An Introduction to Statistical Learning - Chapter 10

Deep Learning

10th November 2024

What is Deep Learning?

Teaching computers how to **learn a task** directly from r**aw data**

Artificial Intelligence

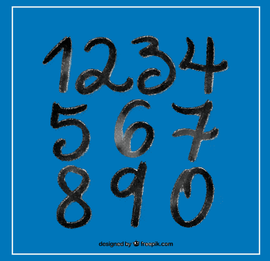
Any technique that enables computers to mimic human behaviour

Machine Learning

Ability to learn without explicitly being programmed

Deep Learning

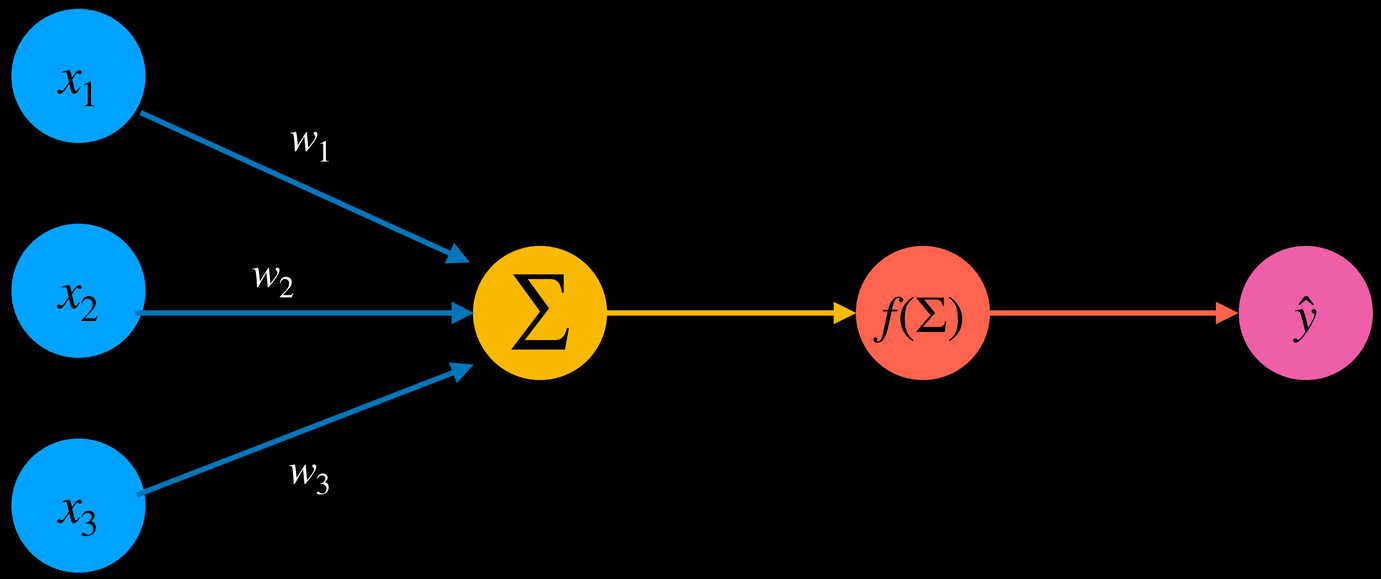
Extract patterns from data using neural networks



