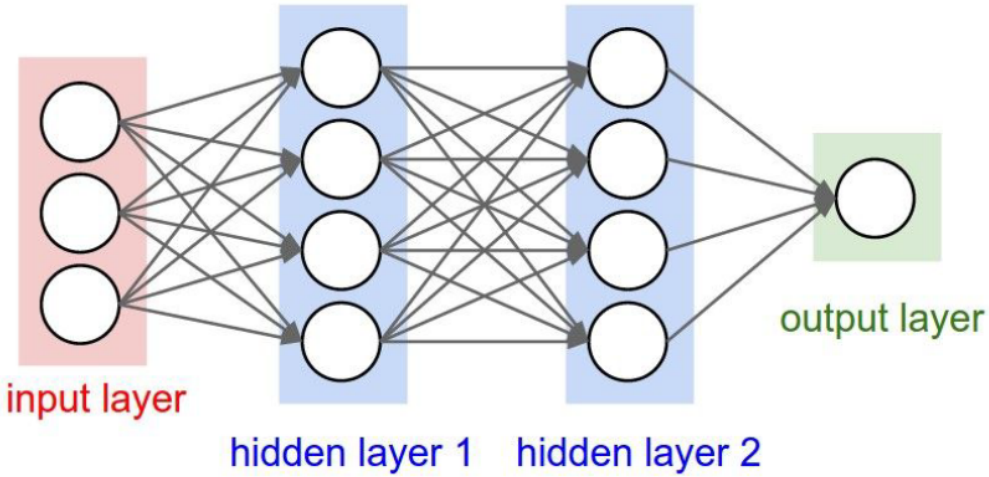
Human Brain and Neural Network



Neural Network Architecture

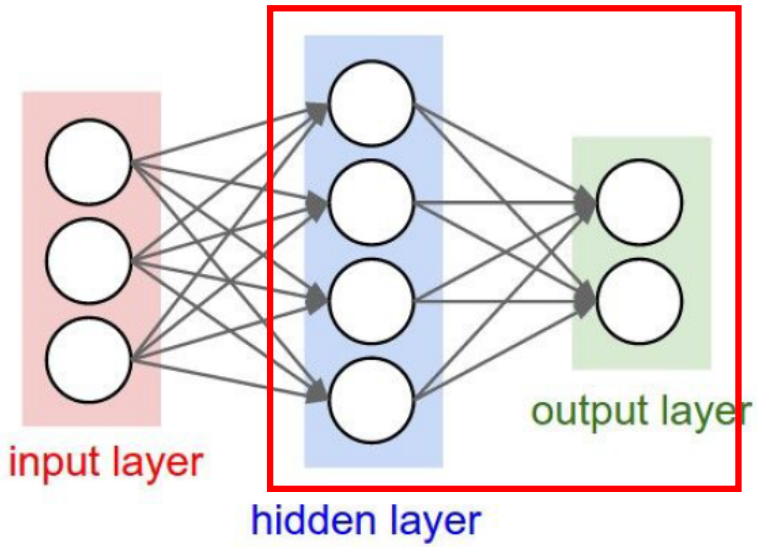


"2-layer Neural Network"
or
"1-hidden-layer Neural Network"

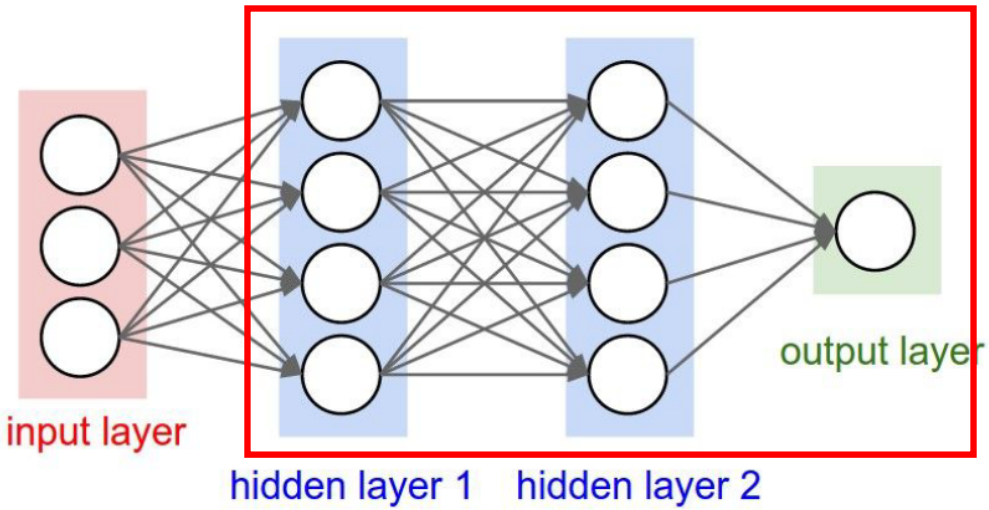


"3-layer Neural Network"
or
"2-hidden-layer Neural Network"

Neural Network Architecture

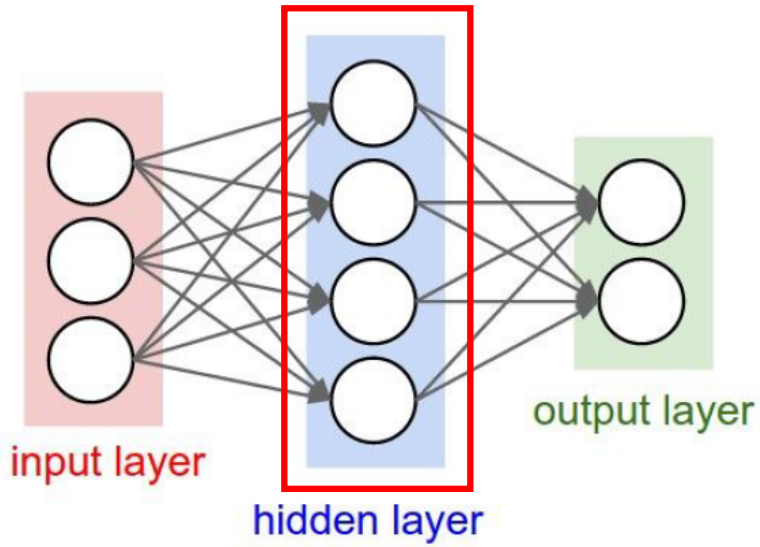


"2-layer Neural Network"
or
"1-hidden-layer Neural Network"

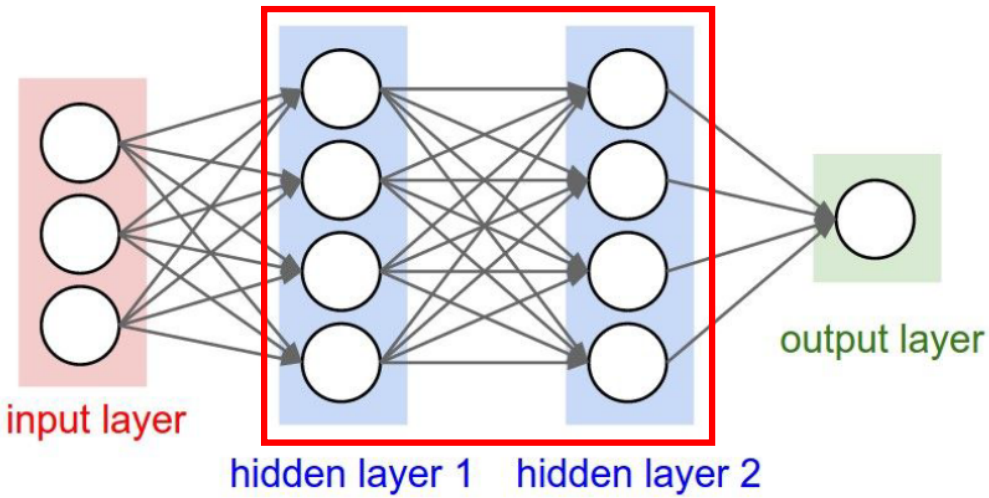


"3-layer Neural Network"
or
"2-hidden-layer Neural Network"

Neural Network Architecture

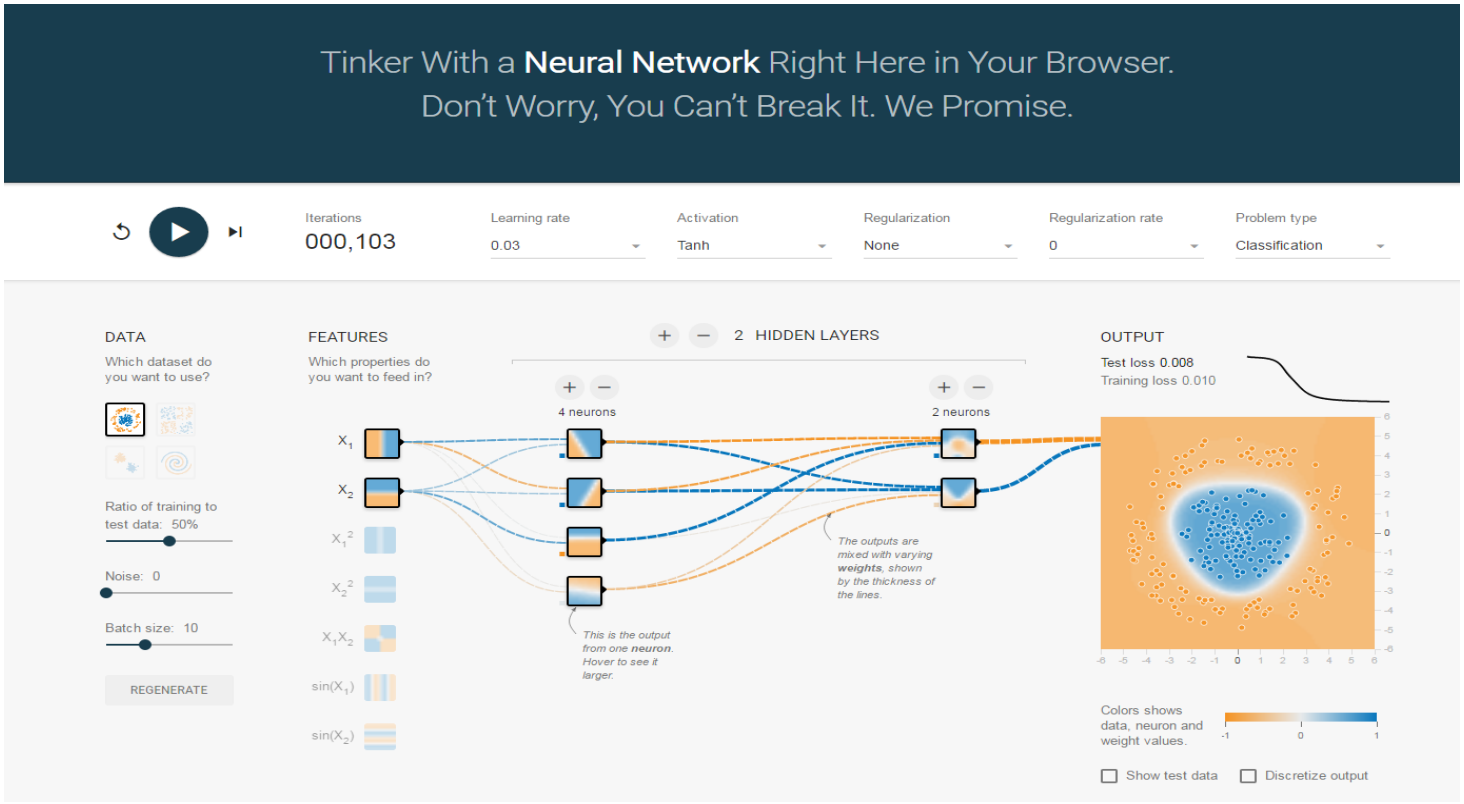


"2-layer Neural Network"
or
"1-hidden-layer Neural Network"



"3-layer Neural Network"
or
"2-hidden-layer Neural Network"

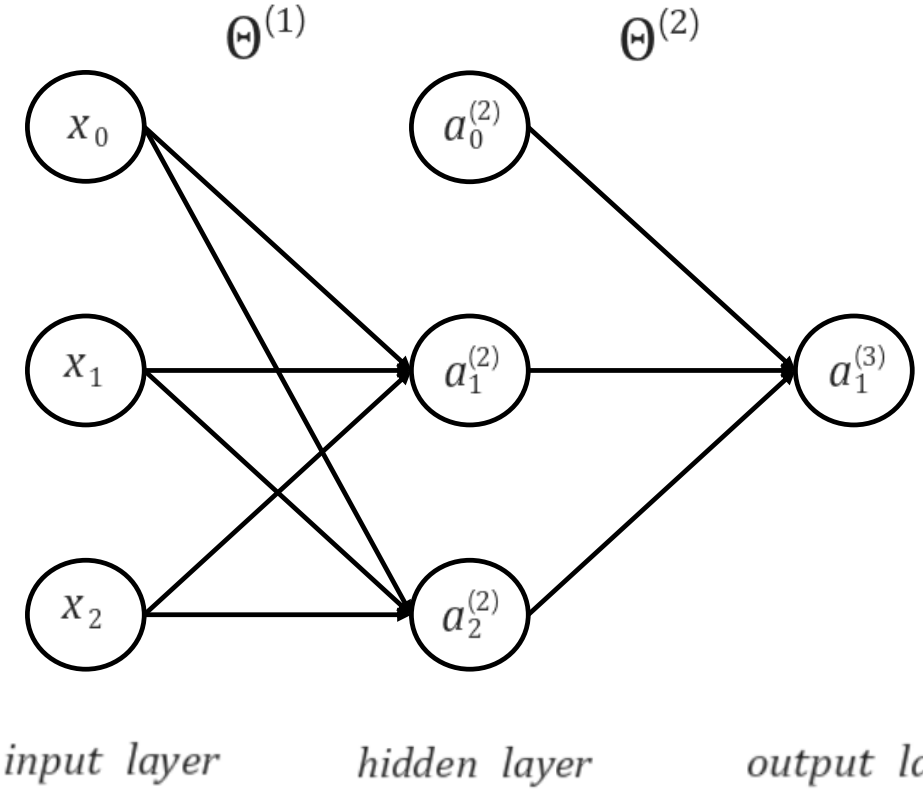
Neural Network



[A Neural Network Playground]

4. Forward Propagation

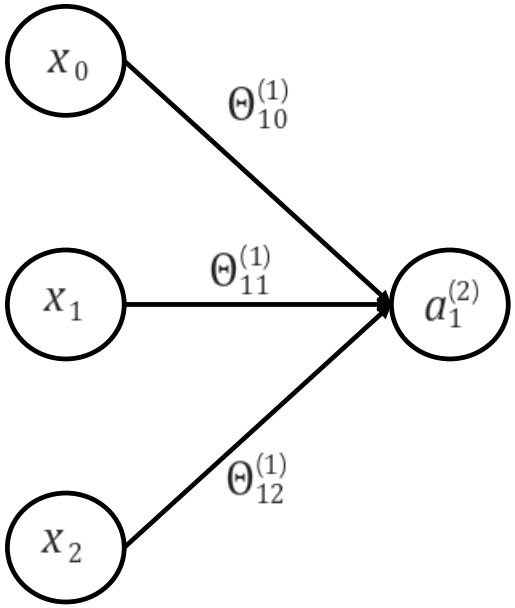
Forward Propagation



a_i^j : "activation" of i-th unit in j-th layer

$\Theta^{(j)}$: matrix of weights controlling function mapping form j-th layer to (j+1)-th layer

Forward Propagation

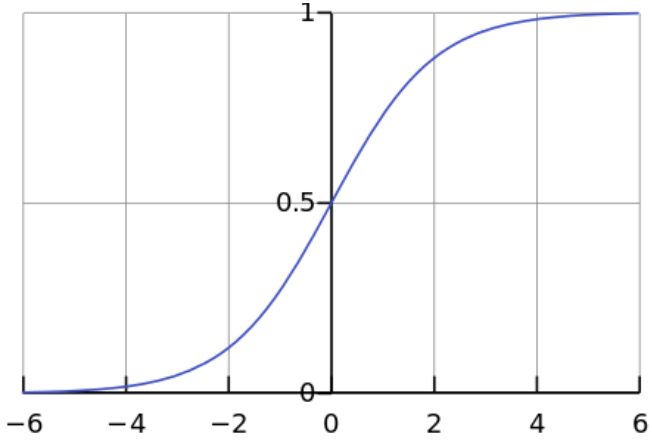


$$z_1^{(2)} = \Theta_{10}^{(1)} x_0 + \Theta_{11}^{(1)} x_1 + \Theta_{12}^{(1)} x_2$$

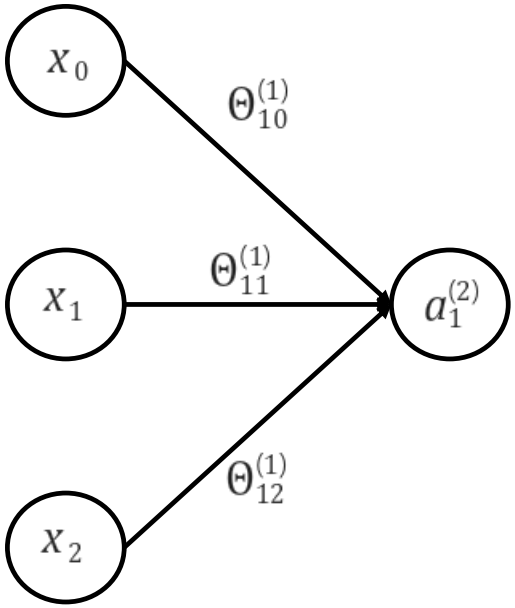
$$a_1^{(2)} = g(z_1^{(2)})$$

$$g(x) = \frac{1}{1 + e^{-x}}$$

logistic sigmoid



Forward Propagation



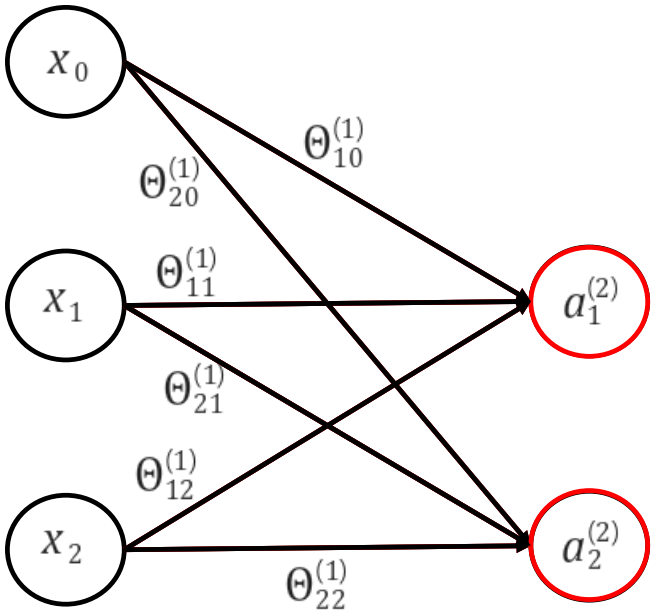
$$z_1^{(2)} = \Theta_{10}^{(1)} x_0 + \Theta_{11}^{(1)} x_1 + \Theta_{12}^{(1)} x_2$$

$$= \begin{pmatrix} \Theta_{10}^{(1)} & \Theta_{11}^{(1)} & \Theta_{12}^{(1)} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix}$$

$$a_1^{(2)} = g(z_1^{(2)})$$

$$g(x) = \frac{1}{1 + e^{-x}}$$

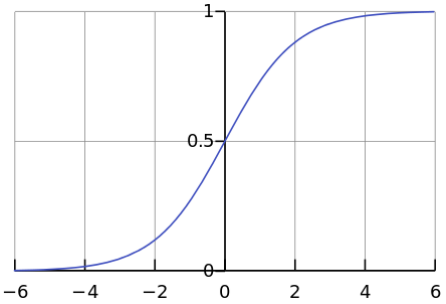
Forward Propagation



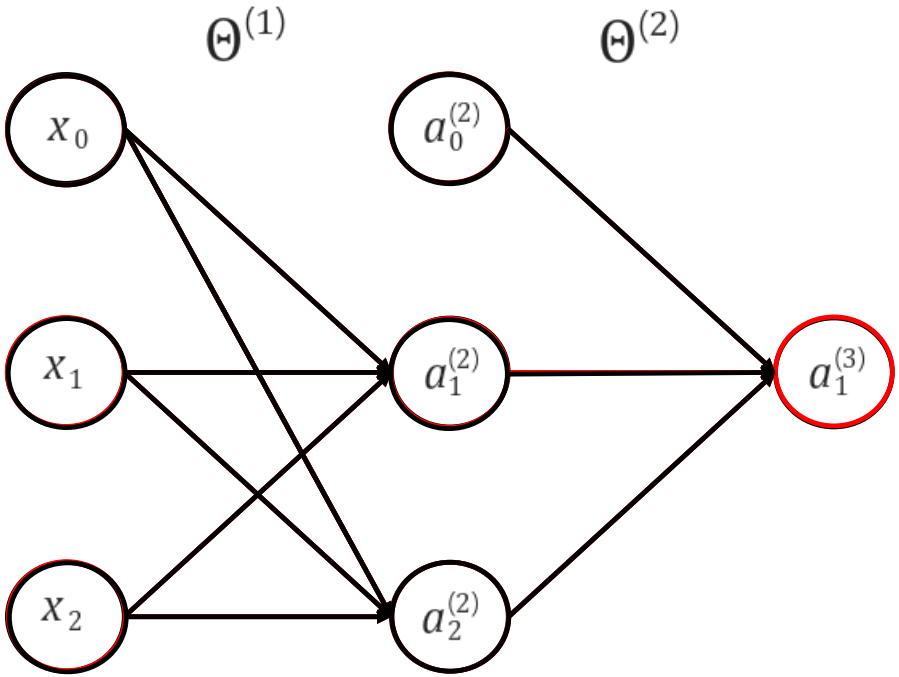
$$\begin{pmatrix} z_1^{(2)} \\ z_2^{(2)} \end{pmatrix} = \begin{pmatrix} \Theta_{10}^{(1)} & \Theta_{11}^{(1)} & \Theta_{12}^{(1)} \\ \Theta_{20}^{(1)} & \Theta_{21}^{(1)} & \Theta_{22}^{(1)} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix}$$

$$\begin{pmatrix} a_1^{(2)} \\ a_2^{(2)} \end{pmatrix} = \begin{pmatrix} g(z_1^{(2)}) \\ g(z_2^{(2)}) \end{pmatrix}$$

logistic sigmoid



Forward Propagation



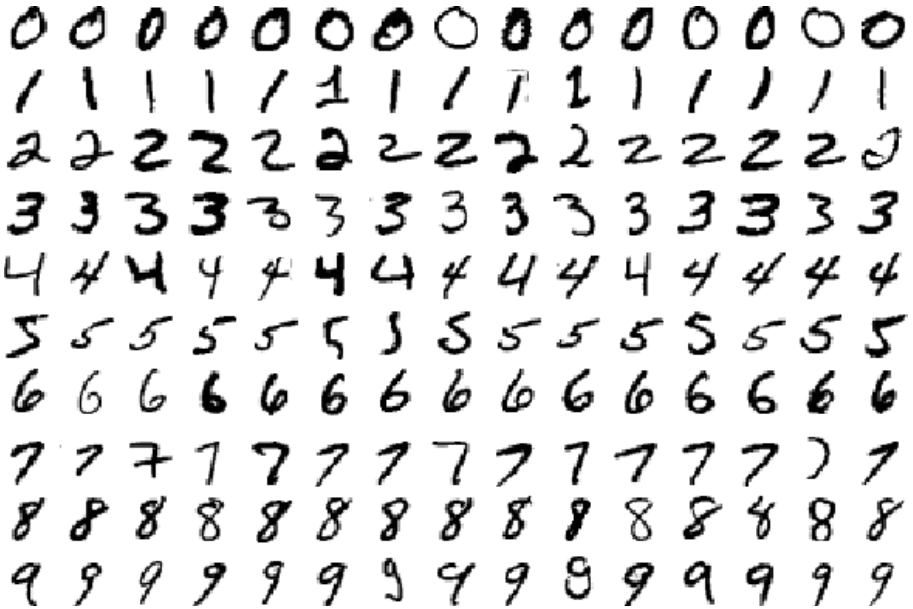
$$\begin{pmatrix} z_1^{(2)} \\ z_2^{(2)} \end{pmatrix} = \begin{pmatrix} \Theta_{10}^{(1)} & \Theta_{11}^{(1)} & \Theta_{12}^{(1)} \\ \Theta_{20}^{(1)} & \Theta_{21}^{(1)} & \Theta_{22}^{(1)} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix}$$

$$\begin{pmatrix} a_1^{(2)} \\ a_2^{(2)} \end{pmatrix} = \begin{pmatrix} g(z_1^{(2)}) \\ g(z_2^{(2)}) \end{pmatrix}$$

$$z_1^{(3)} = \begin{pmatrix} \Theta_{20}^{(1)} & \Theta_{21}^{(1)} & \Theta_{22}^{(1)} \end{pmatrix} \begin{pmatrix} a_0^{(2)} \\ a_1^{(2)} \\ a_2^{(2)} \end{pmatrix}$$

$$a_1^{(3)} = g(z_1^{(3)})$$

MNIST Example

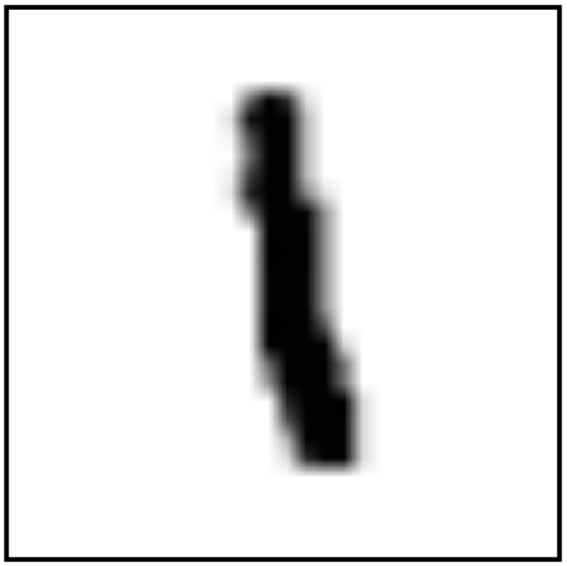


MNIST dataset of handwritten digits for 0 to 9.