The Perceptron

The Perceptron

The building blocks of any neural network



Inputs Weights Sum Non-Linearity Output

Output

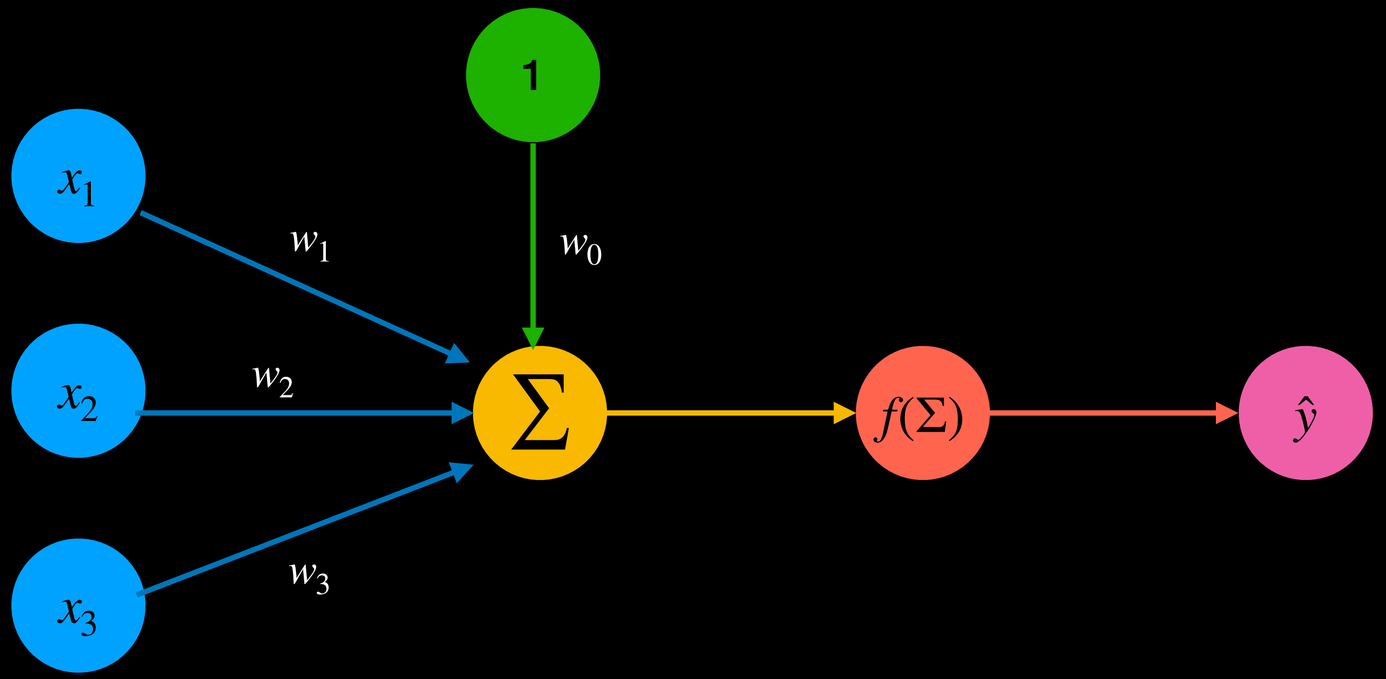
Linear Combination of Inputs

$$\hat{y} = f\left(\sum_{i=1}^{N} x_i w_i\right)$$

Non-Linear Activation Function

The Perceptron

Adding the bias term



Inputs Weights Sum Non-Linearity Output

Output

Linear Combination of Inputs

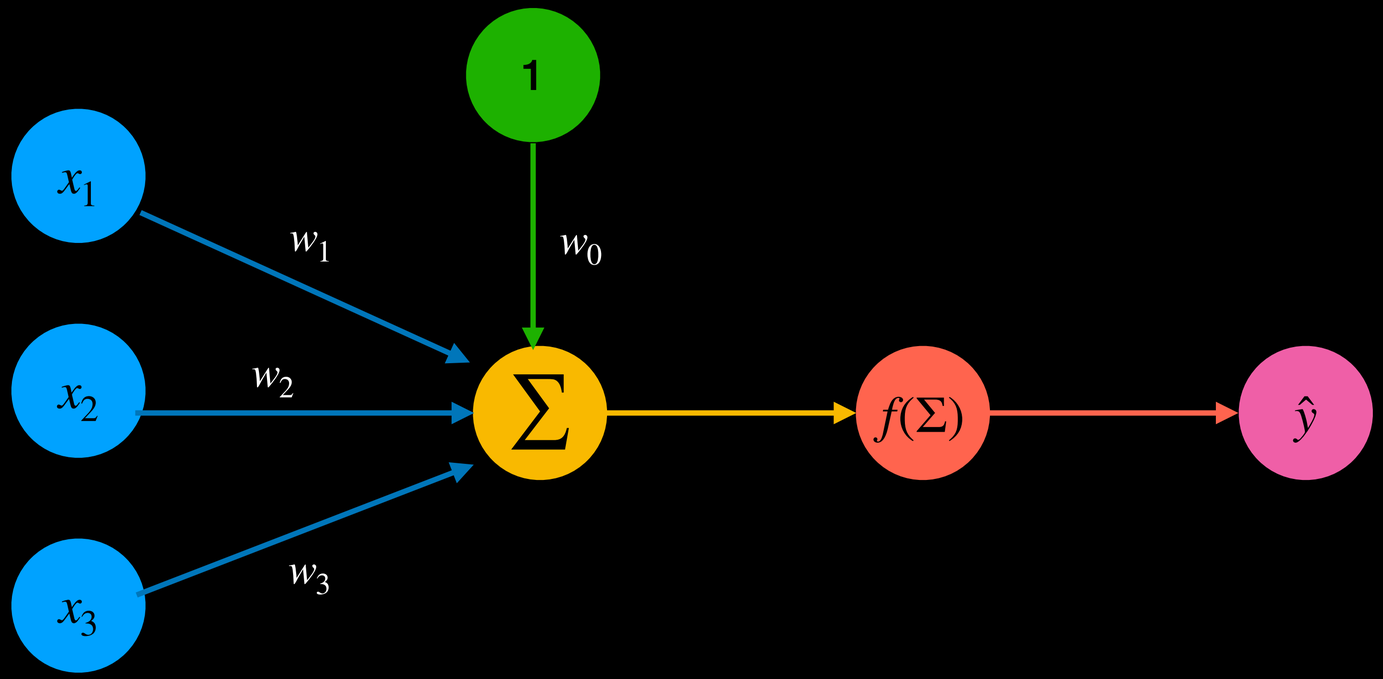
$$\hat{y} = f\left(w_0 + \sum_{i=1}^{N} x_i w_i\right)$$

Non-Linear Activation Function

Bias

The Perceptron

Simplifying Notation



Inputs Weights Sum Non-Linearity Output

$$\hat{y} = f\left(w_0 + \sum_{i=1}^{N} x_i w_i \right)$$

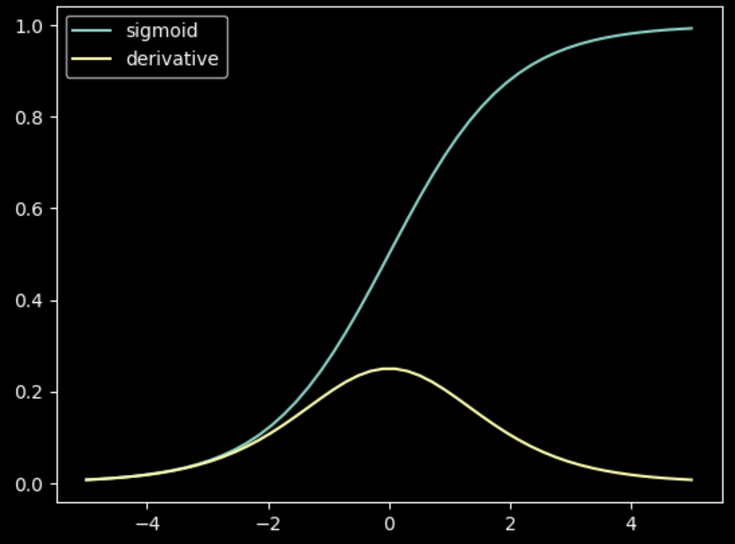
$$\hat{y} = f\left(w_0 + X^T W \right)$$

$$X = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{bmatrix} \quad W = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_N \end{bmatrix}$$