It has 55,000 examples for training and 10,000 examples for testing.

The digits have been size-normalized and centered in a fixed-size image, so each sample is represented as a 2D matrix of size 28x28 with values from 0 to 1.

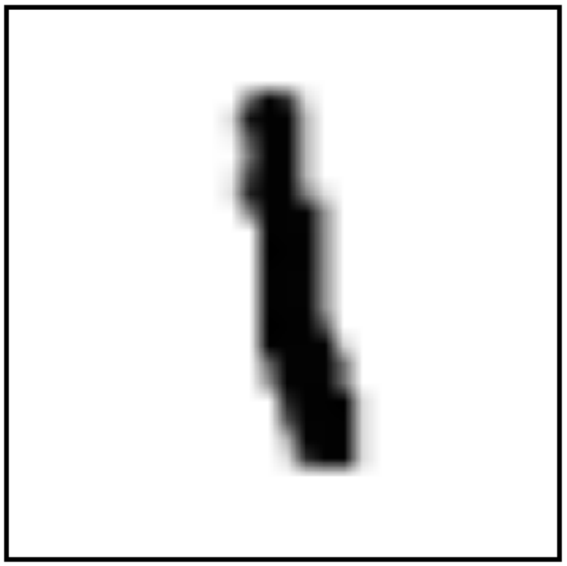
MNIST Example



$\simeq$

$$\begin{bmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & .6 & .8 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & .7 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & .7 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & .5 & 1 & .4 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & .4 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & .4 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & .7 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & .9 & 1 & .1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & .3 & 1 & .1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}$$

MNIST Example



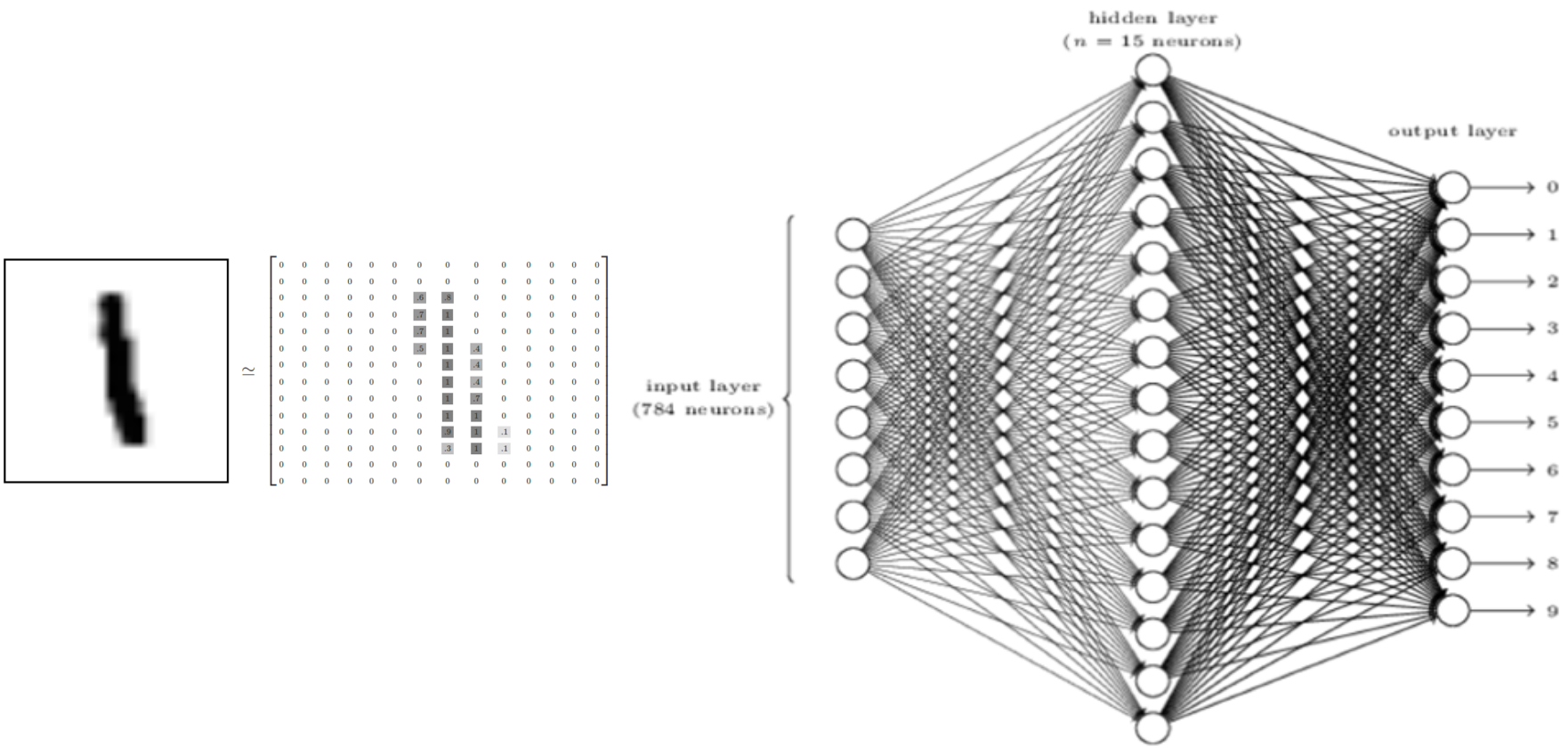
$\simeq$

28

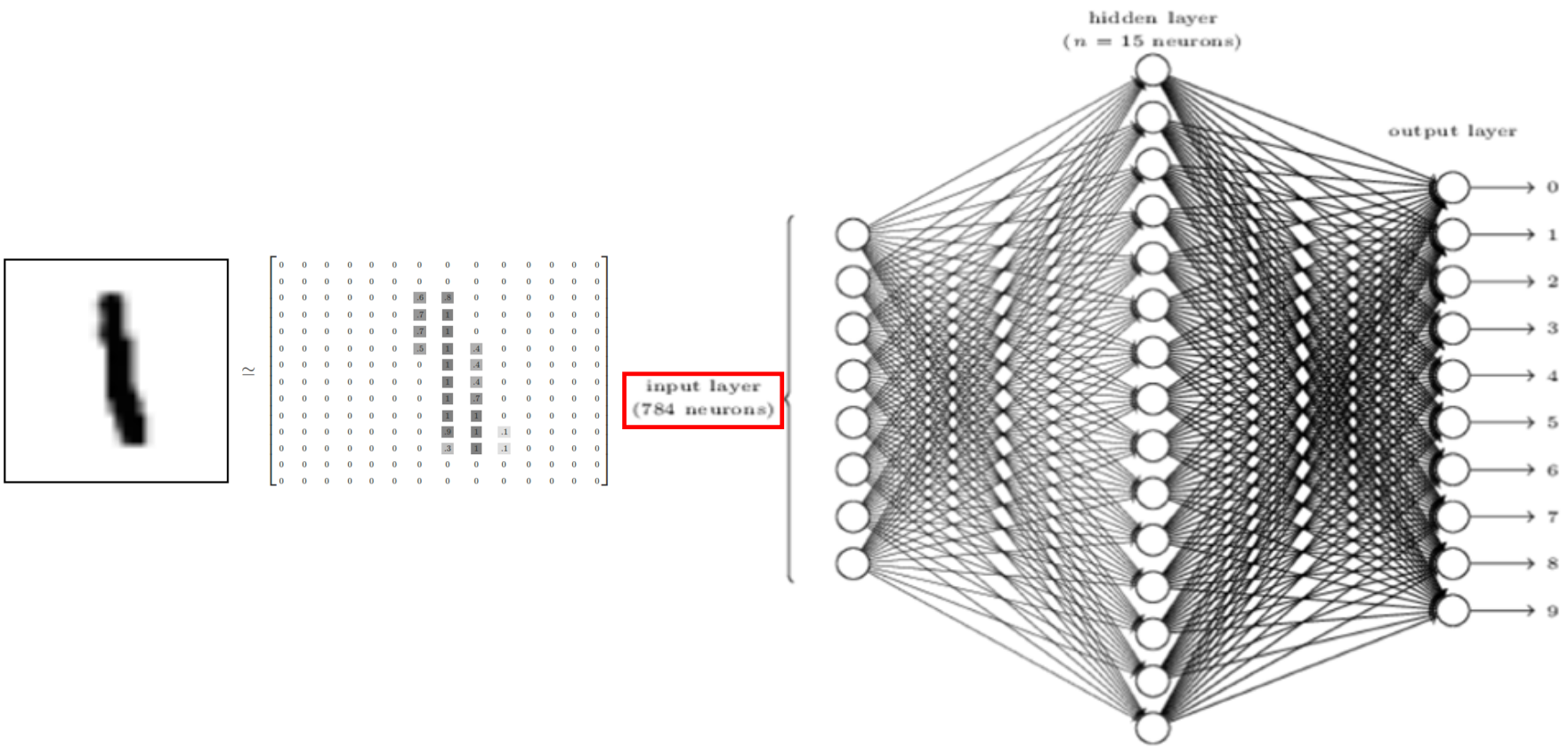
$$\begin{bmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & .6 & .8 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & .7 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & .7 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & .5 & 1 & .4 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & .4 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & .4 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & .7 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & .9 & 1 & .1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & .3 & 1 & .1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}$$

28

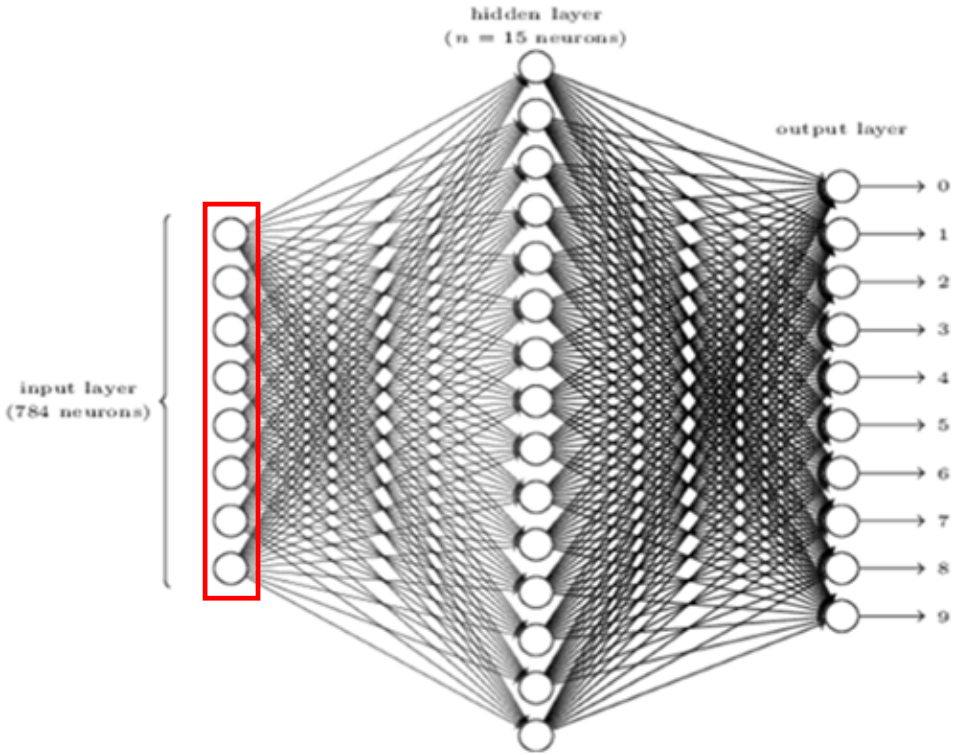
MNIST Example



MNIST Example

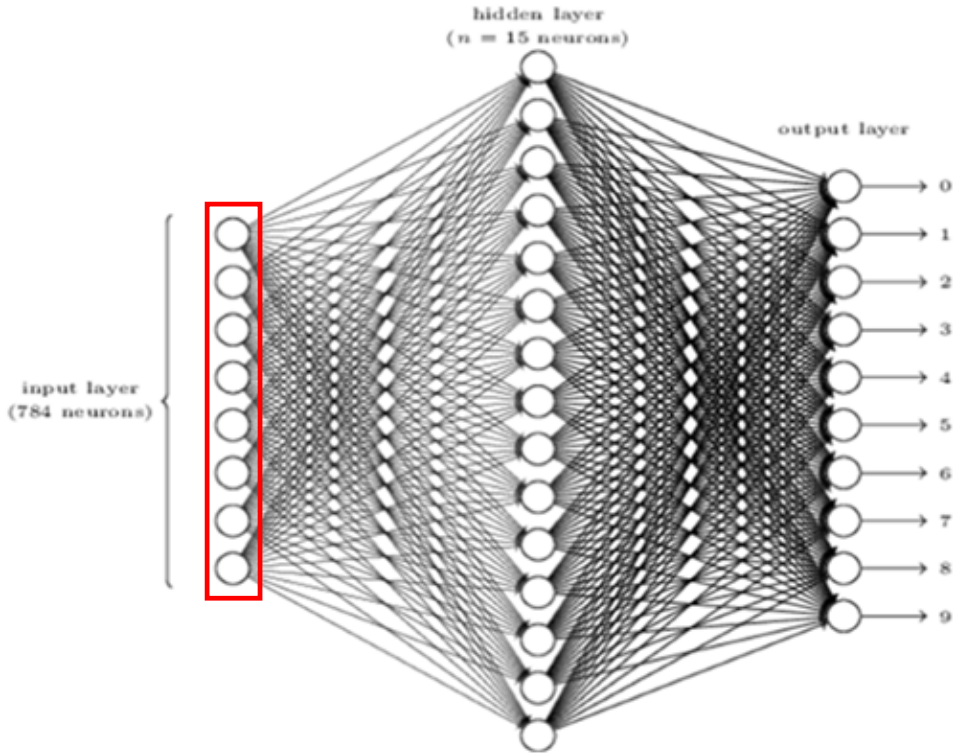


MNIST Example



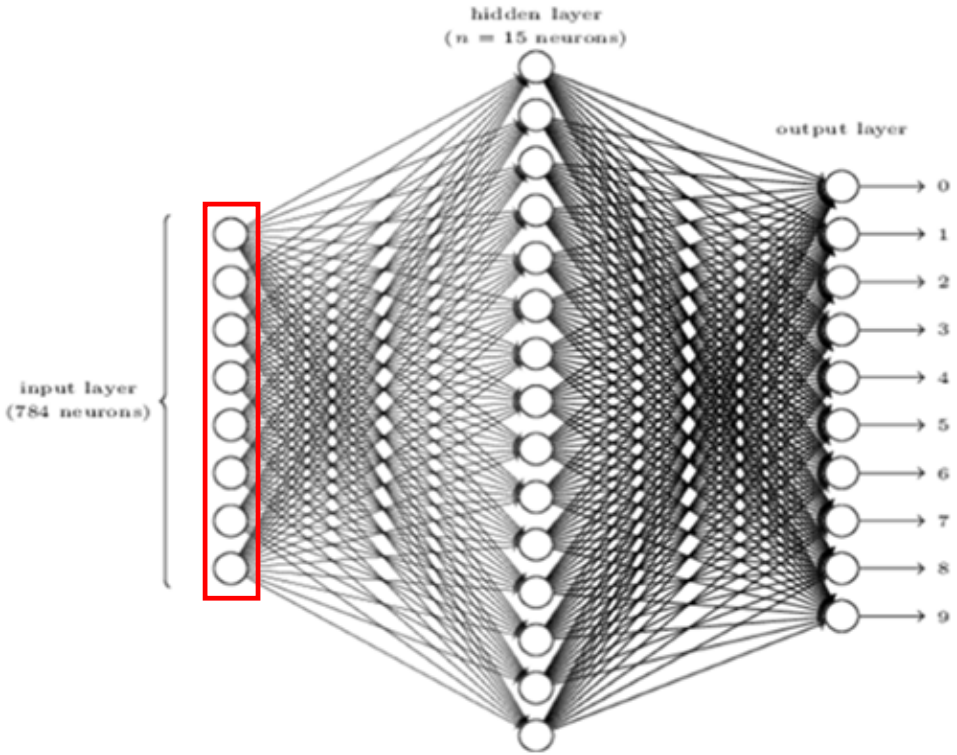

```
x: 784 x 1
w1: 15 x (784 + 1)
w2: 10 x (15 + 1)


import numpy as np

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def neural_network(x, w1, w2):
    a1 = np.concatenate([x, [[1]]], 0)
    z2 = np.dot(w1, a1)
    a2 = sigmoid(z2)
    a2 = np.concatenate([a2, [[1]]], 0)
    z3 = np.dot(w2, a2)
    a3 = sigmoid(z3)
    return a3
```

MNIST Example



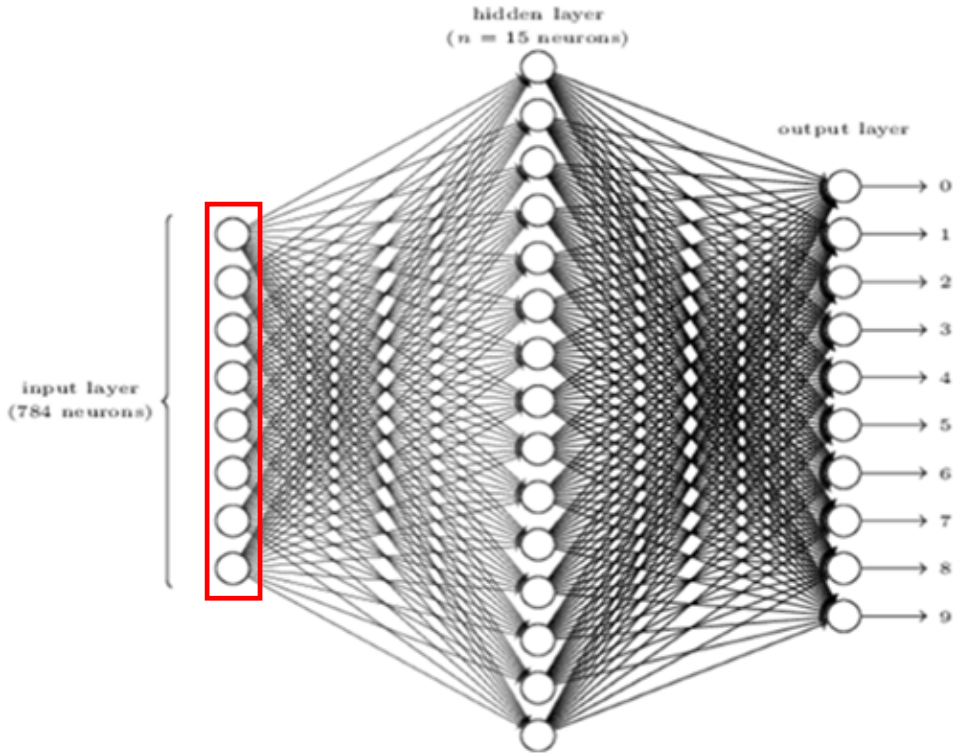

```
x: 784 x 1
w1: 15 x (784 + 1)
w2: 10 x (15 + 1)


import numpy as np

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def neural_network(x, w1, w2):
    a1 = np.concatenate([x, [[1]]], 0)
    z2 = np.dot(w1, a1)
    a2 = sigmoid(z2)
    a2 = np.concatenate([a2, [[1]]], 0)
    z3 = np.dot(w2, a2)
    a3 = sigmoid(z3)
    return a3
```

MNIST Example




```
x: 784 x 1
w1: 15 x (784 + 1)
w2: 10 x (15 + 1)


import numpy as np

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def neural_network(x, w1, w2):
    a1 = np.concatenate([x, [[1]]], 0)
    z2 = np.dot(w1, a1)
    a2 = sigmoid(z2)
    a2 = np.concatenate([a2, [[1]]], 0)
    z3 = np.dot(w2, a2)
    a3 = sigmoid(z3)
    return a3
```

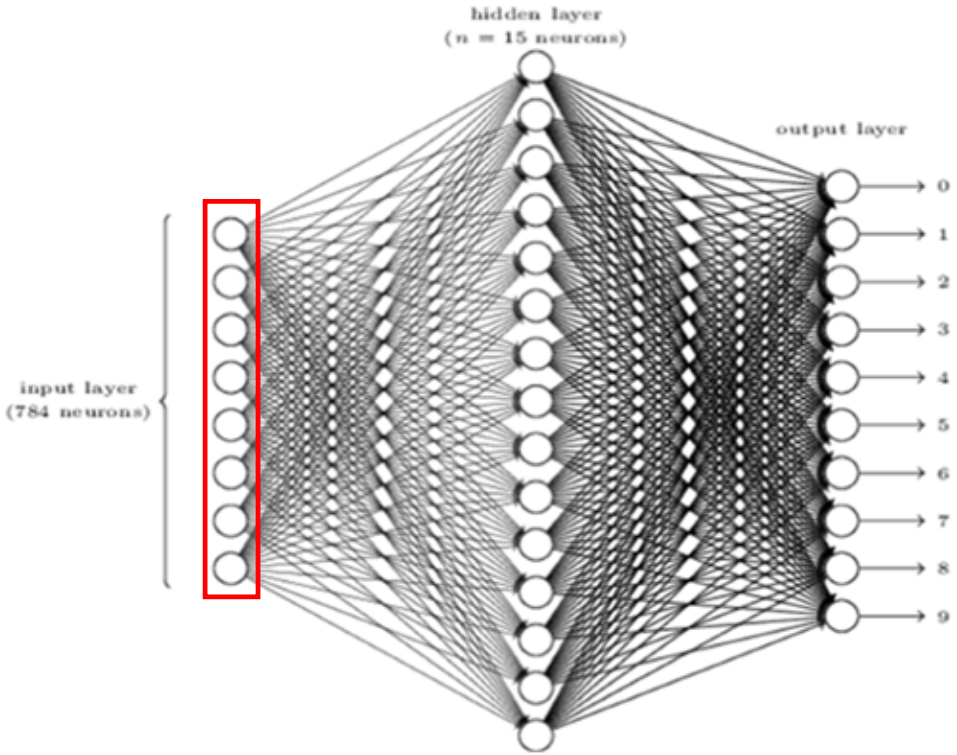
MNIST Example




```
x: 784 x 1
w1: 15 x (784 + 1)
w2: 10 x (15 + 1)


import numpy as np

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def neural_network(x, w1, w2):
    a1 = np.concatenate([x, [[1]]], 0)
    z2 = np.dot(w1, a1)
    a2 = sigmoid(z2)
    a2 = np.concatenate([a2, [[1]]], 0)
    z3 = np.dot(w2, a2)
    a3 = sigmoid(z3)
    return a3
```

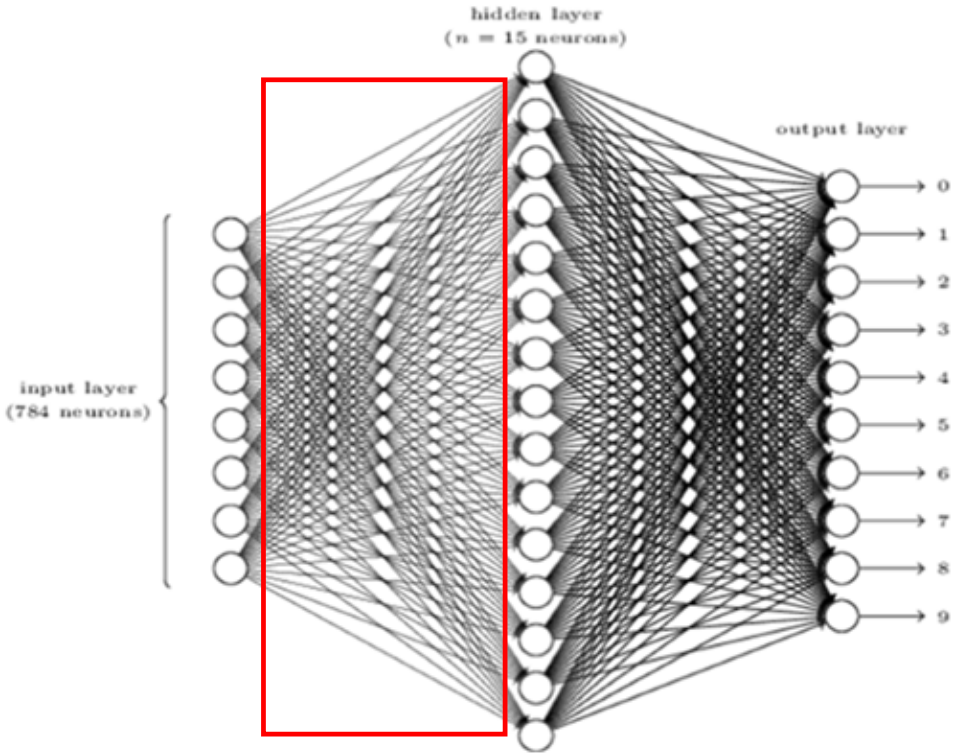
785 x 1

MNIST Example



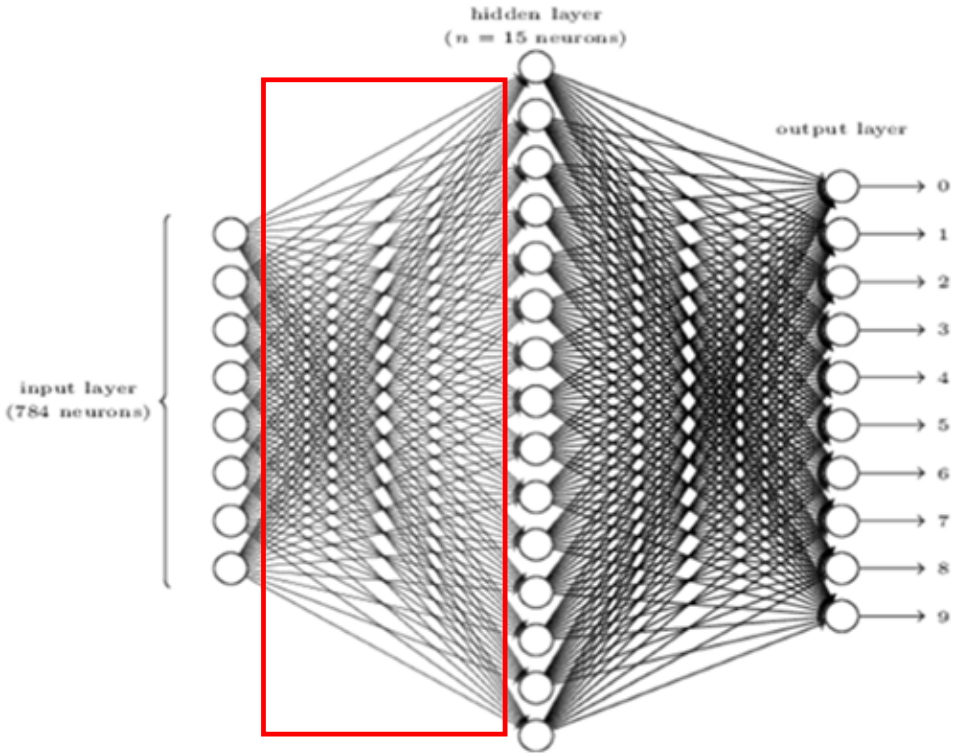

```
x: 784 x 1                    a1: 785 x 1
w1: 15 x (784 + 1)
w2: 10 x (15 + 1)


import numpy as np

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def neural_network(x, w1, w2):
    a1 = np.concatenate([x, [[1]]], 0)
    z2 = np.dot(w1, a1)
    a2 = sigmoid(z2)
    a2 = np.concatenate([a2, [[1]]], 0)
    z3 = np.dot(w2, a2)
    a3 = sigmoid(z3)
    return a3
```

MNIST Example



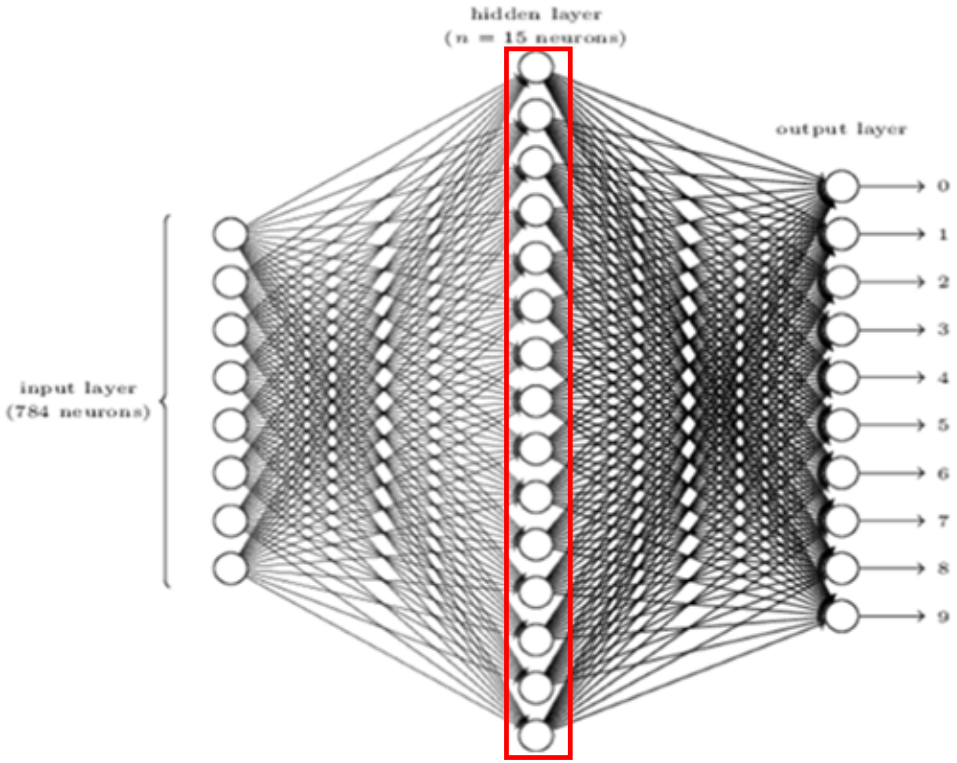

```
x: 784 x 1                    a1: 785 x 1
w1: 15 x (784 + 1)
w2: 10 x (15 + 1)


import numpy as np

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def neural_network(x, w1, w2):
    a1 = np.concatenate([x, [[1]]], 0)
    z2 = np.dot(w1, a1)
    a2 = sigmoid(z2)
    a2 = np.concatenate([a2, [[1]]], 0)
    z3 = np.dot(w2, a2)
    a3 = sigmoid(z3)
    return a3
```

MNIST Example



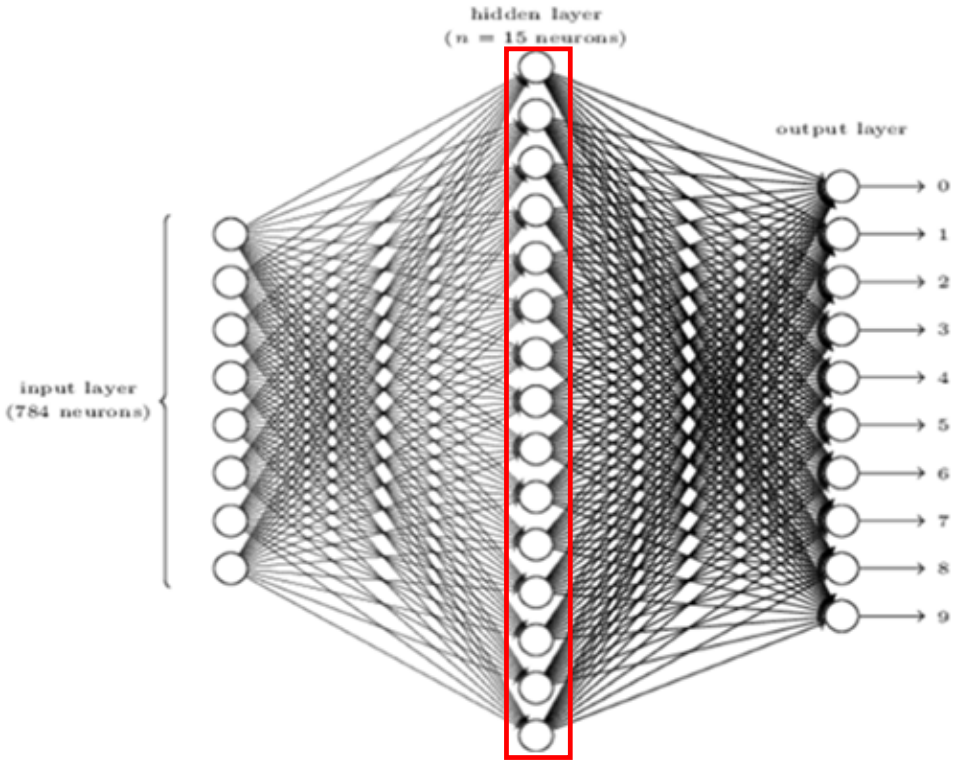

```
x: 784 x 1                    a1: 785 x 1
w1: 15 x (784 + 1)
w2: 10 x (15 + 1)


import numpy as np

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def neural_network(x, w1, w2):
    a1 = np.concatenate([x, [[1]]], 0)
    z2 = np.dot(w1, a1)
    a2 = sigmoid(z2)
    a2 = np.concatenate([a2, [[1]]], 0)
    z3 = np.dot(w2, a2)
    a3 = sigmoid(z3)
    return a3
```

MNIST Example



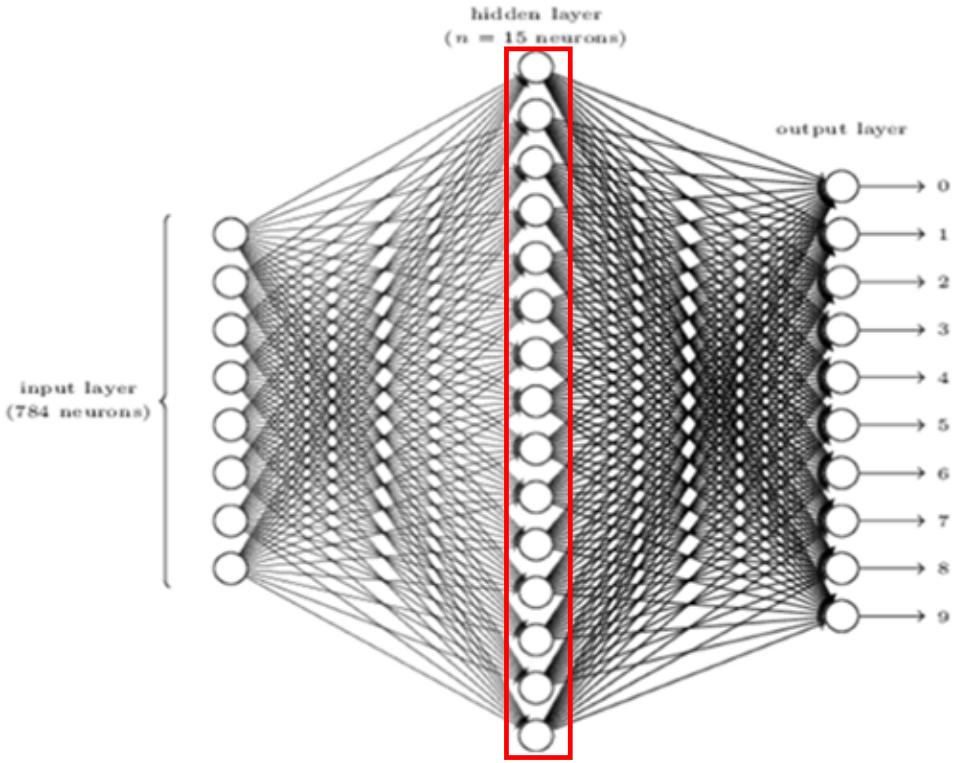

```
x: 784 x 1                    a1: 785 x 1
w1: 15 x (784 + 1)
w2: 10 x (15 + 1)


import numpy as np

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def neural_network(x, w1, w2):
    a1 = np.concatenate([x, [[1]]], 0)
    z2 = np.dot(w1, a1)
    a2 = sigmoid(z2)
    a2 = np.concatenate([a2, [[1]]], 0)
    z3 = np.dot(w2, a2)
    a3 = sigmoid(z3)
    return a3
```

MNIST Example




```
x: 784 x 1                    a1: 785 x 1
w1: 15 x (784 + 1)
w2: 10 x (15 + 1)


import numpy as np

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def neural_network(x, w1, w2):
    a1 = np.concatenate([x, [[1]]], 0)
    z2 = np.dot(w1, a1)
    a2 = sigmoid(z2)
    a2 = np.concatenate([a2, [[1]]], 0)
    z3 = np.dot(w2, a2)
    a3 = sigmoid(z3)
    return a3
```

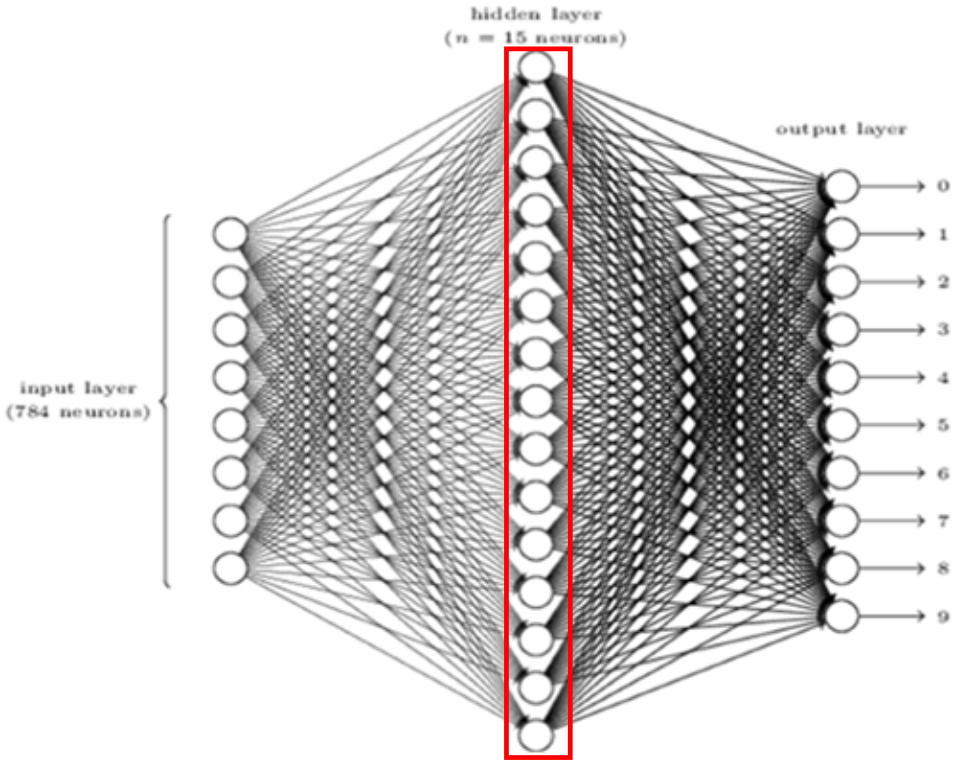
MNIST Example




```
x: 784 x 1                    a1: 785 x 1
w1: 15 x (784 + 1)
w2: 10 x (15 + 1)


import numpy as np

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def neural_network(x, w1, w2):
    a1 = np.concatenate([x, [[1]]], 0)
    z2 = np.dot(w1, a1)
    a2 = sigmoid(z2)
    a2 = np.concatenate([a2, [[1]]], 0)
    z3 = np.dot(w2, a2)
    a3 = sigmoid(z3)
    return a3
```

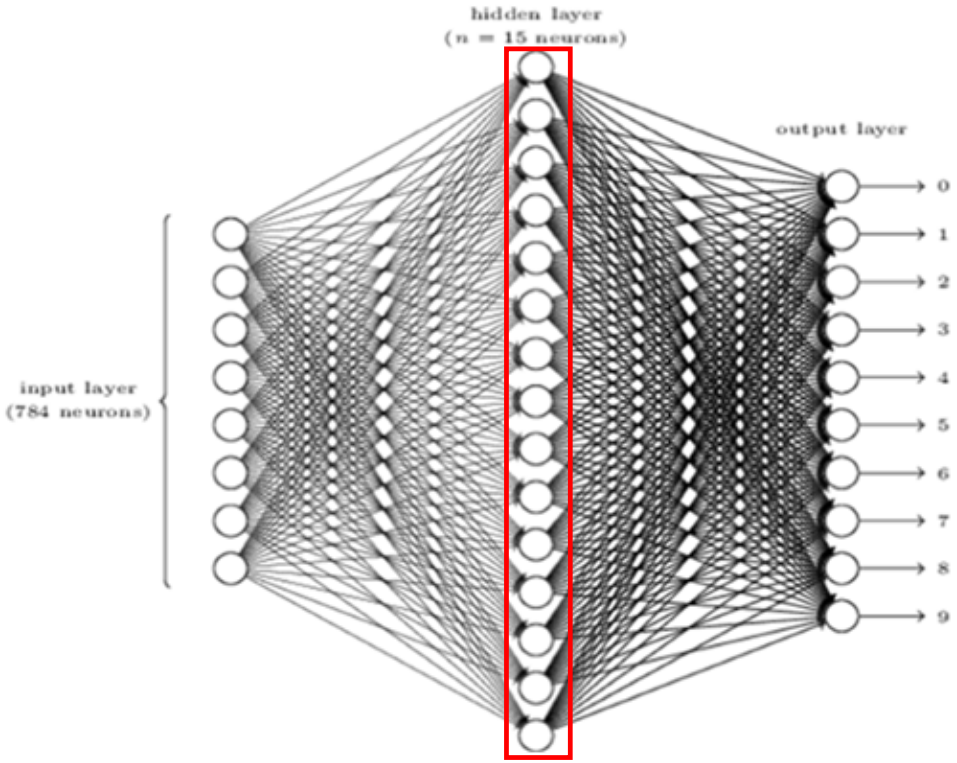
15 x 1

MNIST Example




```
x: 784 x 1                    a1: 785 x 1
w1: 15 x (784 + 1)
w2: 10 x (15 + 1)


import numpy as np

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def neural_network(x, w1, w2):
    a1 = np.concatenate([x, [[1]]], 0)
    z2 = np.dot(w1, a1)
    a2 = sigmoid(z2)
    a2 = np.concatenate([a2, [[1]]], 0)
    z3 = np.dot(w2, a2)
    a3 = sigmoid(z3)
    return a3
```

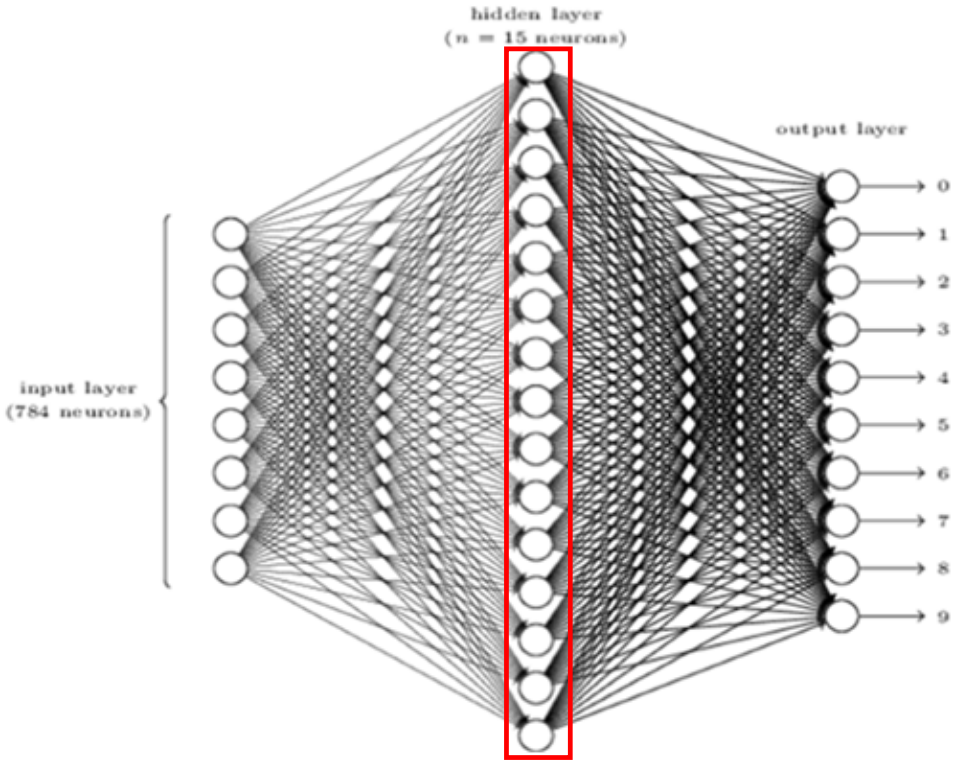
MNIST Example




```
x: 784 x 1                    a1: 785 x 1
w1: 15 x (784 + 1)
w2: 10 x (15 + 1)


import numpy as np

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def neural_network(x, w1, w2):
    a1 = np.concatenate([x, [[1]]], 0)
    z2 = np.dot(w1, a1)
    a2 = sigmoid(z2)
    a2 = np.concatenate([a2, [[1]]], 0)
    z3 = np.dot(w2, a2)
    a3 = sigmoid(z3)
    return a3
```