The Perceptron

Common Activation Functions

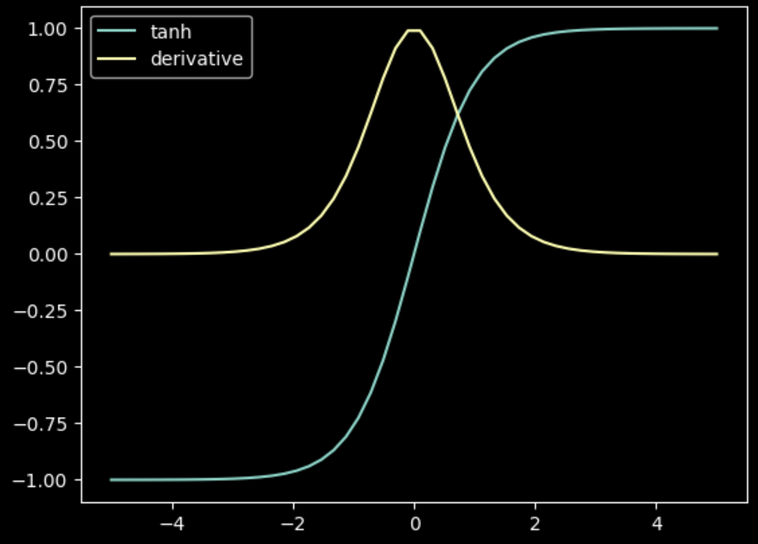
Sigmoid



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

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

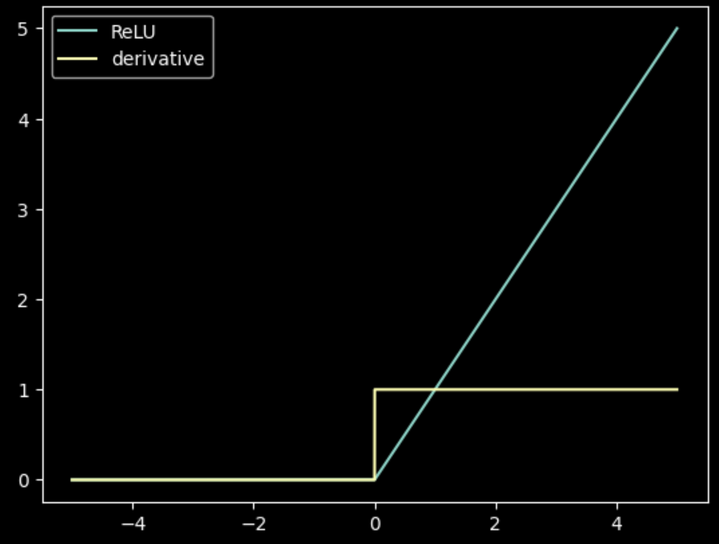
Hyperbolic Tangent



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

$$f'(z) = 1 - f(z)^2$$

Rectified Linear Unit (ReLU)