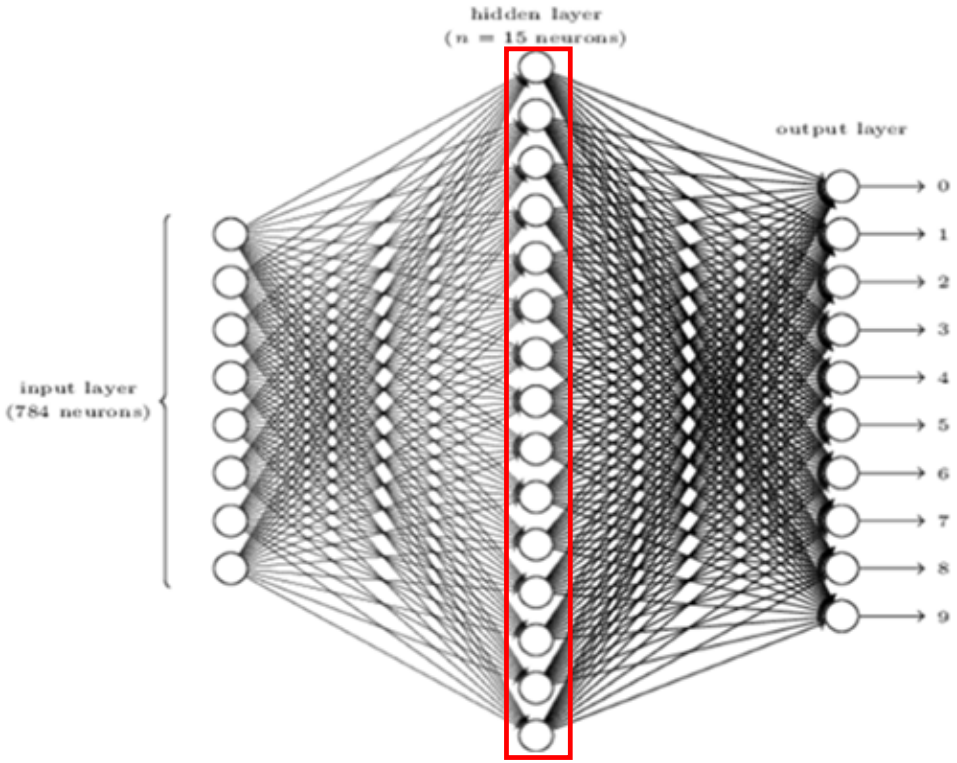
15 x 1

MNIST Example



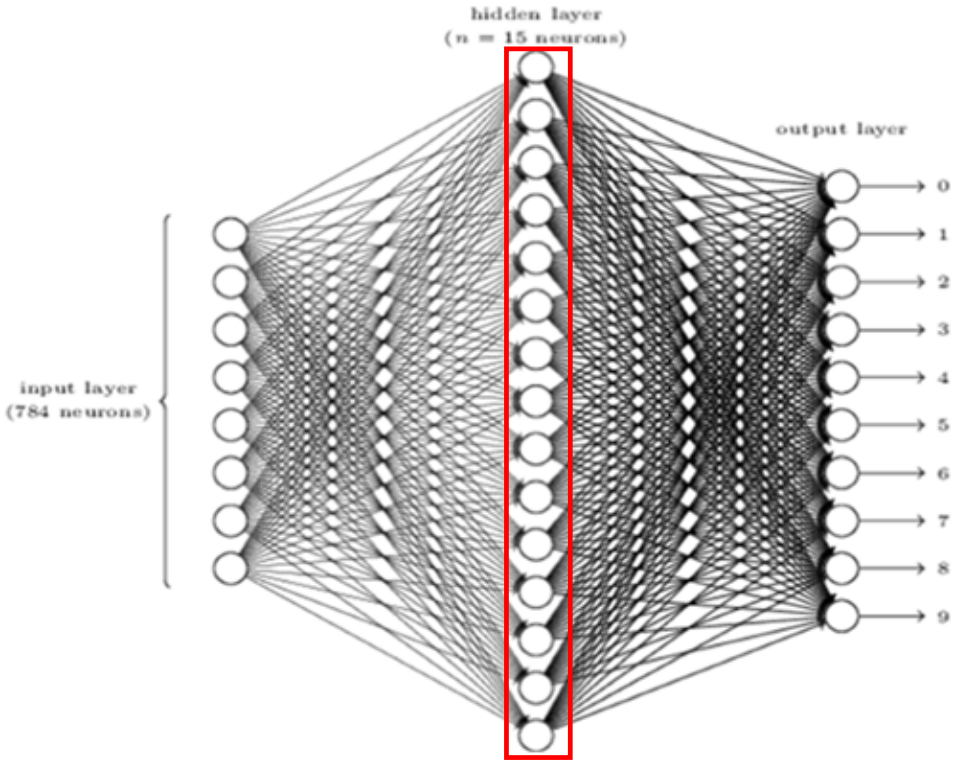

```
x: 784 x 1                    a1: 785 x 1
w1: 15 x (784 + 1)
w2: 10 x (15 + 1)


import numpy as np

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def neural_network(x, w1, w2):
    a1 = np.concatenate([x, [[1]]], 0)
    z2 = np.dot(w1, a1)
    a2 = sigmoid(z2)
    a2 = np.concatenate([a2, [[1]]], 0)
    z3 = np.dot(w2, a2)
    a3 = sigmoid(z3)
    return a3
```

MNIST Example




```
x: 784 x 1                    a1: 785 x 1
w1: 15 x (784 + 1)
w2: 10 x (15 + 1)


import numpy as np

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def neural_network(x, w1, w2):
    a1 = np.concatenate([x, [[1]]], 0)
    z2 = np.dot(w1, a1)
    a2 = sigmoid(z2)                          16 x 1
    a2 = np.concatenate([a2, [[1]]], 0)
    z3 = np.dot(w2, a2)
    a3 = sigmoid(z3)
    return a3
```

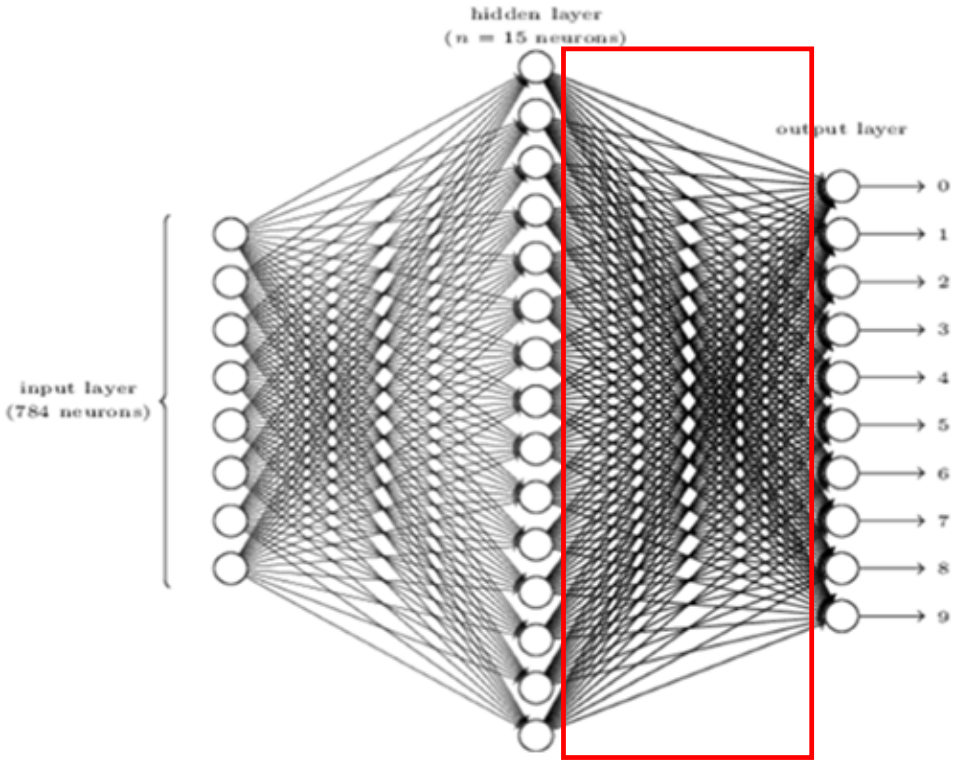
MNIST Example




```
x: 784 x 1                  a1: 785 x 1
w1: 15 x (784 + 1)          a2: 16 x 1
w2: 10 x (15 + 1)
```

```python
import numpy as np

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def neural_network(x, w1, w2):
    a1 = np.concatenate([x, [[1]]], 0)
    z2 = np.dot(w1, a1)
    a2 = sigmoid(z2)
    a2 = np.concatenate([a2, [[1]]], 0)
    z3 = np.dot(w2, a2)
    a3 = sigmoid(z3)
    return a3
```

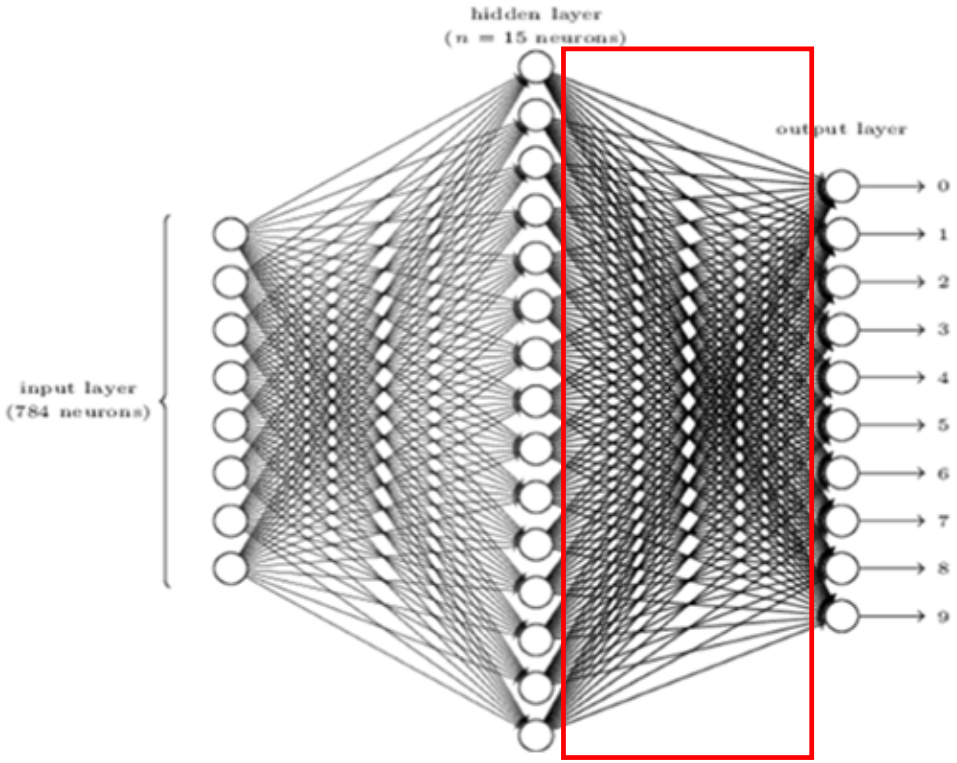
MNIST Example




```
x: 784 x 1                  a1: 785 x 1
w1: 15 x (784 + 1)          a2: 16 x 1
w2: 10 x (15 + 1)
```

```python
import numpy as np

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def neural_network(x, w1, w2):
    a1 = np.concatenate([x, [[1]]], 0)
    z2 = np.dot(w1, a1)
    a2 = sigmoid(z2)
    a2 = np.concatenate([a2, [[1]]], 0)
    z3 = np.dot(w2, a2)
    a3 = sigmoid(z3)
    return a3
```

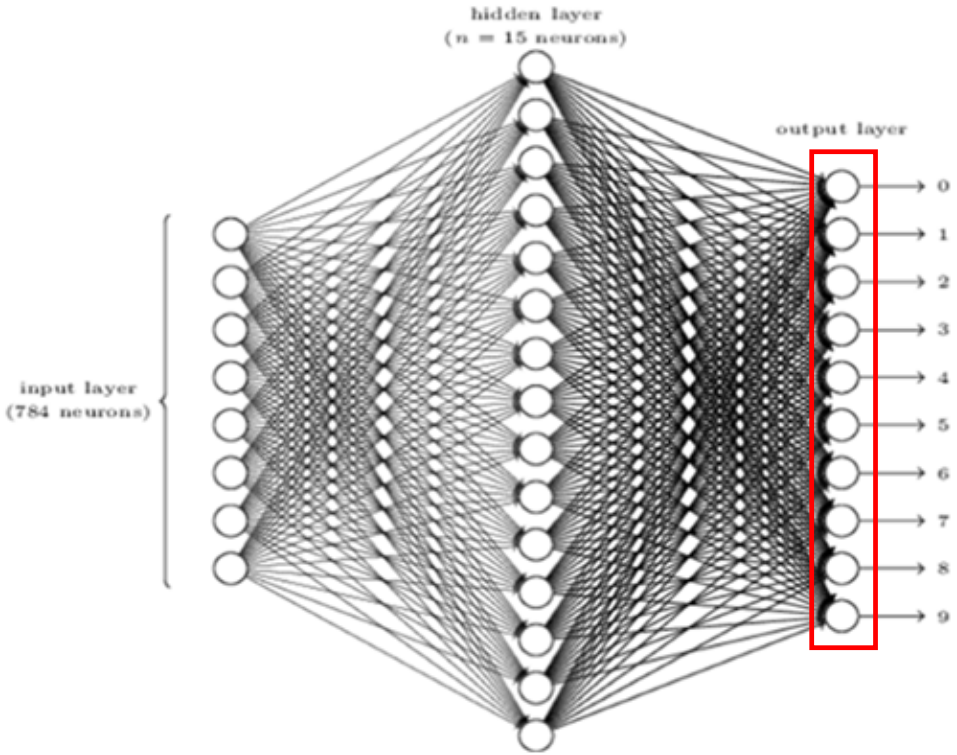
MNIST Example




```
x: 784 x 1                  a1: 785 x 1
w1: 15 x (784 + 1)          a2: 16 x 1
w2: 10 x (15 + 1)
```

```python
import numpy as np

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def neural_network(x, w1, w2):
    a1 = np.concatenate([x, [[1]]], 0)
    z2 = np.dot(w1, a1)
    a2 = sigmoid(z2)
    a2 = np.concatenate([a2, [[1]]], 0)
    z3 = np.dot(w2, a2)
    a3 = sigmoid(z3)
    return a3
```

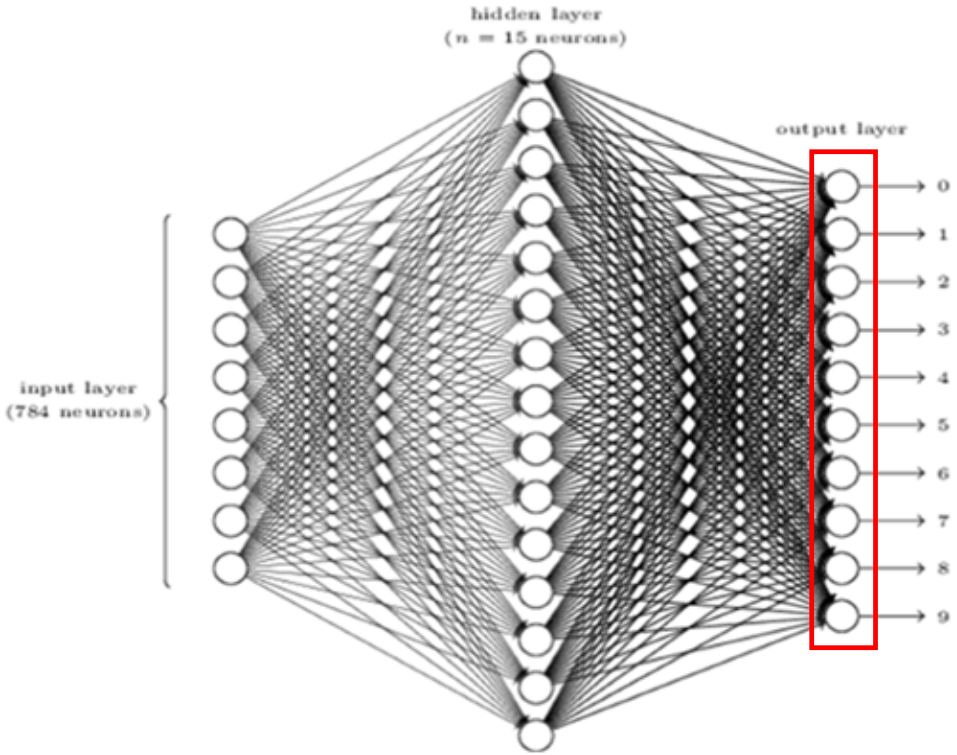
MNIST Example




```
x: 784 x 1                  a1: 785 x 1
w1: 15 x (784 + 1)          a2: 16 x 1
w2: 10 x (15 + 1)
```

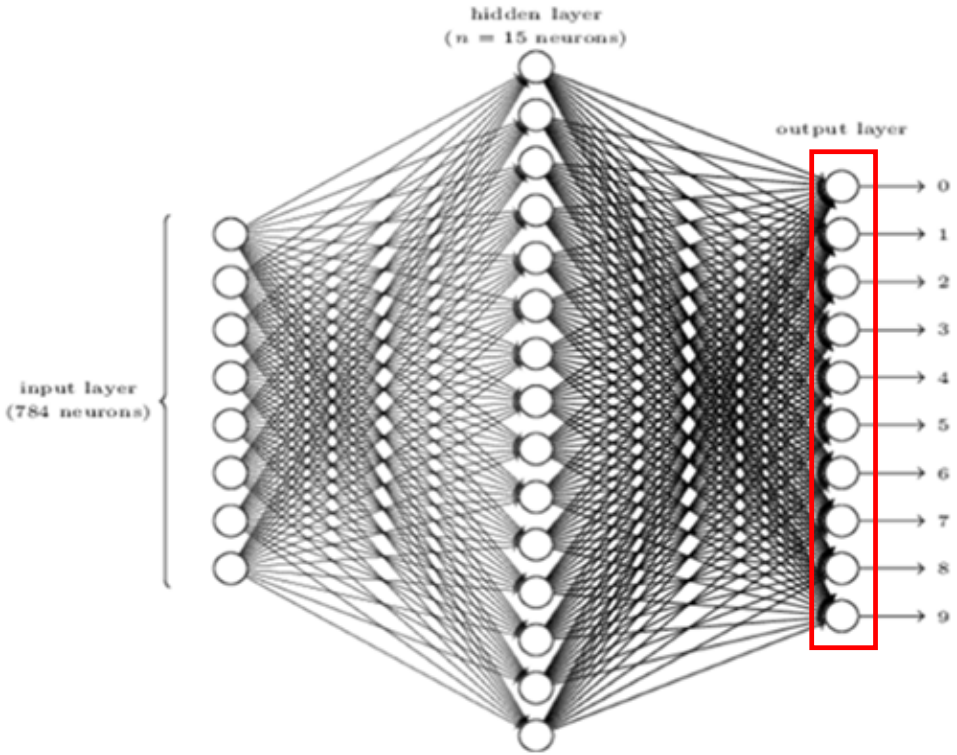
```python
import numpy as np

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def neural_network(x, w1, w2):
    a1 = np.concatenate([x, [[1]]], 0)
    z2 = np.dot(w1, a1)
    a2 = sigmoid(z2)
    a2 = np.concatenate([a2, [[1]]], 0)
    z3 = np.dot(w2, a2)
    a3 = sigmoid(z3)
    return a3
```

MNIST Example




```
x: 784 x 1                  a1: 785 x 1
w1: 15 x (784 + 1)          a2: 16 x 1
w2: 10 x (15 + 1)


import numpy as np

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def neural_network(x, w1, w2):
    a1 = np.concatenate([x, [[1]]], 0)
    z2 = np.dot(w1, a1)
    a2 = sigmoid(z2)
    a2 = np.concatenate([a2, [[1]]], 0)
    z3 = np.dot(w2, a2)
    a3 = sigmoid(z3)
    return a3
```

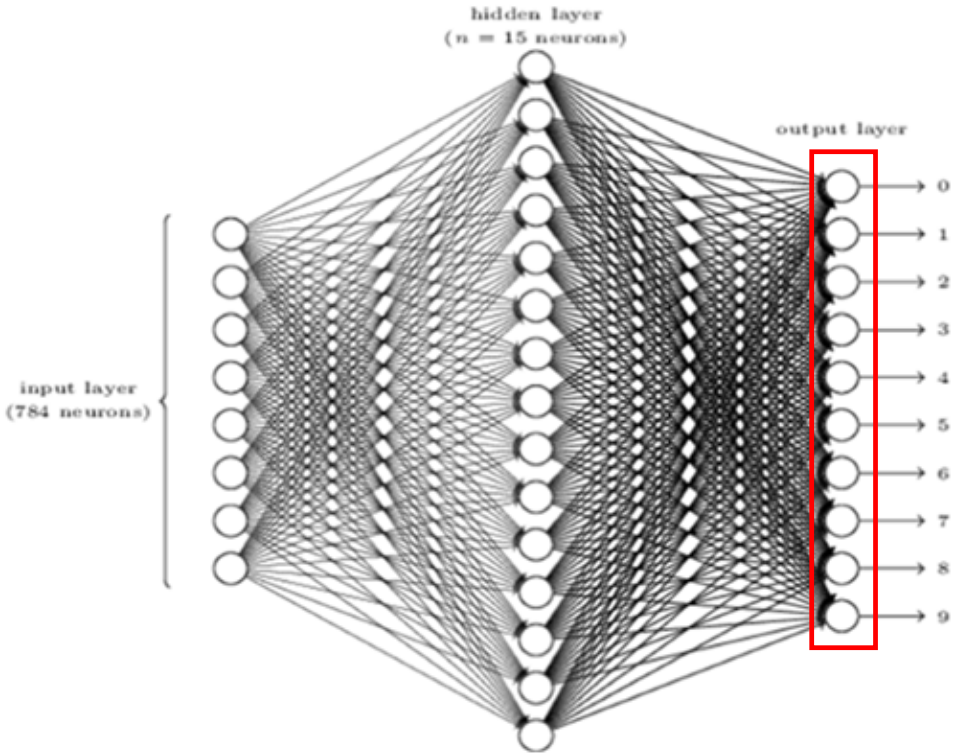
MNIST Example




```
x: 784 x 1                  a1: 785 x 1
w1: 15 x (784 + 1)          a2: 16 x 1
w2: 10 x (15 + 1)
```

```
import numpy as np

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def neural_network(x, w1, w2):
    a1 = np.concatenate([x, [[1]]], 0)
    z2 = np.dot(w1, a1)
    a2 = sigmoid(z2)
    a2 = np.concatenate([a2, [[1]]], 0)
    z3 = np.dot(w2, a2)
    a3 = sigmoid(z3)
    return a3
```