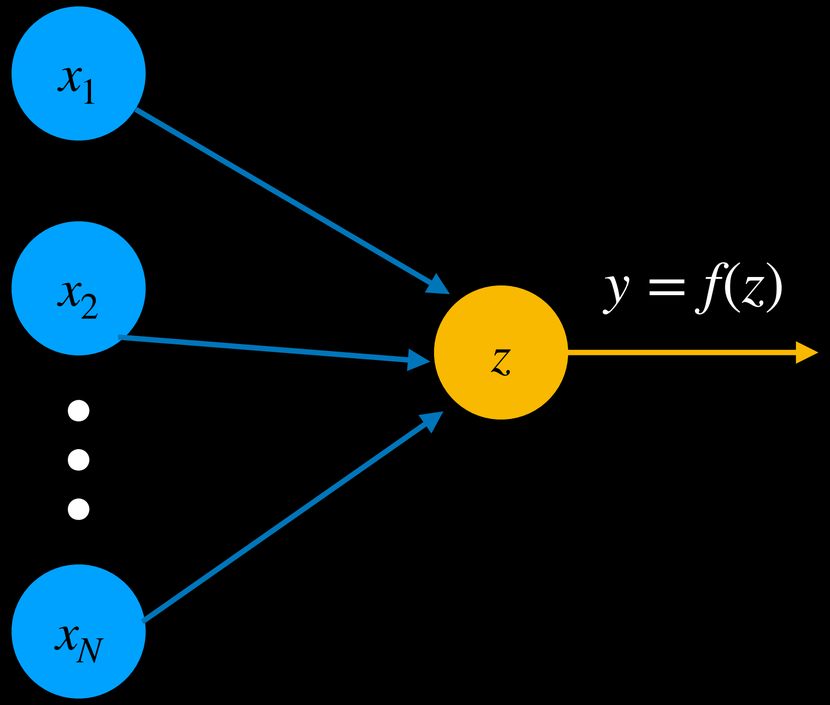


$$f(z) = max\ (\ 0\ ,\ z\ )$$

$$f'(z) = \begin{cases} 1, & z > 0 \\ 0, & \text{otherwise} \end{cases}$$

The Perceptron

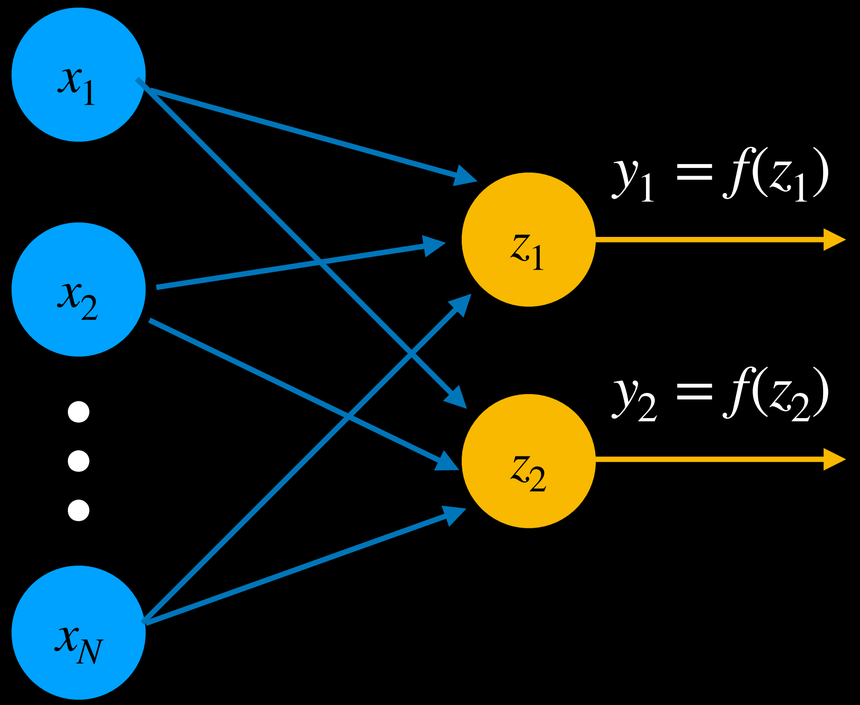
Simplified



$$z = w_0 + \sum_{i=1}^{N} x_i w_i$$

The Perceptron

Multi-Output Version

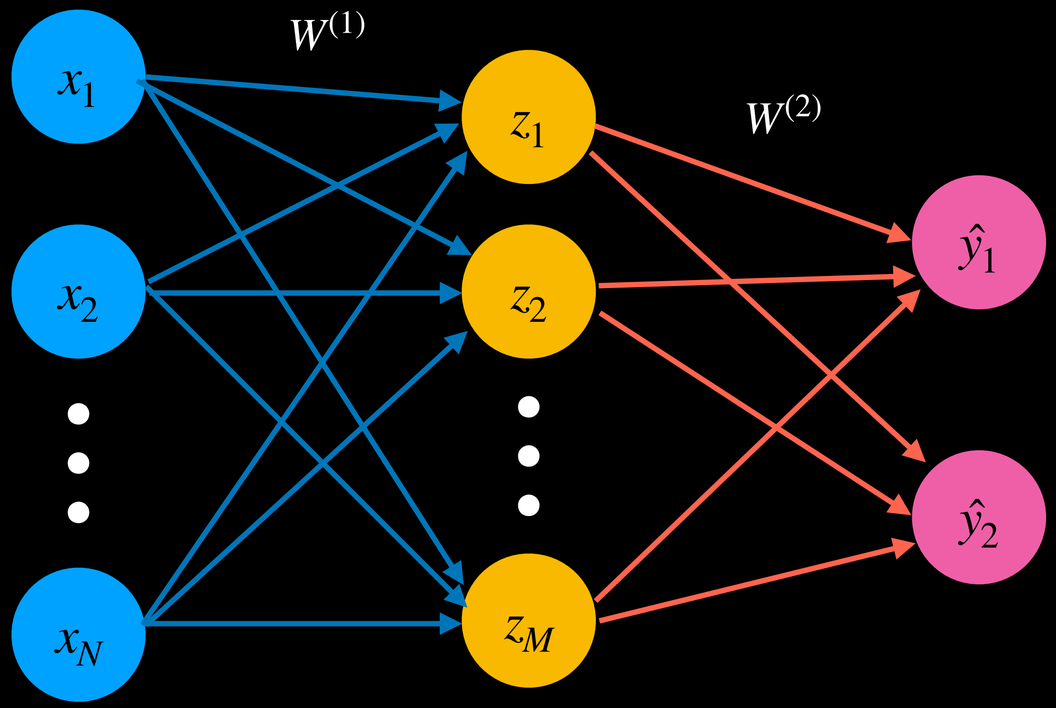


$$z_j = w_{0j} + \sum_{i=1}^{N} x_i w_{ij}$$

Neural Networks

Single Layer Neural Network

One Hidden Layer