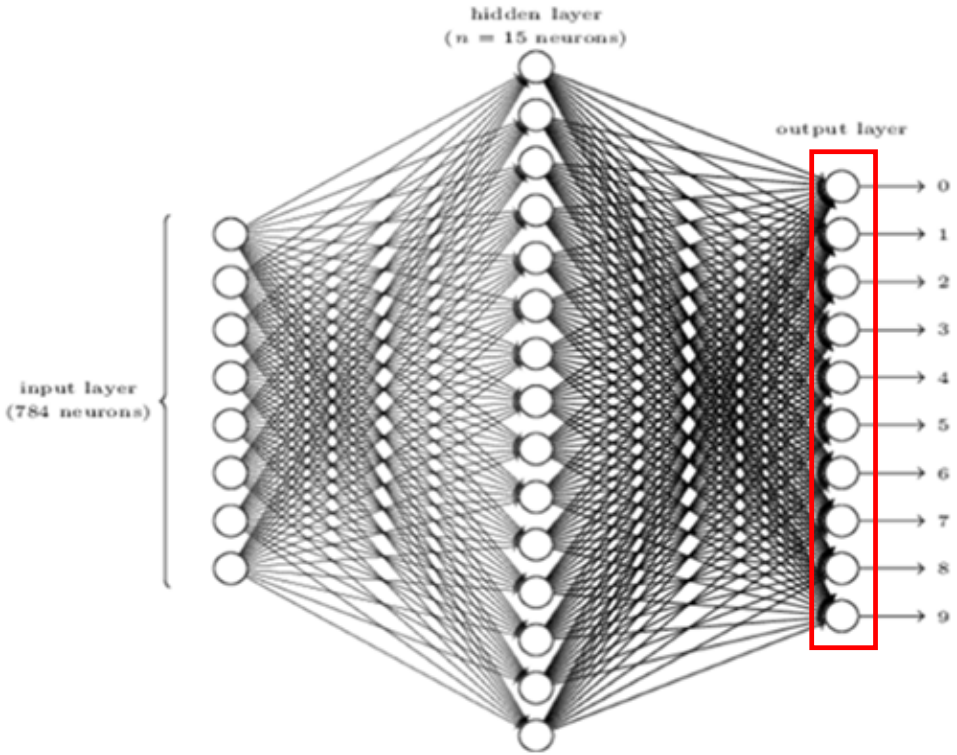
MNIST Example




```
x: 784 x 1                    a1: 785 x 1
w1: 15 x (784 + 1)            a2: 16 x 1
w2: 10 x (15 + 1)


import numpy as np

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def neural_network(x, w1, w2):
    a1 = np.concatenate([x, [[1]]], 0)
    z2 = np.dot(w1, a1)
    a2 = sigmoid(z2)
    a2 = np.concatenate([a2, [[1]]], 0)
    z3 = np.dot(w2, a2)
    a3 = sigmoid(z3)
    return a3
```

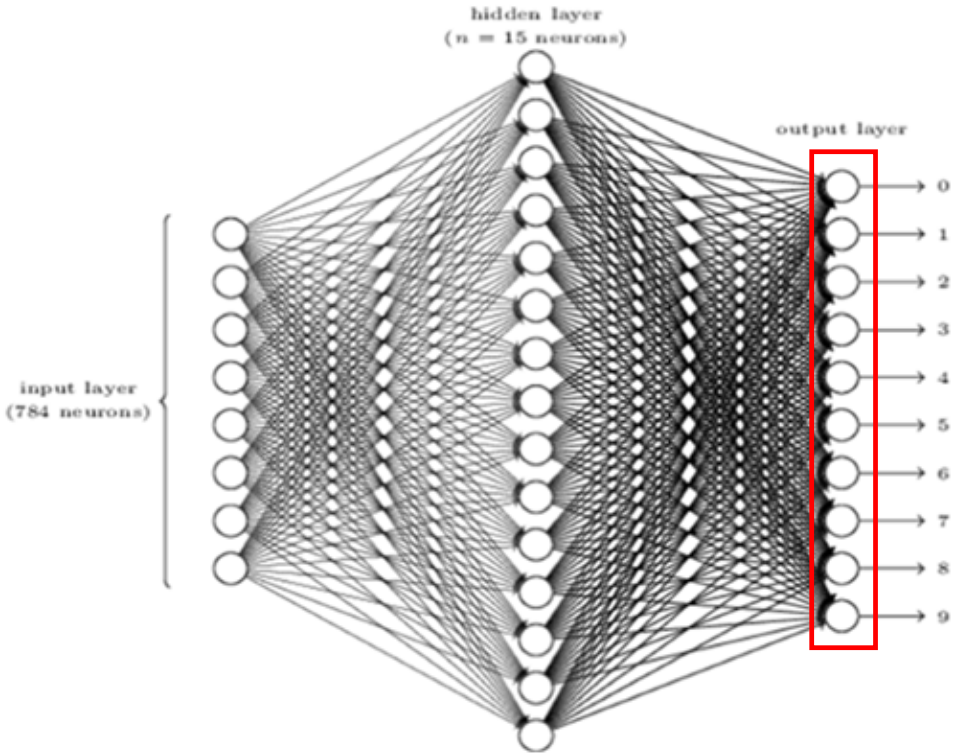
10 x 1

MNIST Example




```
x: 784 x 1                  a1: 785 x 1
w1: 15 x (784 + 1)          a2: 16 x 1
w2: 10 x (15 + 1)


import numpy as np

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def neural_network(x, w1, w2):
    a1 = np.concatenate([x, [[1]]], 0)
    z2 = np.dot(w1, a1)
    a2 = sigmoid(z2)
    a2 = np.concatenate([a2, [[1]]], 0)
    z3 = np.dot(w2, a2)
    a3 = sigmoid(z3)
    return a3
```

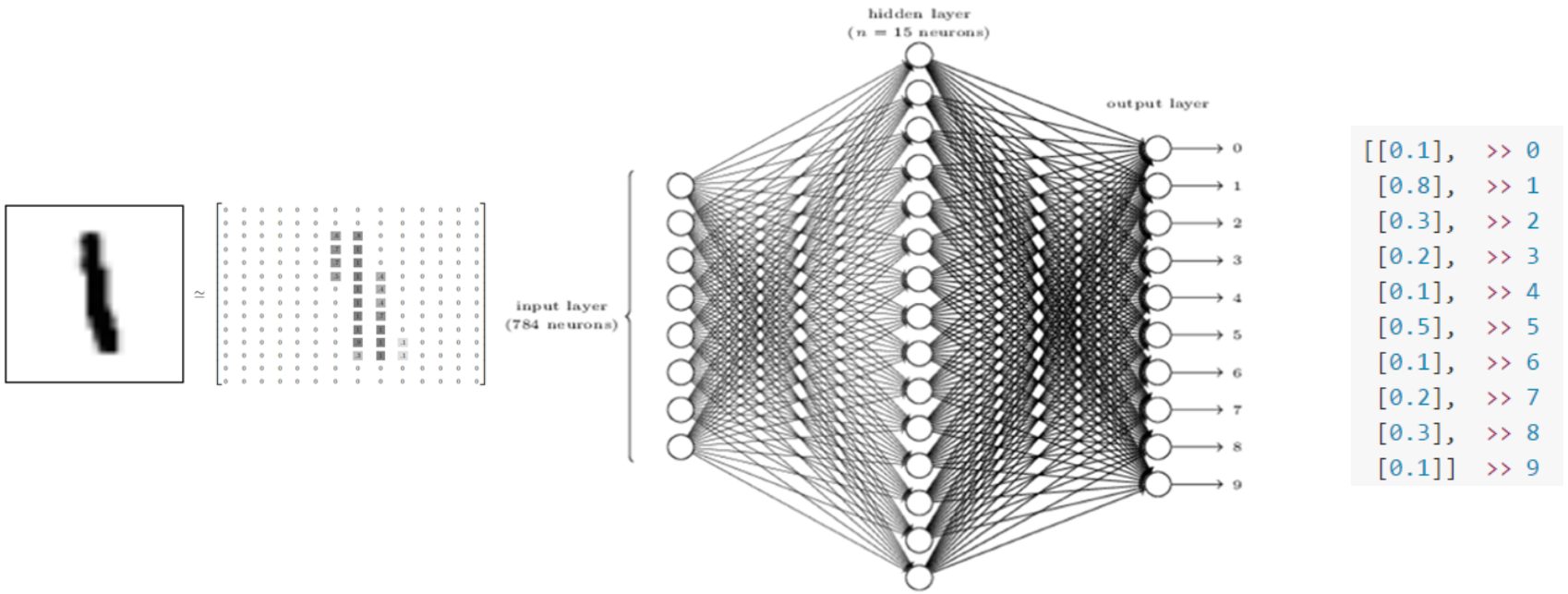
MNIST Example




```
x: 784 x 1                  a1: 785 x 1
w1: 15 x (784 + 1)          a2: 16 x 1
w2: 10 x (15 + 1)


import numpy as np

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def neural_network(x, w1, w2):
    a1 = np.concatenate([x, [[1]]], 0)
    z2 = np.dot(w1, a1)
    a2 = sigmoid(z2)
    a2 = np.concatenate([a2, [[1]]], 0)
    z3 = np.dot(w2, a2)
    a3 = sigmoid(z3)
    return a3
```

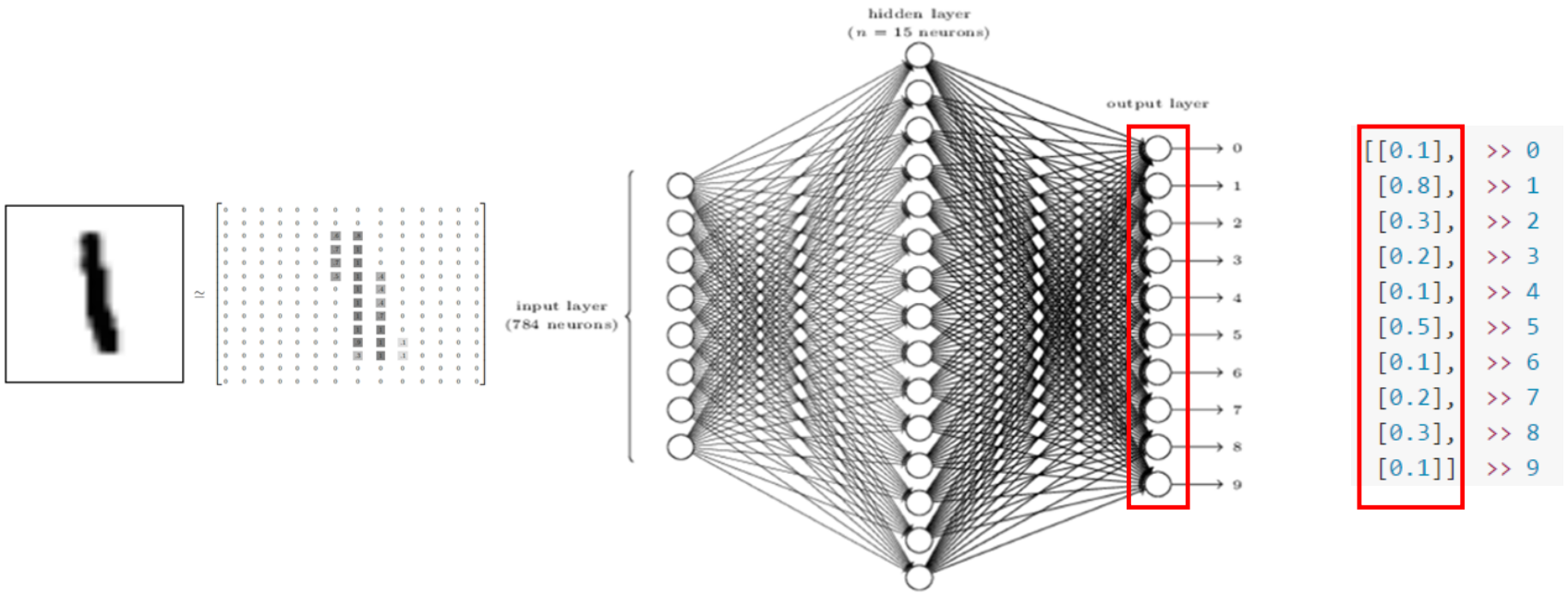
10 x 1

MNIST Example

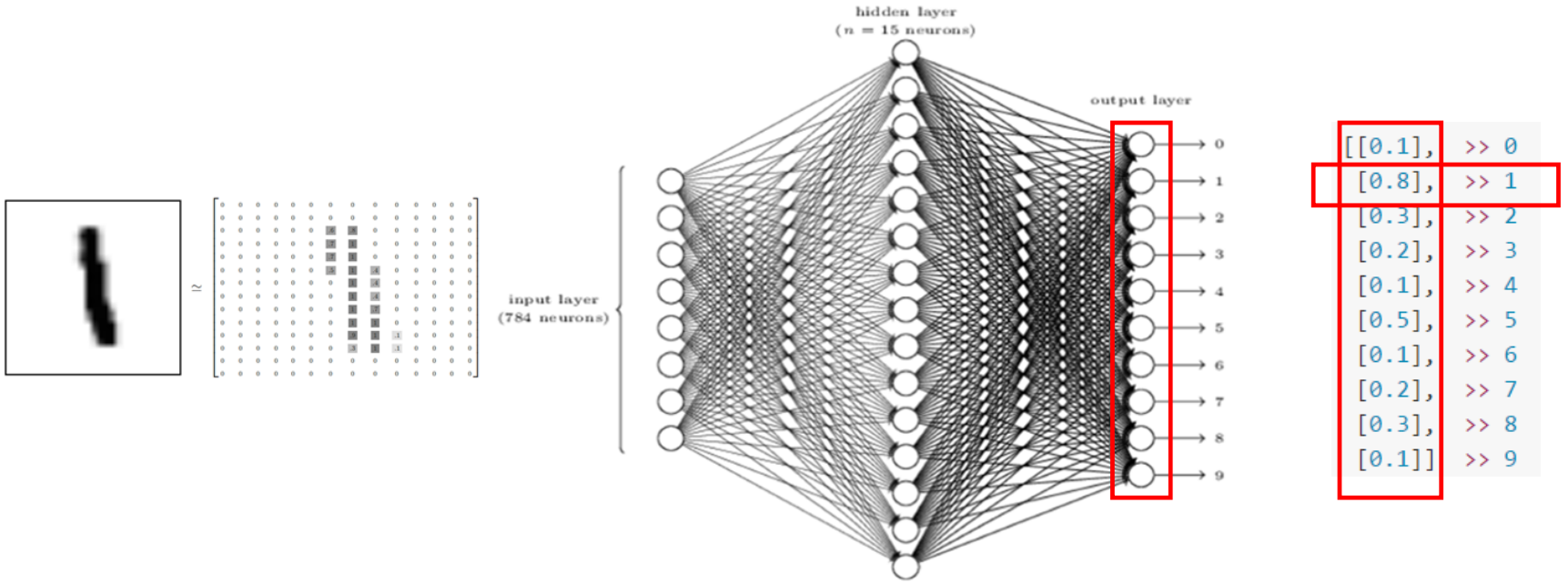



```
[[0.1],  >> 0
 [0.8],  >> 1
 [0.3],  >> 2
 [0.2],  >> 3
 [0.1],  >> 4
 [0.5],  >> 5
 [0.1],  >> 6
 [0.2],  >> 7
 [0.3],  >> 8
 [0.1]]  >> 9
```

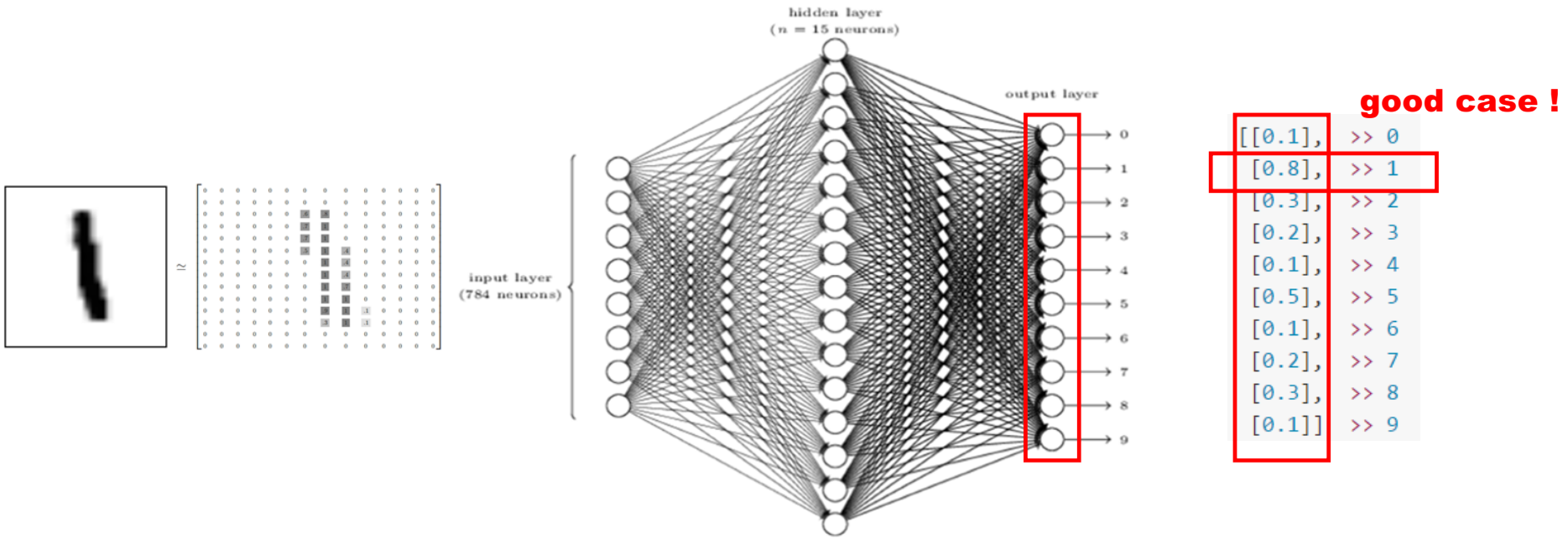
MNIST Example



MNIST Example



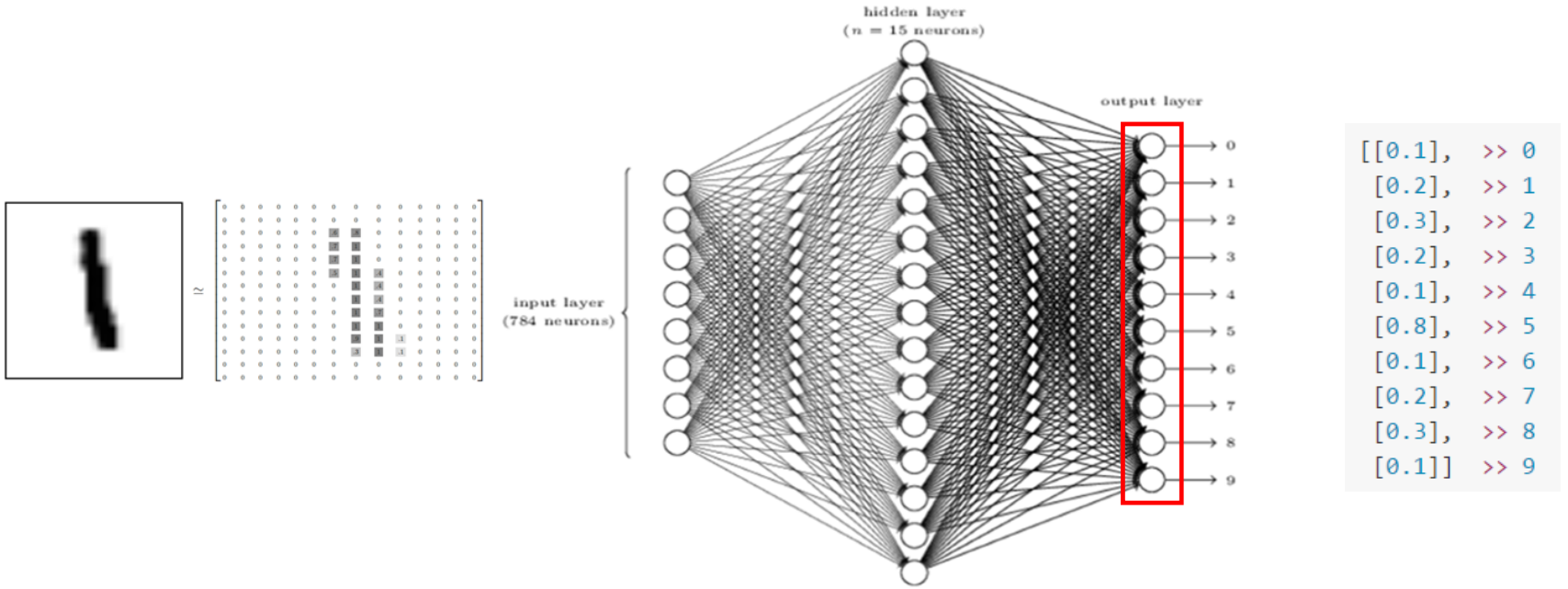
MNIST Example



good case !

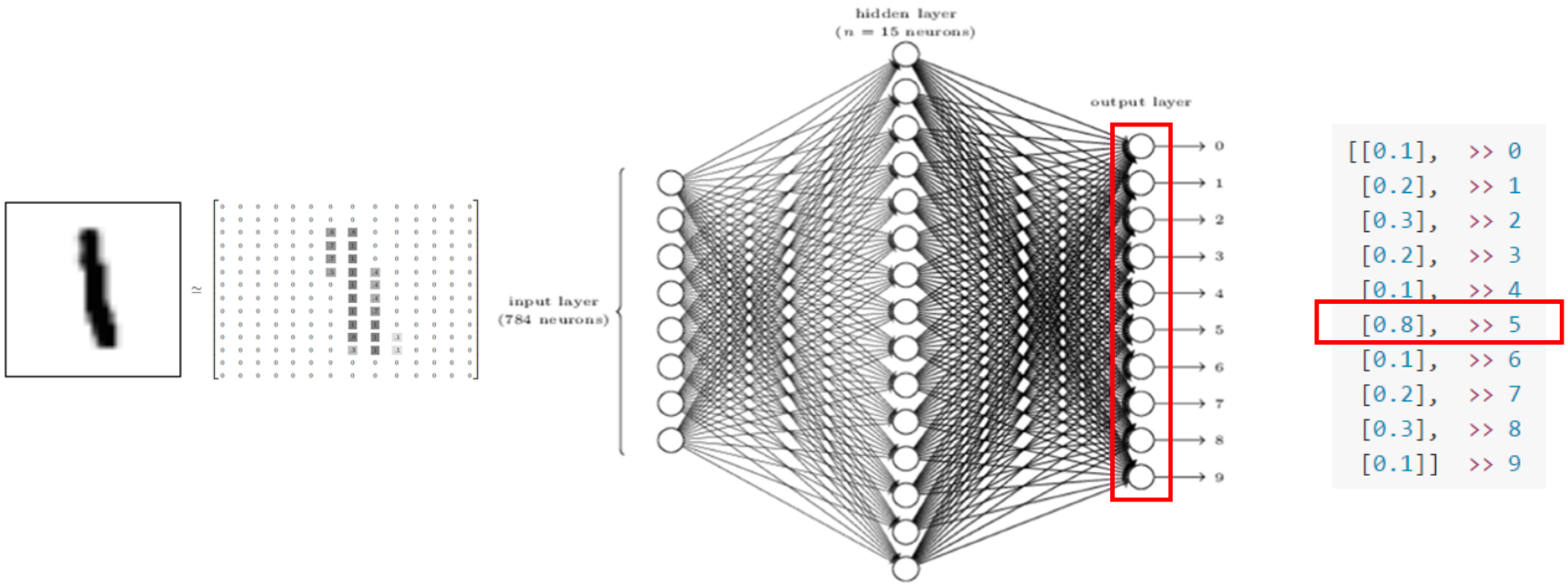
```
[[0.1],   >> 0
 [0.8],   >> 1
 [0.3],   >> 2
 [0.2],   >> 3
 [0.1],   >> 4
 [0.5],   >> 5
 [0.1],   >> 6
 [0.2],   >> 7
 [0.3],   >> 8
 [0.1]]   >> 9
```

MNIST Example



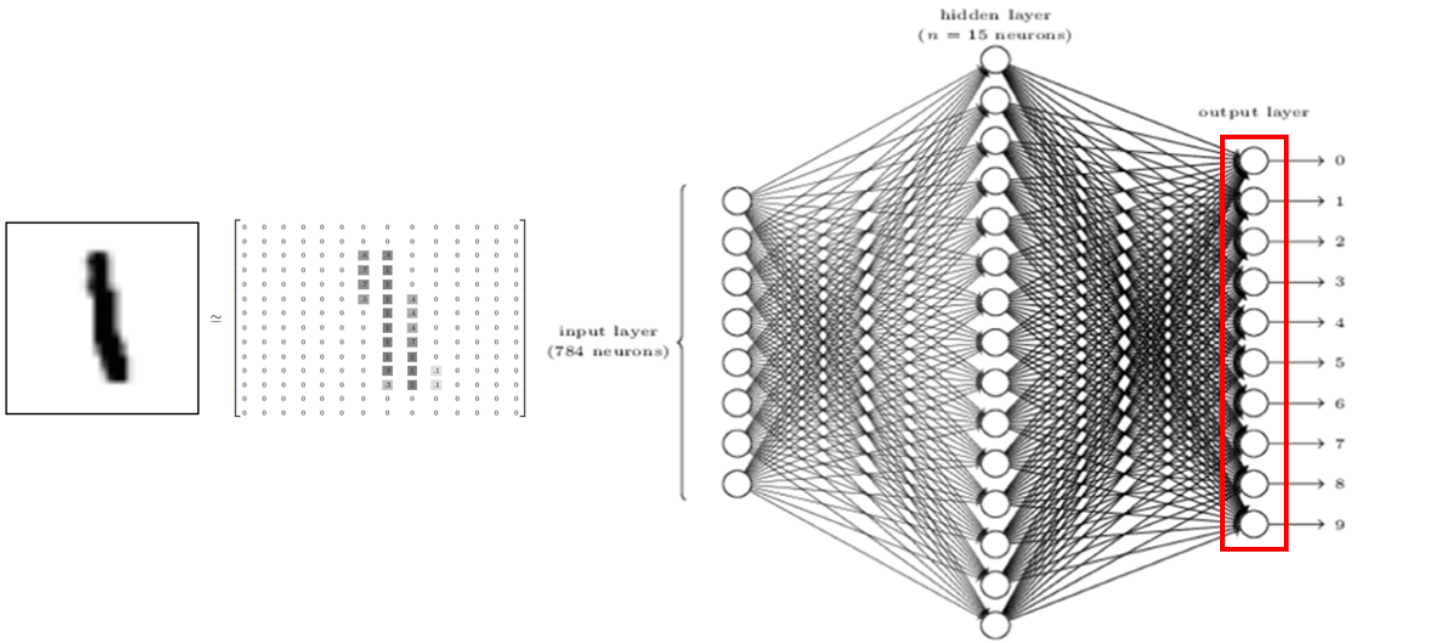

```
[[0.1],  >> 0
 [0.2],  >> 1
 [0.3],  >> 2
 [0.2],  >> 3
 [0.1],  >> 4
 [0.8],  >> 5
 [0.1],  >> 6
 [0.2],  >> 7
 [0.3],  >> 8
 [0.1]]  >> 9
```

MNIST Example




```
[[0.1],  >> 0
 [0.2],  >> 1
 [0.3],  >> 2
 [0.2],  >> 3
 [0.1],  >> 4
 [0.8],  >> 5
 [0.1],  >> 6
 [0.2],  >> 7
 [0.3],  >> 8
 [0.1]]  >> 9
```

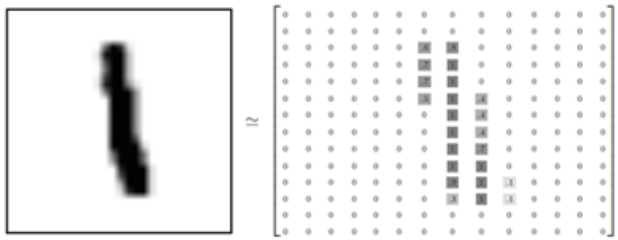
MNIST Example




```
[[0.1],   >> 0
 [0.2],   >> 1
 [0.3],   >> 2
 [0.2],   >> 3
 [0.1],   >> 4
 [0.8],   >> 5
 [0.1],   >> 6
 [0.2],   >> 7
 [0.3],   >> 8
 [0.1]]   >> 9
```

bad case !

MNIST Example



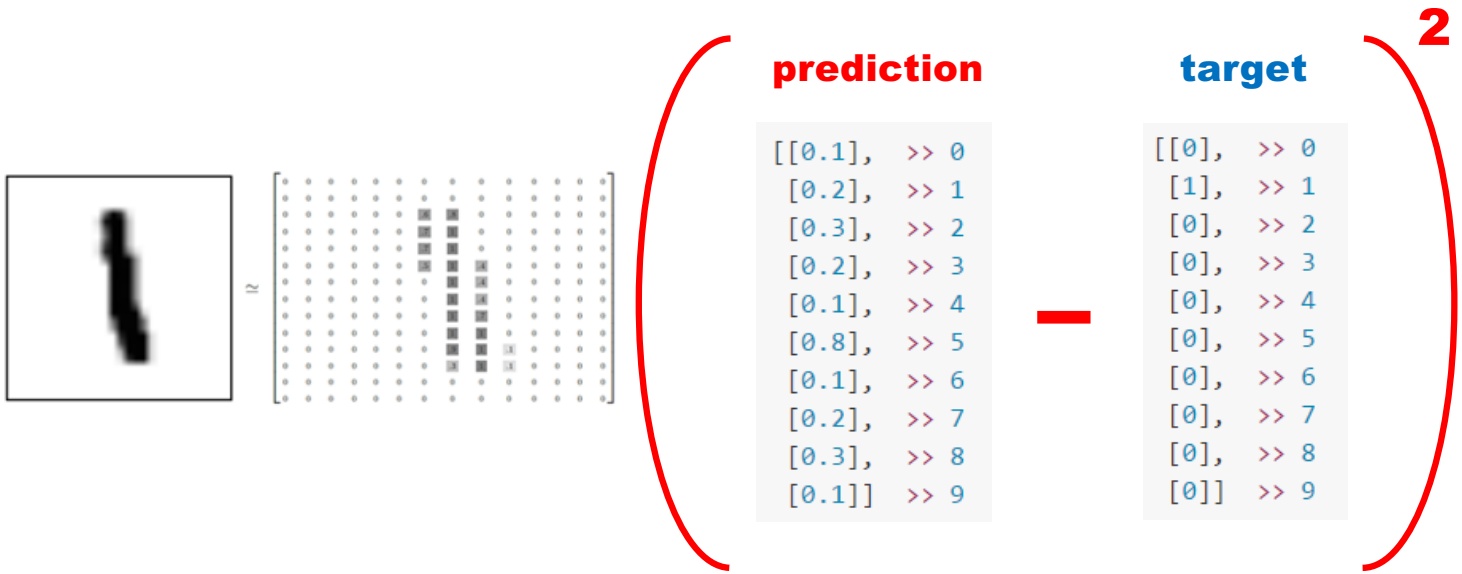
prediction

```
[[0.1],  >> 0
 [0.2],  >> 1
 [0.3],  >> 2
 [0.2],  >> 3
 [0.1],  >> 4
 [0.8],  >> 5
 [0.1],  >> 6
 [0.2],  >> 7
 [0.3],  >> 8
 [0.1]]  >> 9
```

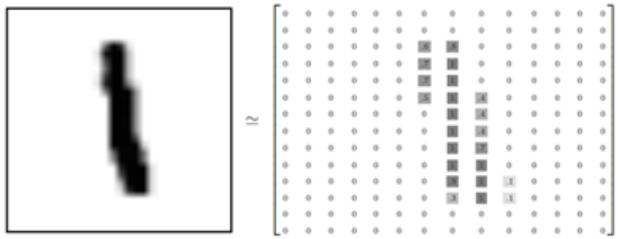
target

```
[[0],  >> 0
 [1],  >> 1
 [0],  >> 2
 [0],  >> 3
 [0],  >> 4
 [0],  >> 5
 [0],  >> 6
 [0],  >> 7
 [0],  >> 8
 [0]]  >> 9
```

MNIST Example



MNIST Example



prediction

```
[[0.1],  >> 0
 [0.2],  >> 1
 [0.3],  >> 2
 [0.2],  >> 3
 [0.1],  >> 4
 [0.8],  >> 5
 [0.1],  >> 6
 [0.2],  >> 7
 [0.3],  >> 8
 [0.1]]  >> 9
```

target

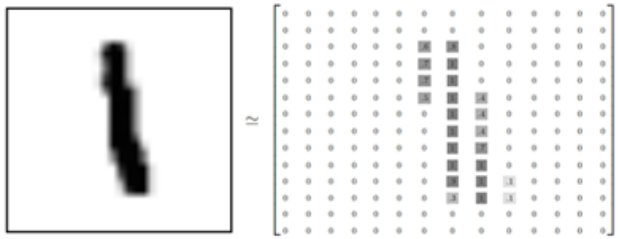
```
[[0],  >> 0
 [1],  >> 1
 [0],  >> 2
 [0],  >> 3
 [0],  >> 4
 [0],  >> 5
 [0],  >> 6
 [0],  >> 7
 [0],  >> 8
 [0]]  >> 9
```

$$(\text{prediction} - \text{target})^2 =$$

```
[[0.01], >> 0
 [0.64],  >> 1
 [0.09],  >> 2
 [0.04],  >> 3
 [0.01],  >> 4
 [0.64],  >> 5
 [0.1],   >> 6
 [0.04],  >> 7
 [0.09],  >> 8
 [0.01]]  >> 9
```

```
squared error: 1.67
```

MNIST Example



prediction

```
[[0.1],  >> 0
 [0.2],  >> 1
 [0.3],  >> 2
 [0.2],  >> 3
 [0.1],  >> 4
 [0.8],  >> 5
 [0.1],  >> 6
 [0.2],  >> 7
 [0.3],  >> 8
 [0.1]]  >> 9
```

target

```
[[0],  >> 0
 [1],  >> 1
 [0],  >> 2
 [0],  >> 3
 [0],  >> 4
 [0],  >> 5
 [0],  >> 6
 [0],  >> 7
 [0],  >> 8
 [0]]  >> 9
```

$$\text{(prediction} - \text{target)}^2 =$$

```
[[0.01], >> 0
 [0.64],  >> 1
 [0.09],  >> 2
 [0.04],  >> 3
 [0.01],  >> 4
 [0.64],  >> 5
 [0.1],   >> 6
 [0.04],  >> 7
 [0.09],  >> 8
 [0.01]]  >> 9
```

`squared error: 1.67`

we should minimize this error

Cost Function

Assume we have 1 data and dimension of output vector is n.

Regression

Mean Squared Error

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

```
[[0.1],   >> 0
 [0.2],   >> 1
 [0.3],   >> 2
 [0.2],   >> 3
 [0.1],   >> 4
 [0.8],   >> 5
 [0.1],   >> 6
 [0.2],   >> 7
 [0.3],   >> 8
 [0.1]]   >> 9
```

```
[[0],   >> 0
 [1],   >> 1
 [0],   >> 2
 [0],   >> 3
 [0],   >> 4
 [0],   >> 5
 [0],   >> 6
 [0],   >> 7
 [0],   >> 8
 [0]]   >> 9
```

Classification

Sigmoid Cross Entropy Loss

$$\sum_{i=1}^{n} -y_i \log \hat{y}_i - (1-y_i)\log(1-\hat{y}_i)$$

Softmax Cross Entropy Loss

$$\sum_{i=1}^{n} -y_i \log \hat{y}_i$$

Cost Function

Assume we have 1 data and dimension of output vector is n.

Regression

Mean Squared Error

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

Classification

Sigmoid Cross Entropy Loss

$$\sum_{i=1}^{n} -y_i \log \hat{y}_i - (1-y_i)\log(1-\hat{y}_i)$$

when using sigmoid function in last layer

Softmax Cross Entropy Loss

$$\sum_{i=1}^{n} -y_i \log \hat{y}_i$$

Cost Function

Assume we have 1 data and dimension of output vector is n.

Regression

Mean Squared Error

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

Classification

Sigmoid Cross Entropy Loss

$$\sum_{i=1}^{n} -y_i \log \hat{y}_i - (1-y_i)\log(1-\hat{y}_i)$$

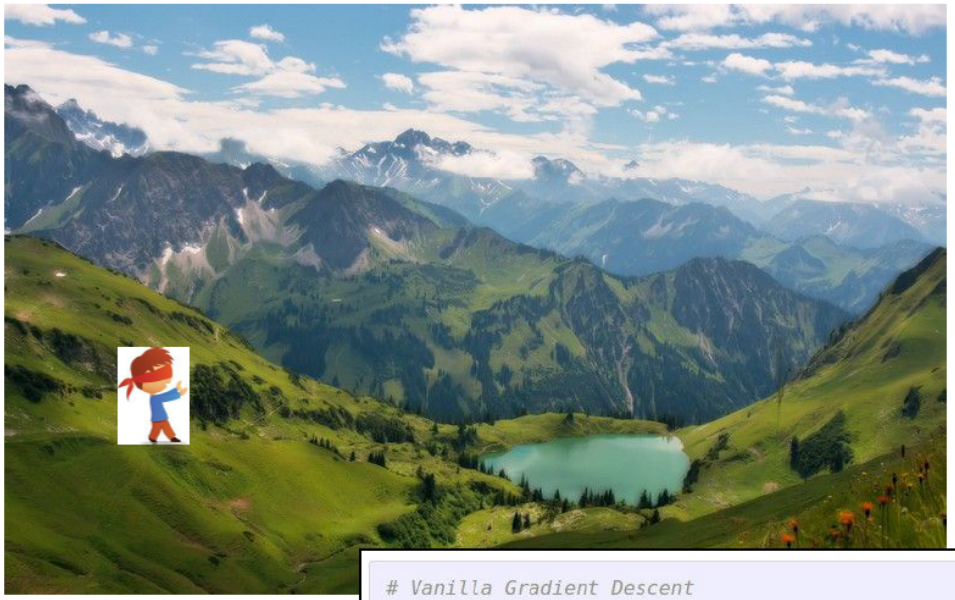
Softmax Cross Entropy Loss

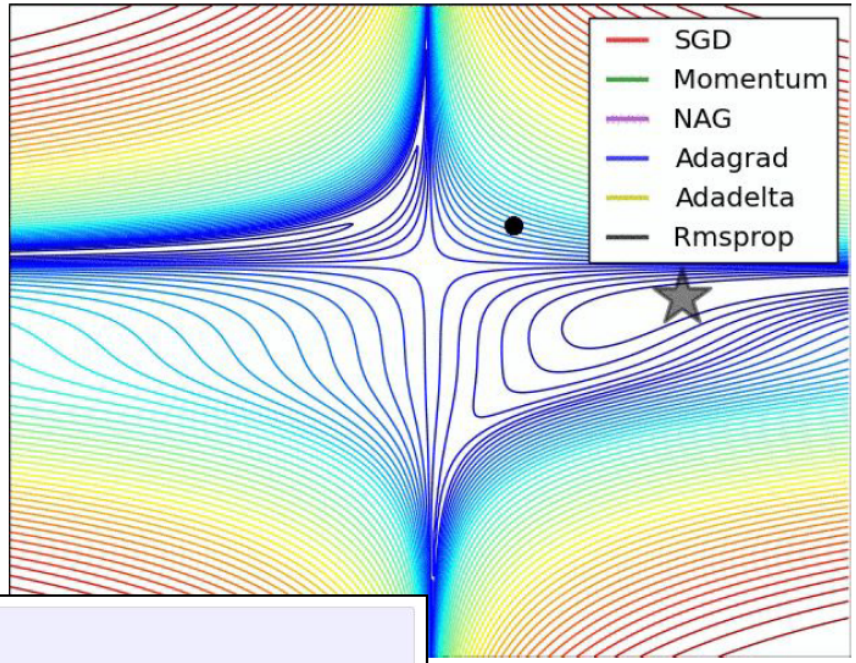
when using softmax function in last layer

$$\sum_{i=1}^{n} -y_i \log \hat{y}_i$$

5. Backpropagation

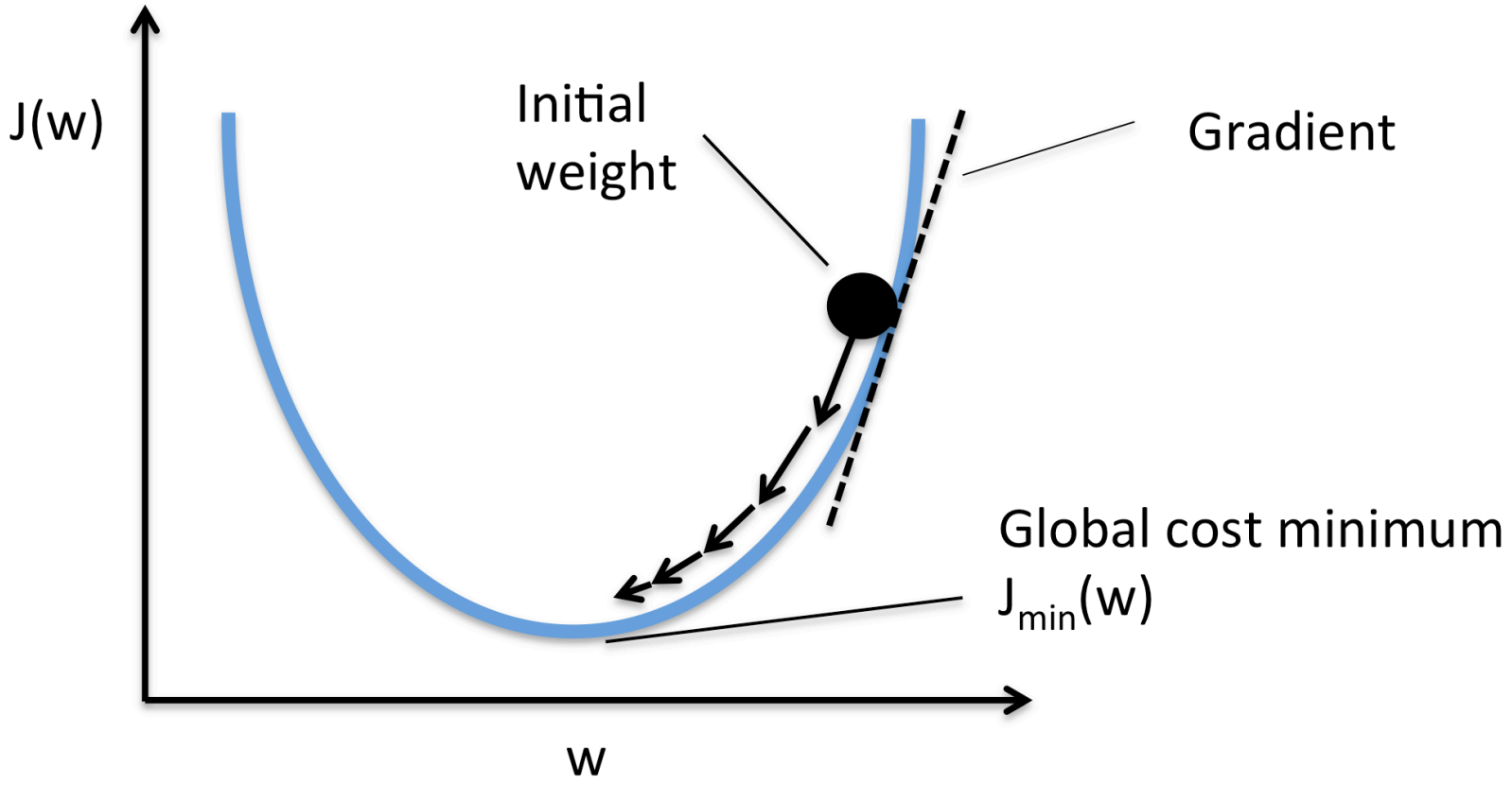
Gradient Descent



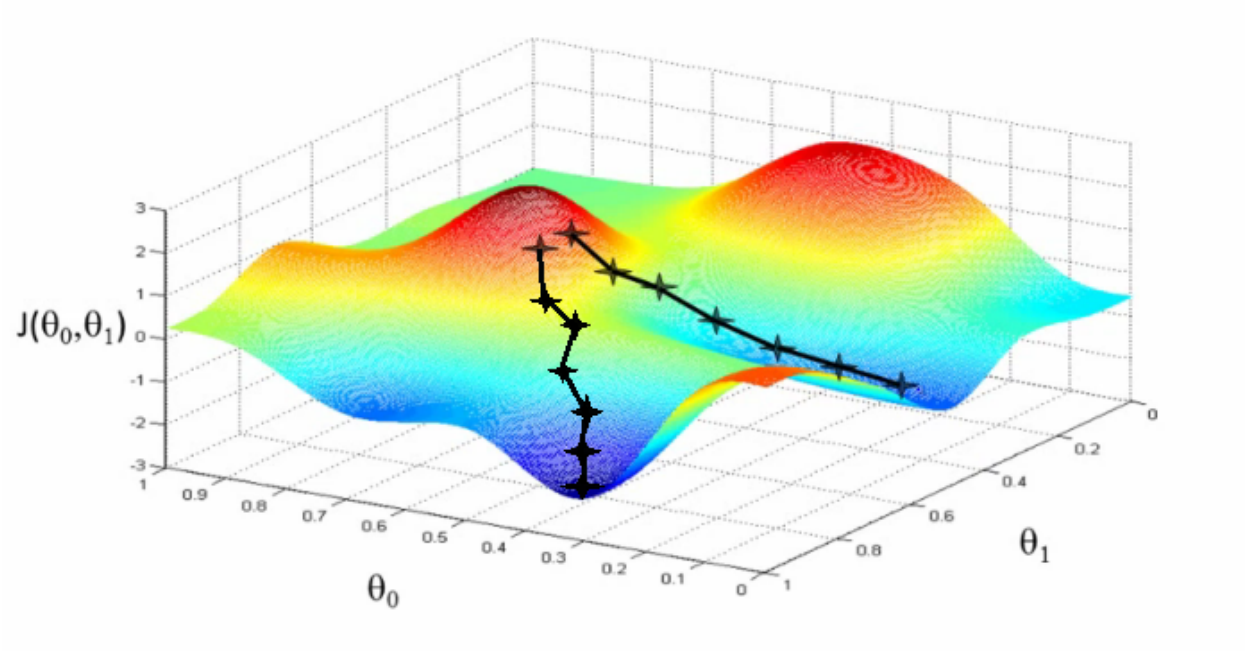



```
# Vanilla Gradient Descent

while True:
  weights_grad = evaluate_gradient(loss_fun, data, weights)
  weights += - step_size * weights_grad # perform parameter update
```

Gradient Descent



Gradient Descent



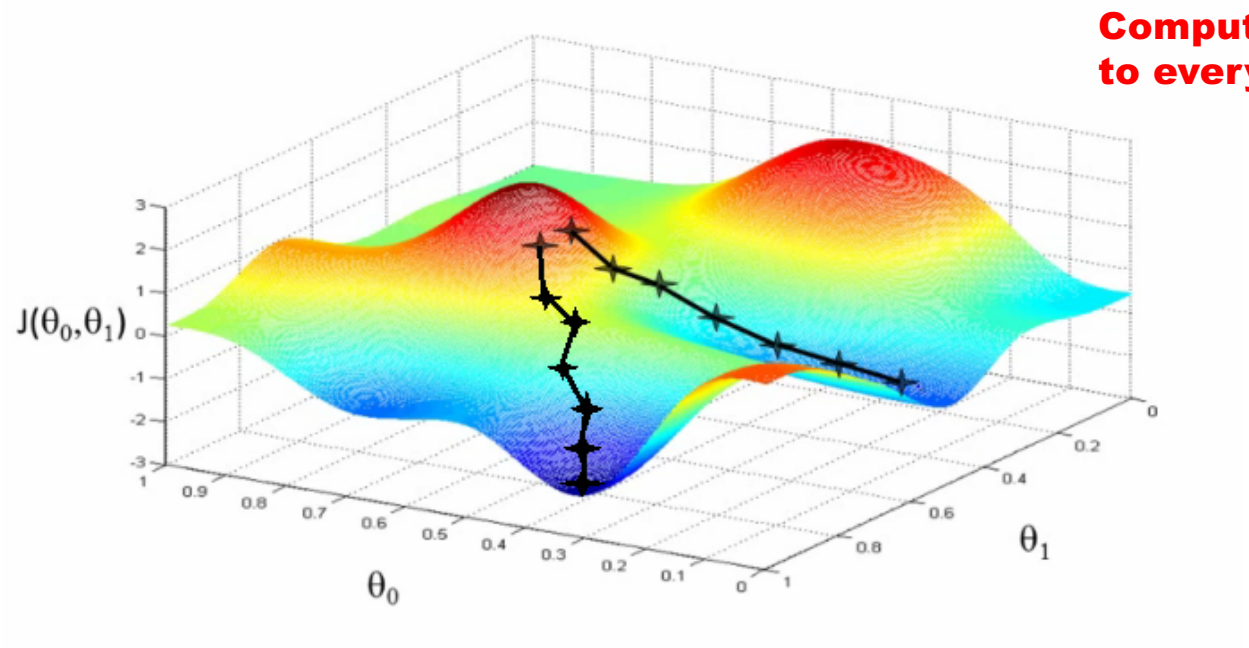
$$compute \quad \frac{\partial J}{\partial \theta_0} \quad and \quad \frac{\partial J}{\partial \theta_1}$$

update weights with

$$\theta_0 := \theta_0 - \alpha \cdot \frac{\partial J}{\partial \theta_0}$$

$$\theta_1 := \theta_1 - \alpha \cdot \frac{\partial J}{\partial \theta_1}$$

Gradient Descent



Compute gradient of loss with respect to every weights (parameters)