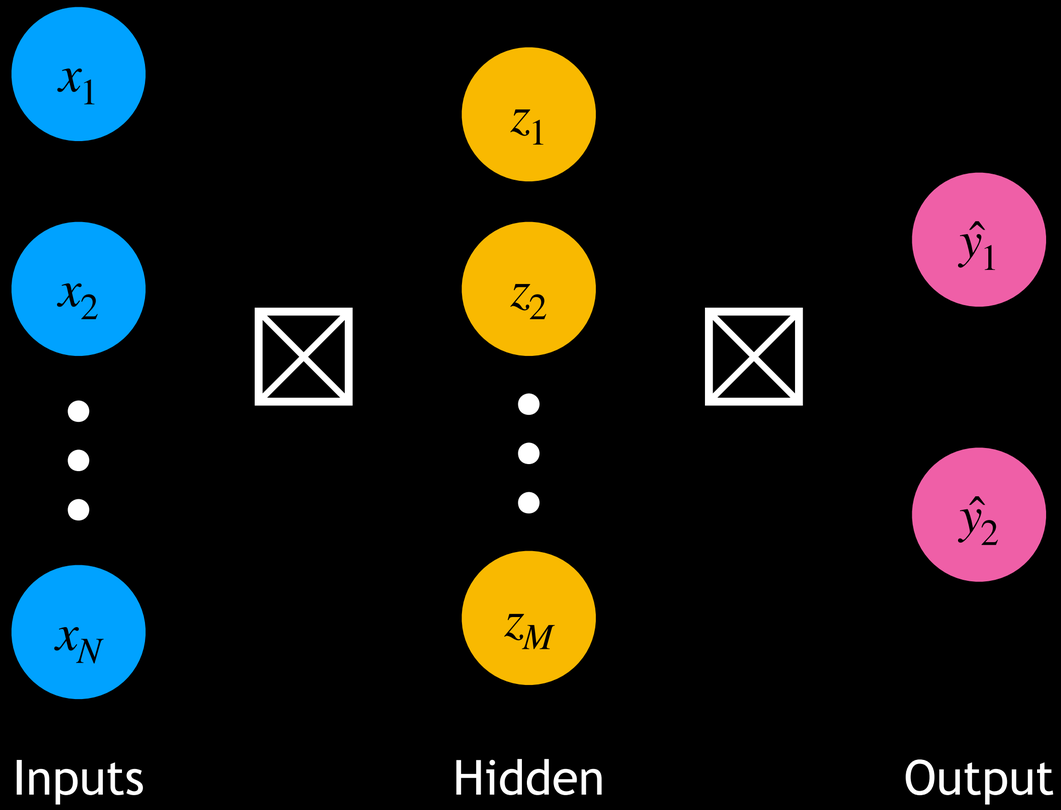


$$z_j = w_{0j}^{(1)} + \sum_{i=1}^{N} x_i w_{ij}^{(1)}$$

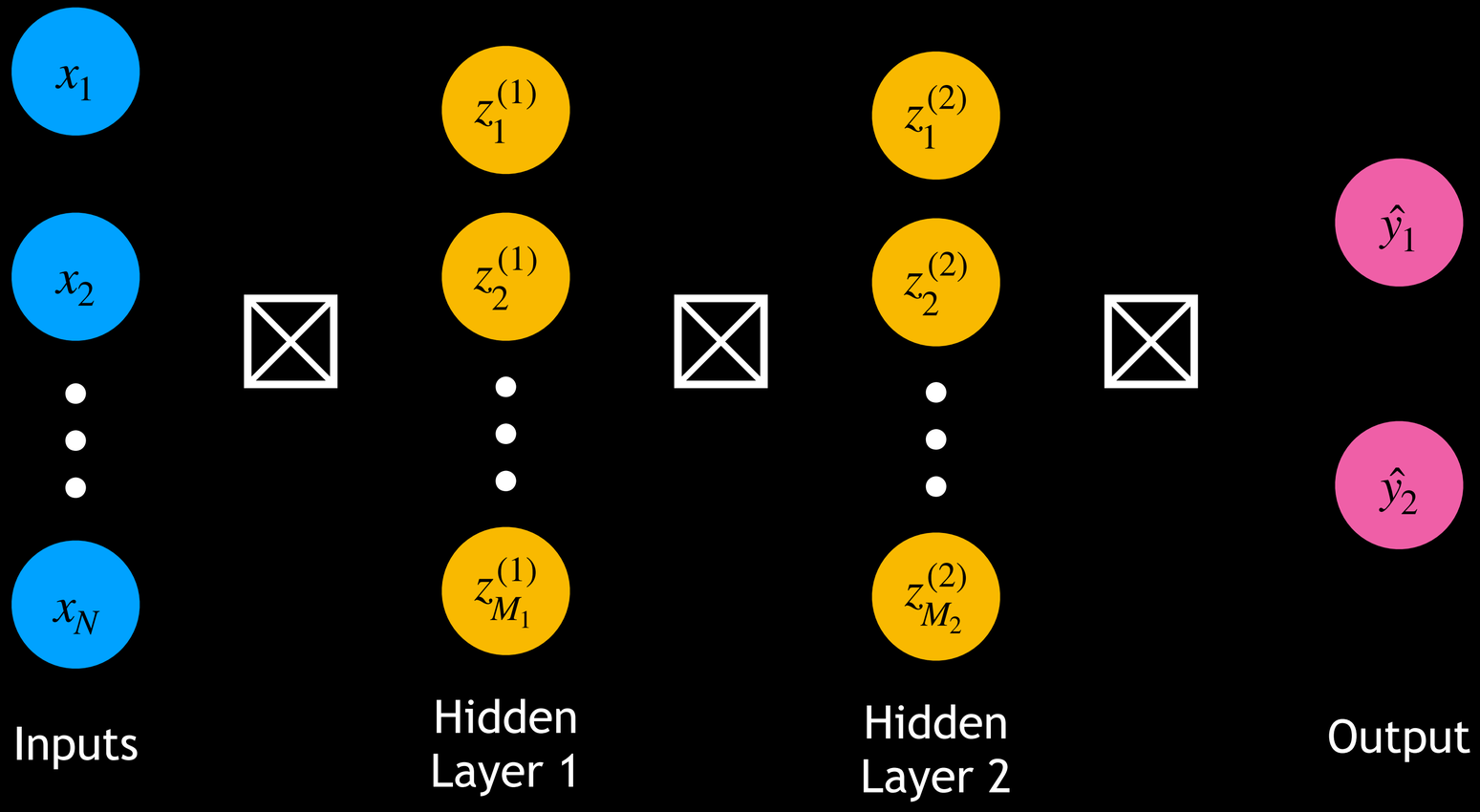
$$\hat{y}_j = f\left(w_{0j}^{(2)} + \sum_{i=1}^{M} f(z_i) w_{ij}^{(2)} \right)$$