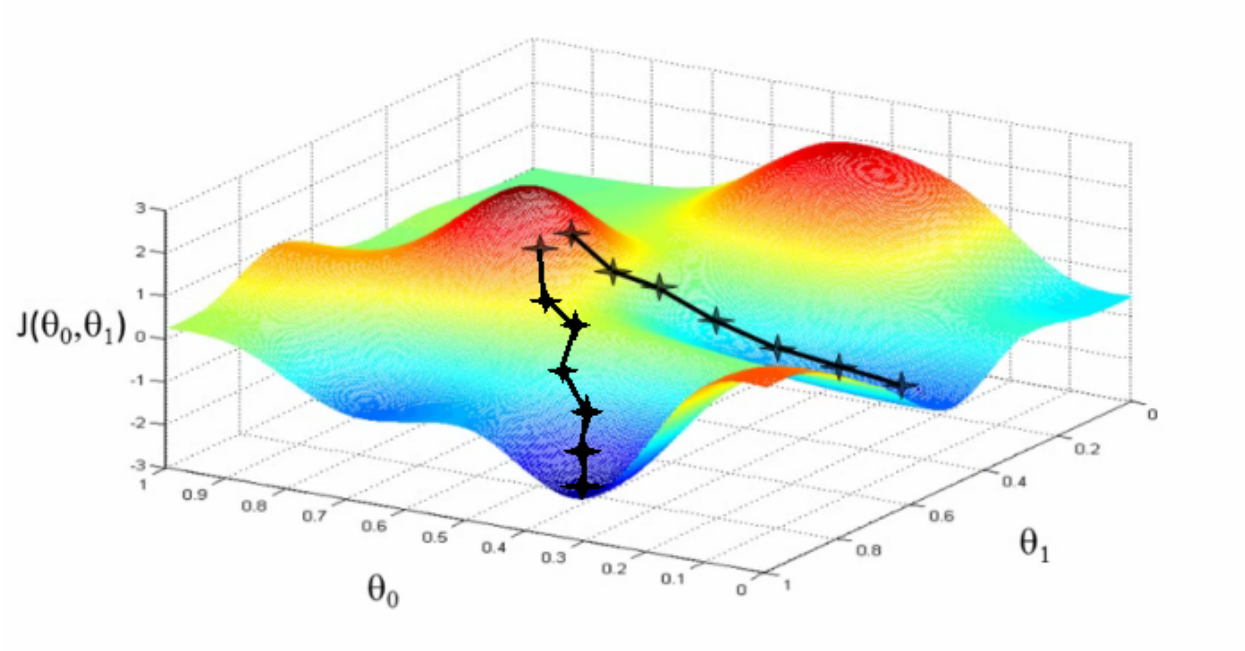
$$compute \quad \frac{\partial J}{\partial \theta_0} \quad and \quad \frac{\partial J}{\partial \theta_1}$$

update weights with

$$\theta_0 := \theta_0 - \alpha \cdot \frac{\partial J}{\partial \theta_0}$$

$$\theta_1 := \theta_1 - \alpha \cdot \frac{\partial J}{\partial \theta_1}$$

Gradient Descent



$$compute \quad \frac{\partial J}{\partial \theta_0} \quad and \quad \frac{\partial J}{\partial \theta_1}$$

update weights with

$$\theta_0 := \theta_0 - \alpha \cdot \frac{\partial J}{\partial \theta_0}$$

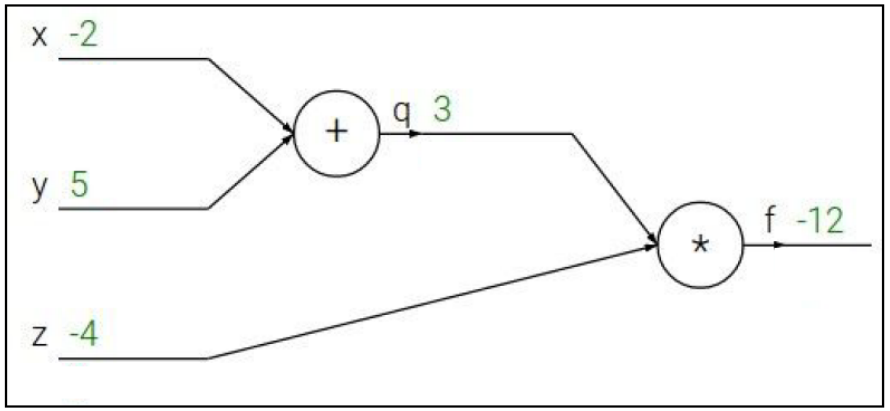
$$\theta_1 := \theta_1 - \alpha \cdot \frac{\partial J}{\partial \theta_1}$$

Gradient "descent" optimization with learning rate 'alpha' (e.g. 0.01)

Computational Graph

$$f(x, y, z) = (x + y)z$$

e.g. x = -2, y = 5, z = -4



Computational Graph

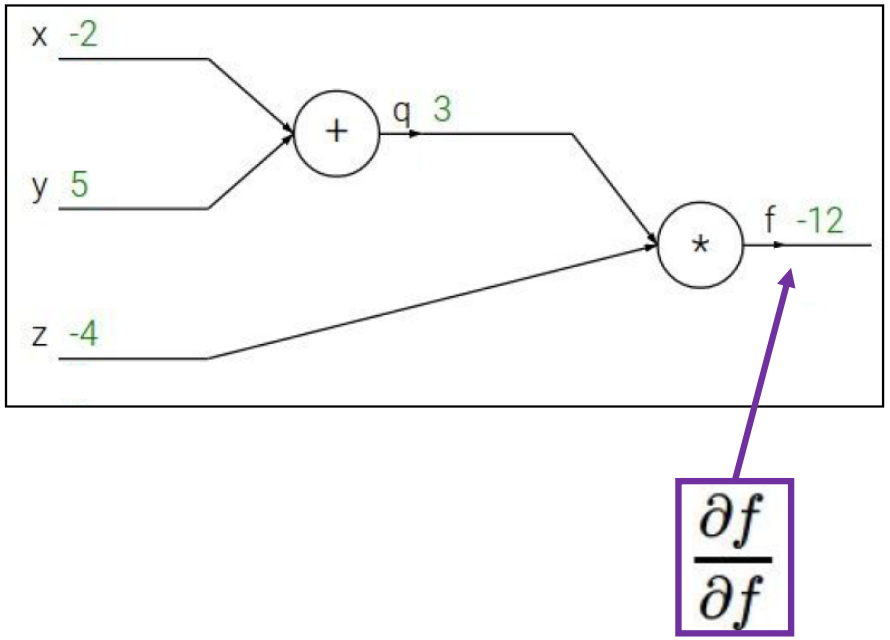
$$f(x, y, z) = (x + y)z$$

e.g. x = -2, y = 5, z = -4

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



Computational Graph

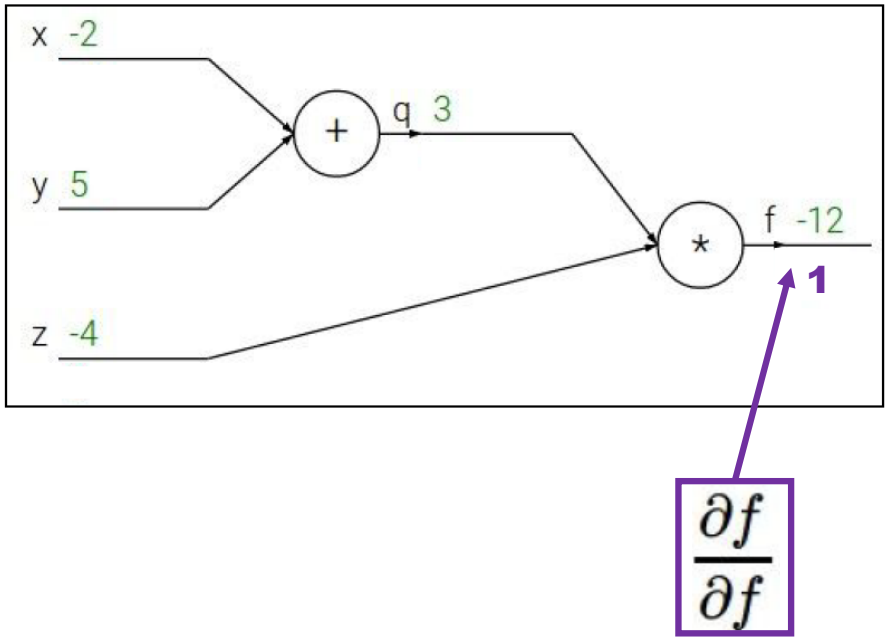
$$f(x, y, z) = (x + y)z$$

e.g. x = -2, y = 5, z = -4

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



Backpropagation using Chain Rule

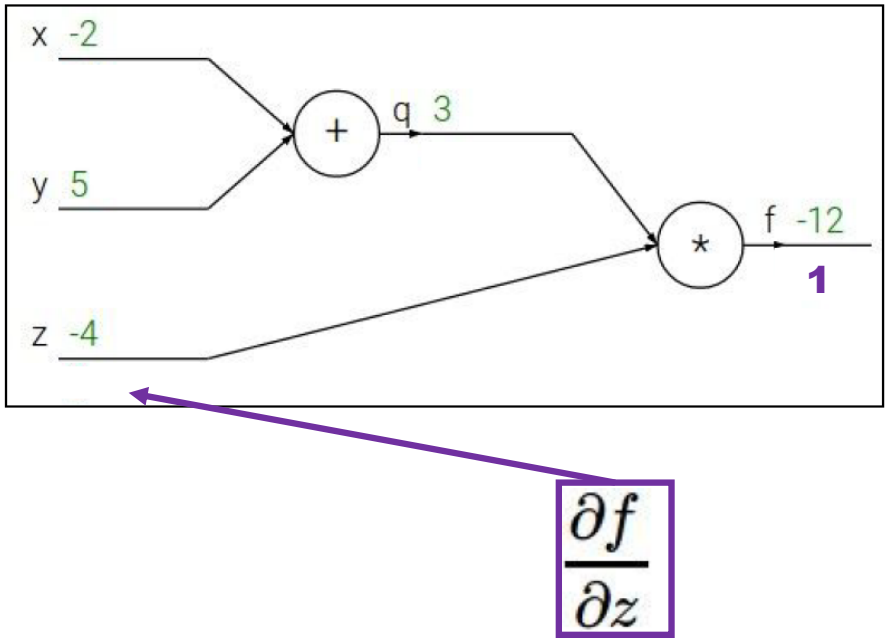
$$f(x, y, z) = (x + y)z$$

e.g. x = -2, y = 5, z = -4

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



Backpropagation using Chain Rule

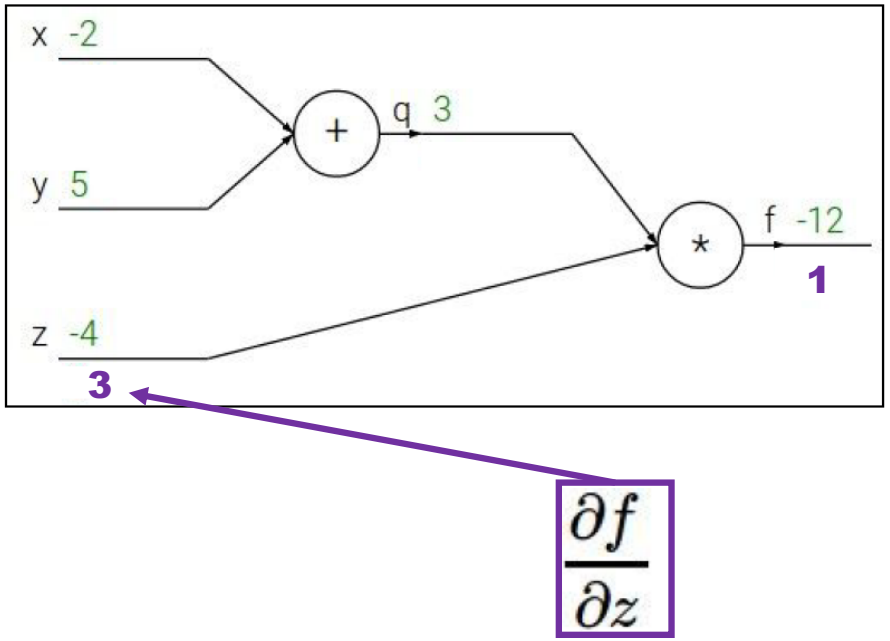
$$f(x, y, z) = (x + y)z$$

e.g. x = -2, y = 5, z = -4

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



Computational Graph

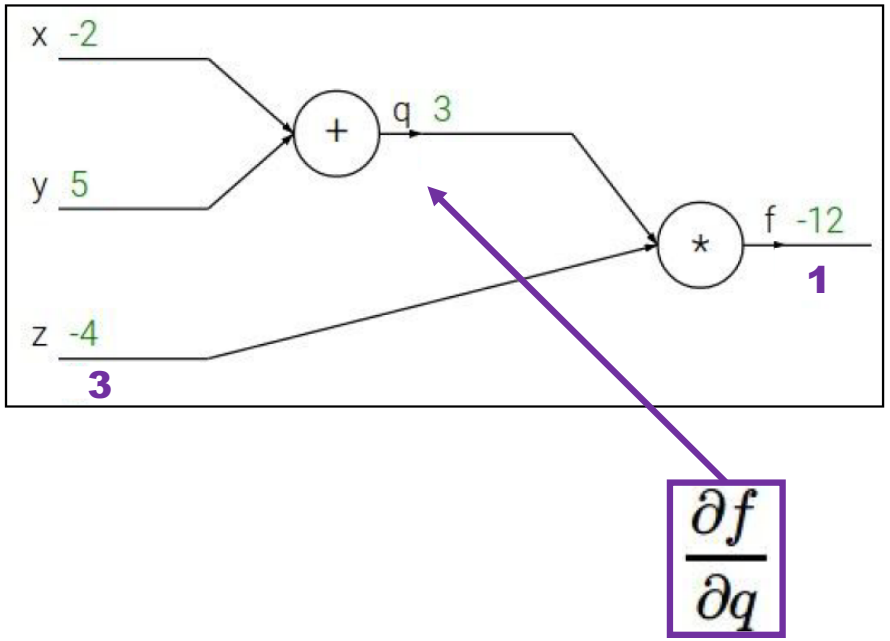
$$f(x, y, z) = (x + y)z$$

e.g. x = -2, y = 5, z = -4

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



Backpropagation using Chain Rule

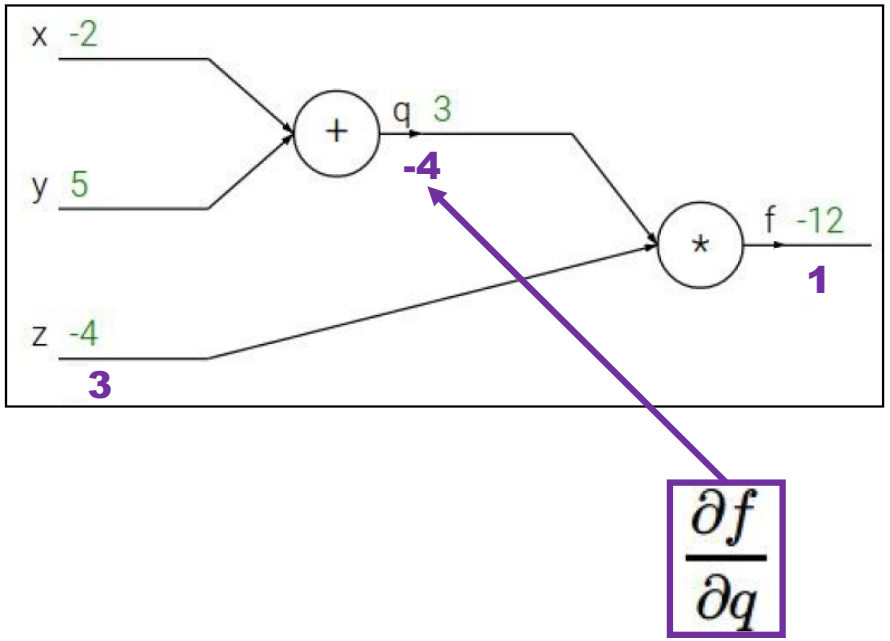
$$f(x, y, z) = (x + y)z$$

e.g. x = -2, y = 5, z = -4

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



Backpropagation using Chain Rule

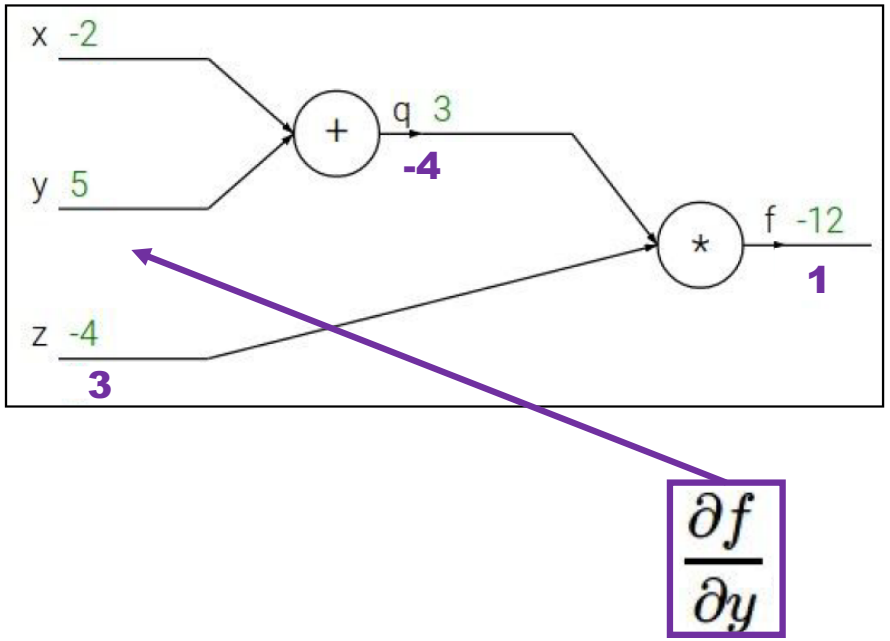
$$f(x, y, z) = (x + y)z$$

e.g. x = -2, y = 5, z = -4

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



Backpropagation using Chain Rule

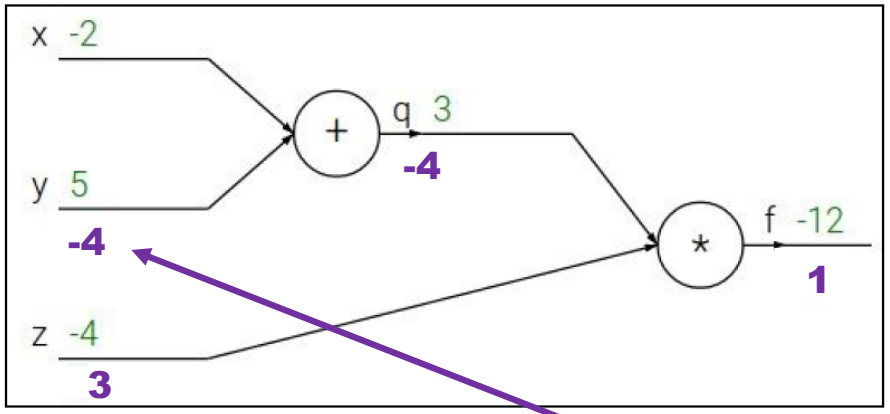
$$f(x, y, z) = (x + y)z$$

e.g. x = -2, y = 5, z = -4

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



$\frac{\partial f}{\partial y}$

Chain rule:

$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial y}$$

Backpropagation using Chain Rule

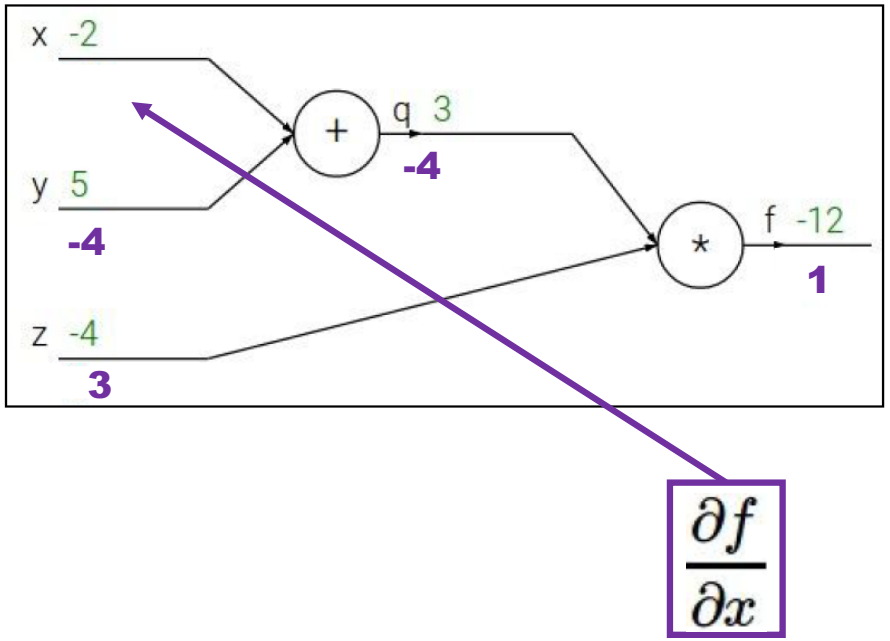
$$f(x, y, z) = (x + y)z$$

e.g. x = -2, y = 5, z = -4

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



Backpropagation using Chain Rule

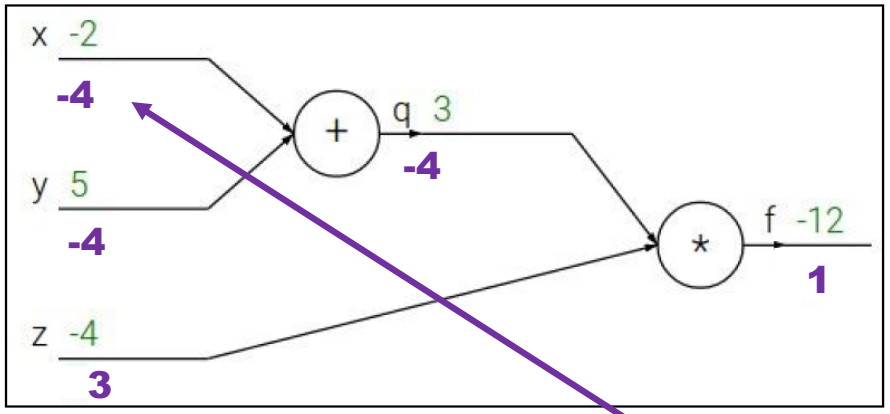
$$f(x, y, z) = (x + y)z$$

e.g. x = -2, y = 5, z = -4

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$

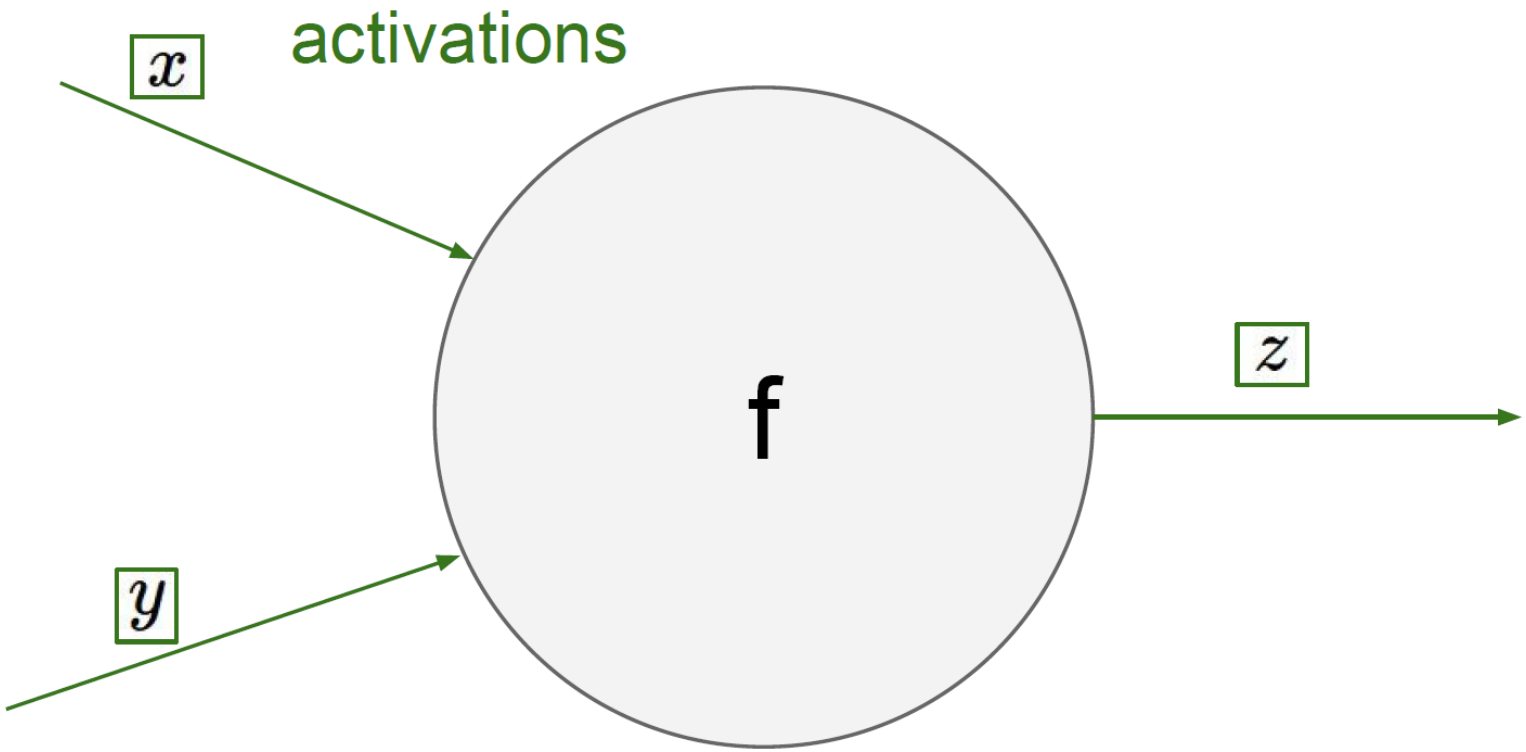


$\frac{\partial f}{\partial x}$

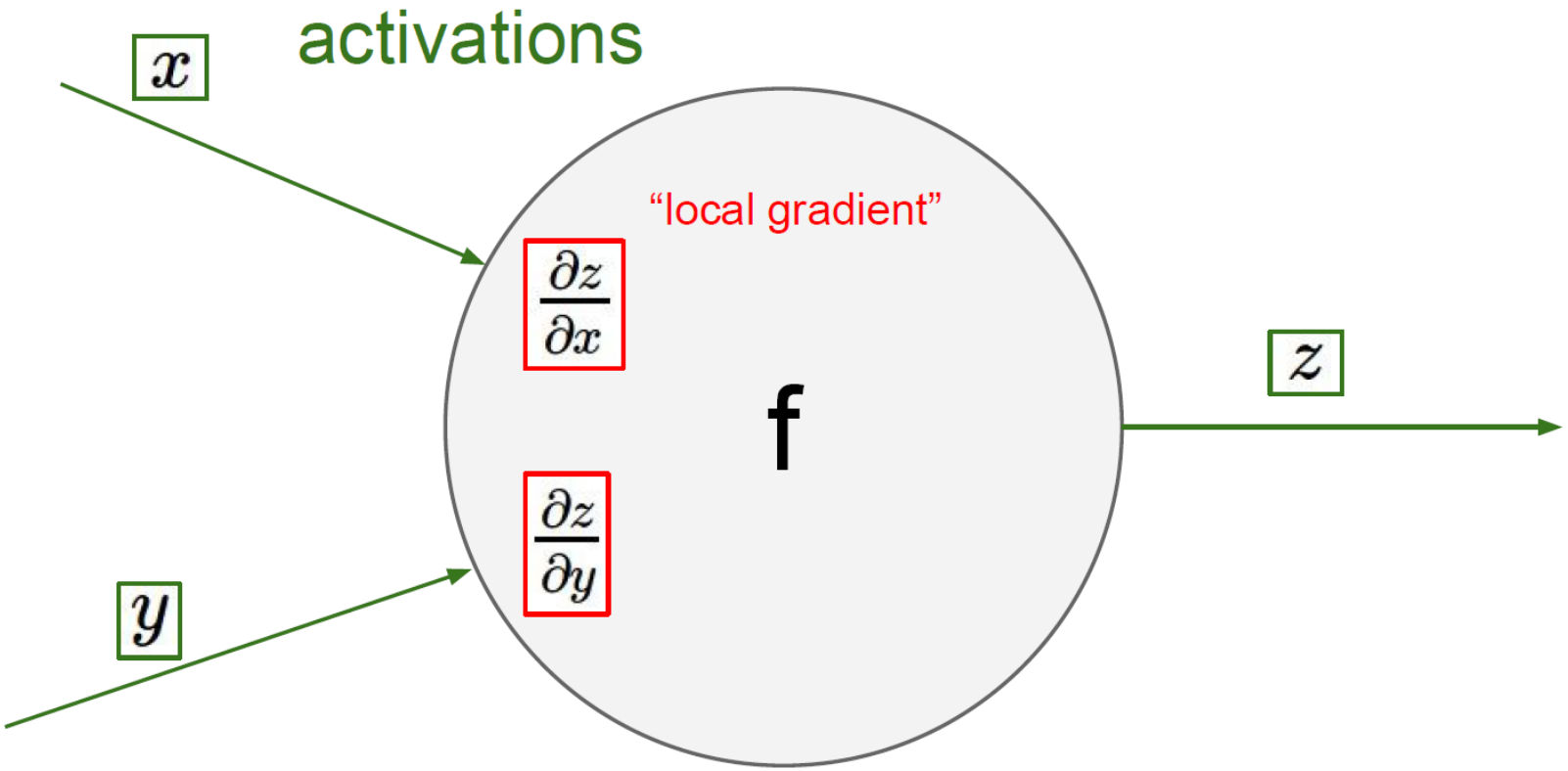
Chain rule:

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial x}$$

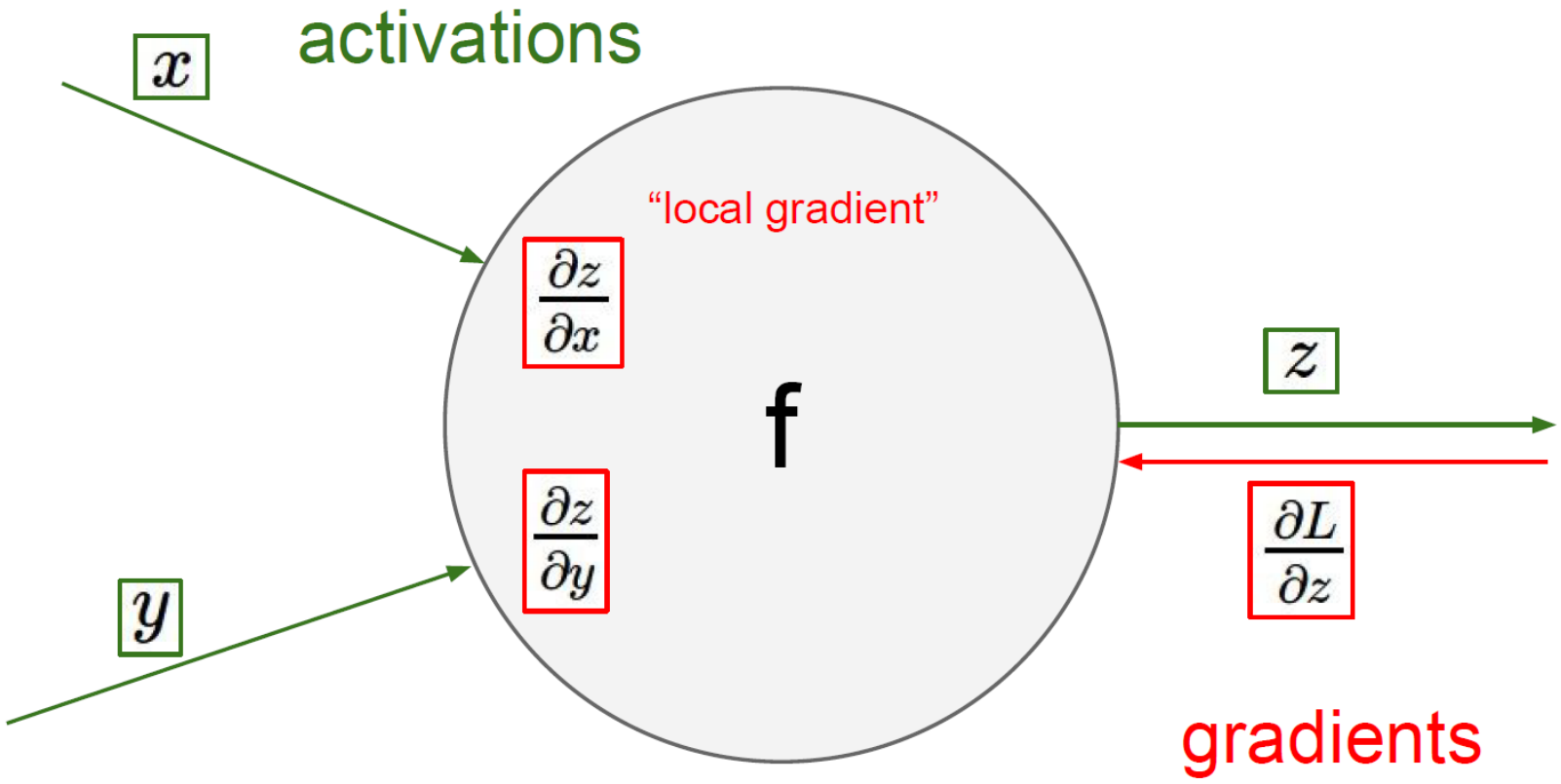
Backpropagation using Chain Rule



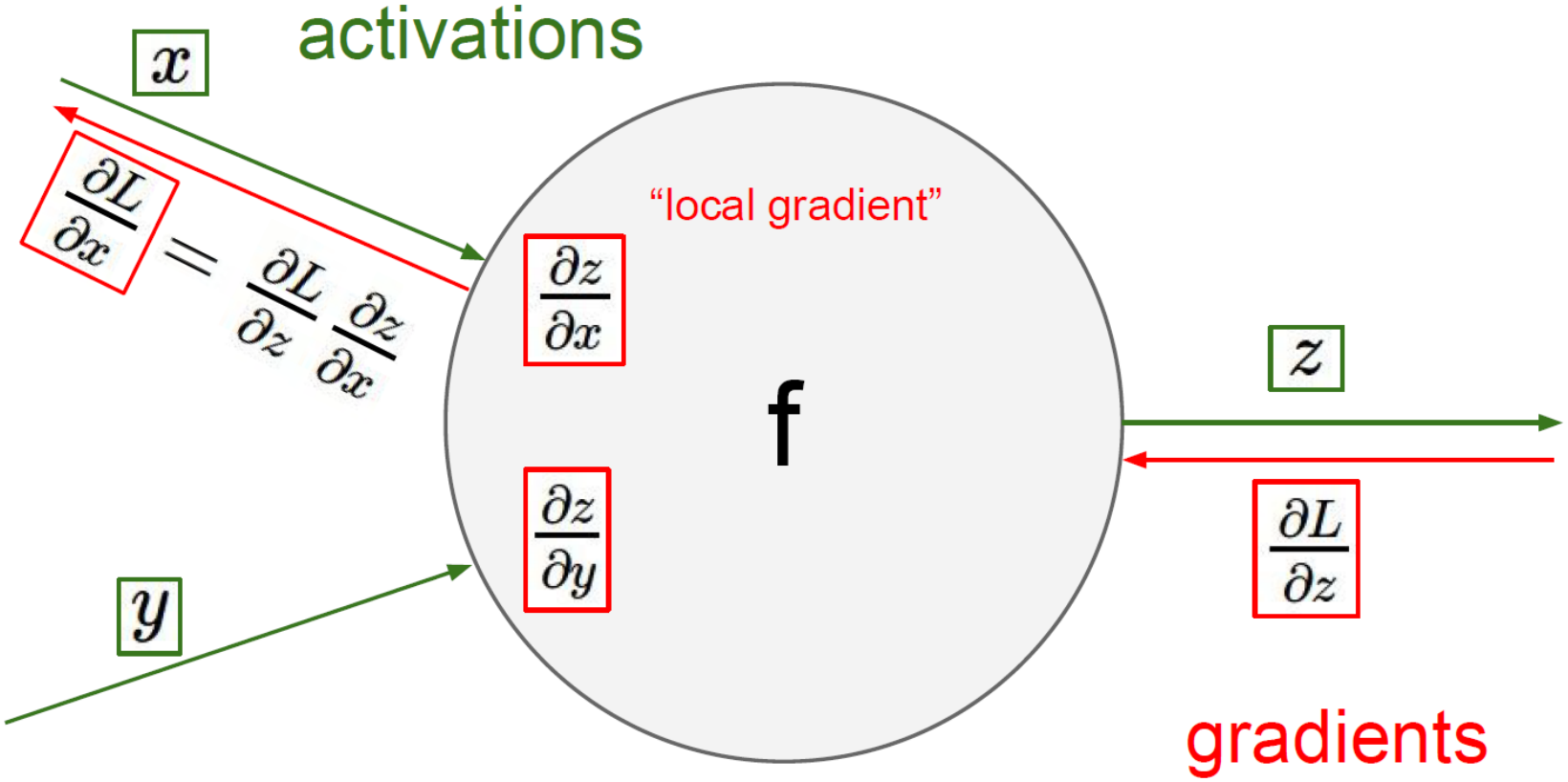
Backpropagation using Chain Rule



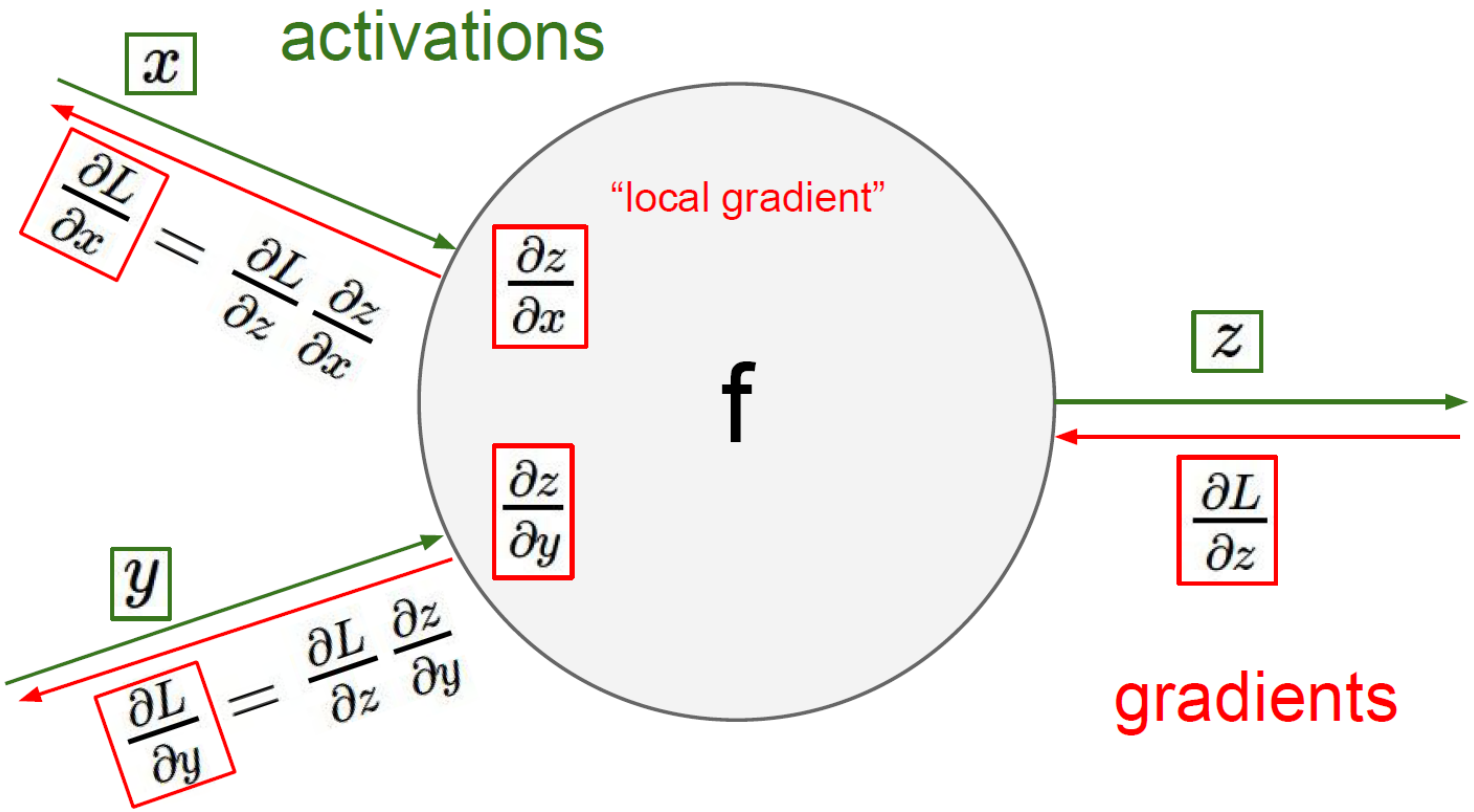
Backpropagation using Chain Rule



Backpropagation using Chain Rule

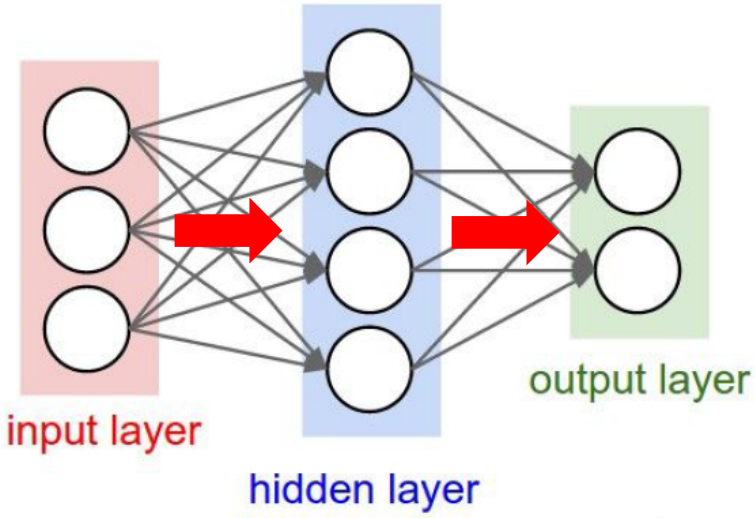


Backpropagation using Chain Rule



Training Neural Network

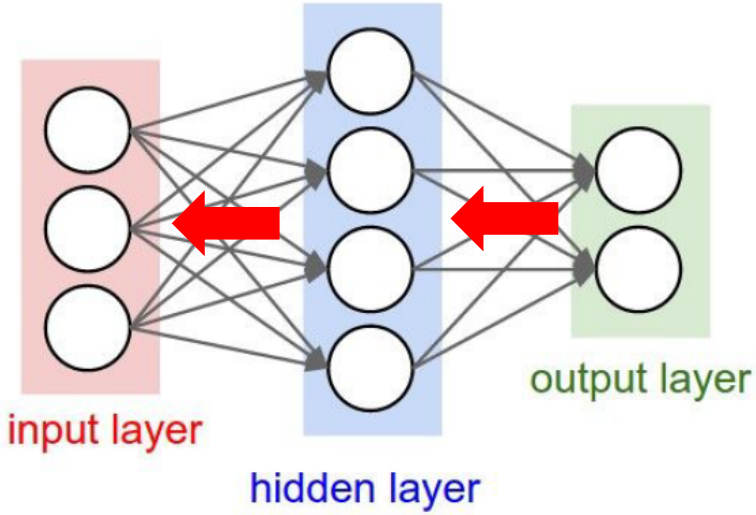
Forward propagate activations for given input data




```
x: 784 x 1                  a1: 785 x 1
w1: 15 x (784 + 1)          a2: 16 x 1
w2: 10 x (15 + 1)


import numpy as np

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def neural_network(x, w1, w2):
    a1 = np.concatenate([x, [[1]]], 0)
    z2 = np.dot(w1, a1)
    a2 = sigmoid(z2)
    a2 = np.concatenate([a2, [[1]]], 0)
    z3 = np.dot(w2, a2)
    a3 = sigmoid(z3)
    return a3
```

Training Neural Network

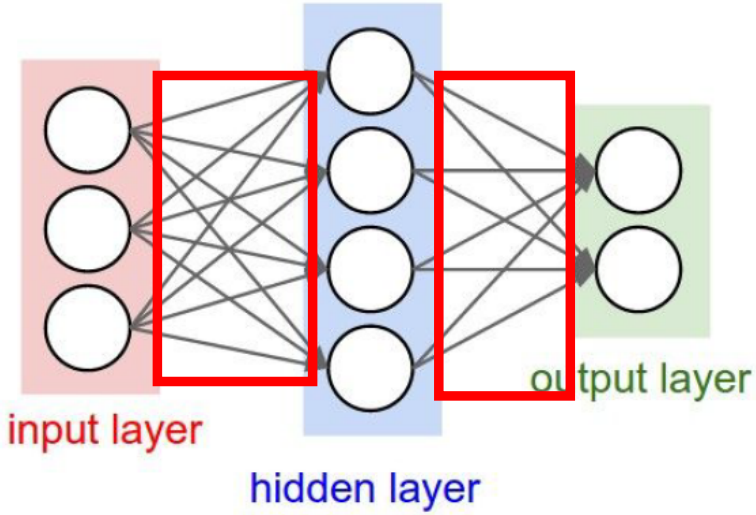
Backpropagate using chain rule and compute gradient of loss with respect to all weights



$$compute \quad \frac{\partial J}{\partial \theta_0} \quad and \quad \frac{\partial J}{\partial \theta_1}$$

Training Neural Network

Update weights with gradient descent optimization



$$update\ weights\ with$$

$$\theta_0 := \theta_0 - \alpha \cdot \frac{\partial J}{\partial \theta_0}$$

$$\theta_1 := \theta_1 - \alpha \cdot \frac{\partial J}{\partial \theta_1}$$

Deep Learning Framework

TensorFlow

K Keras

PYTORCH

mxnet

Deep Learning Community

References

Convolutional Neural Networks for Visual Recognition

Machine Learning - Stanford University | Coursera

Attention and Augmented Recurrent Neural Networks — Distill

Alec Radford's animations for optimization

A Neural Network Playground

DenseCap: Fully Convolutional Localization Networks for Dense Captioning