Single Layer Neural Network

Simplified



Multi-Layer Neural Network



Training a Neutral Network

Digit Recognition Neural Network (NN)
MNIST Database

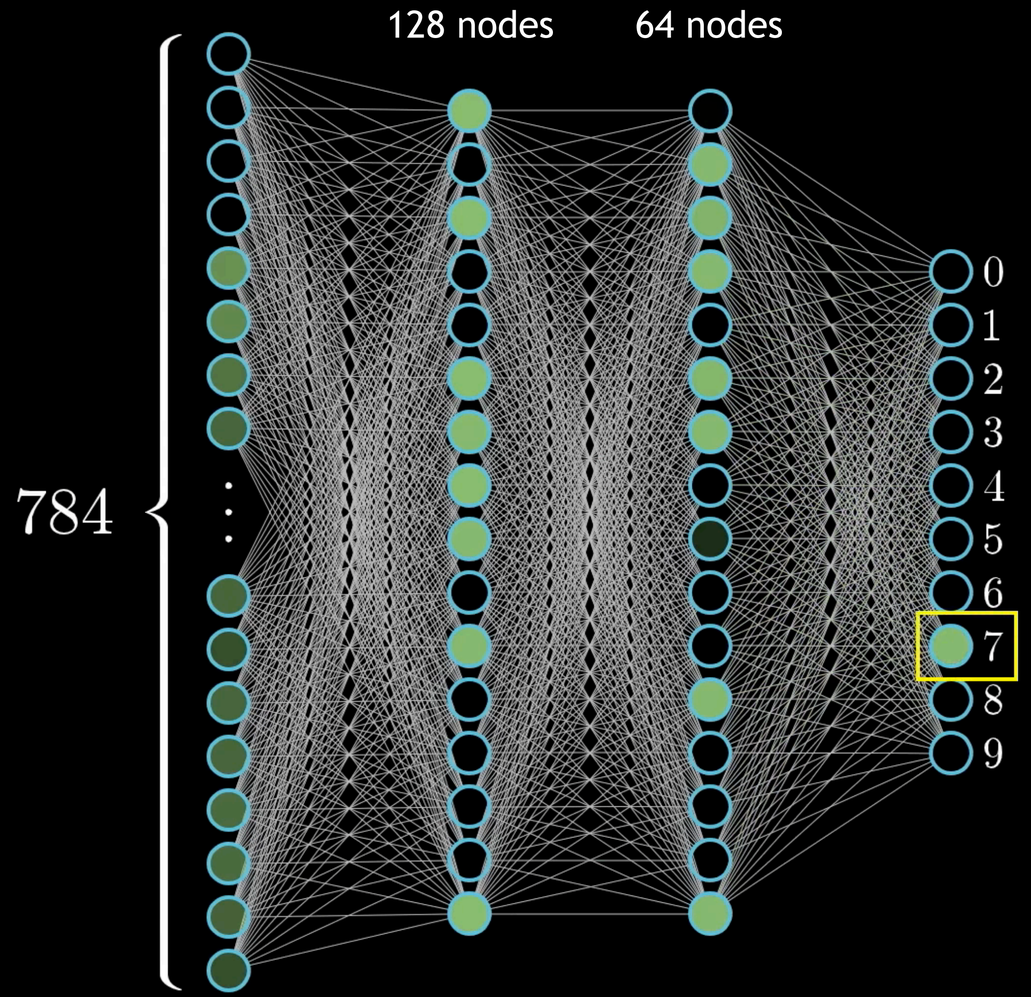
- We will use the MNIST database (it's a database of images of handwritten digits) to create and train a neural network
- This neural network should then be able to recognise the digit in an image
- We will go through two main ways of working through a neural network:
  - Forward Propagation: This is how a neural network provides the output
  - Backward Propagation: This is how we train a neural network
- In the next slide, we will start with the structure of this neural network and delve deeper from there

MNIST NN

Structure and Overview

We will create a neural network with 784 (28 x 28) input nodes, each one representing the grayscale numeric value of a pixel in the image.

28 x 28 pixels



The output layer clearly being made of 10 nodes as we need the neural network to pick between one of these digits.

We will use 2 hidden layers in this neural network. The first hidden layer comprising of 128 nodes and the second one with 64 nodes.

MNIST NN

The Intuition behind Layers and Nodes

The number of hidden layers and nodes is determined by the complexity of the decision-making we expect from the NN.

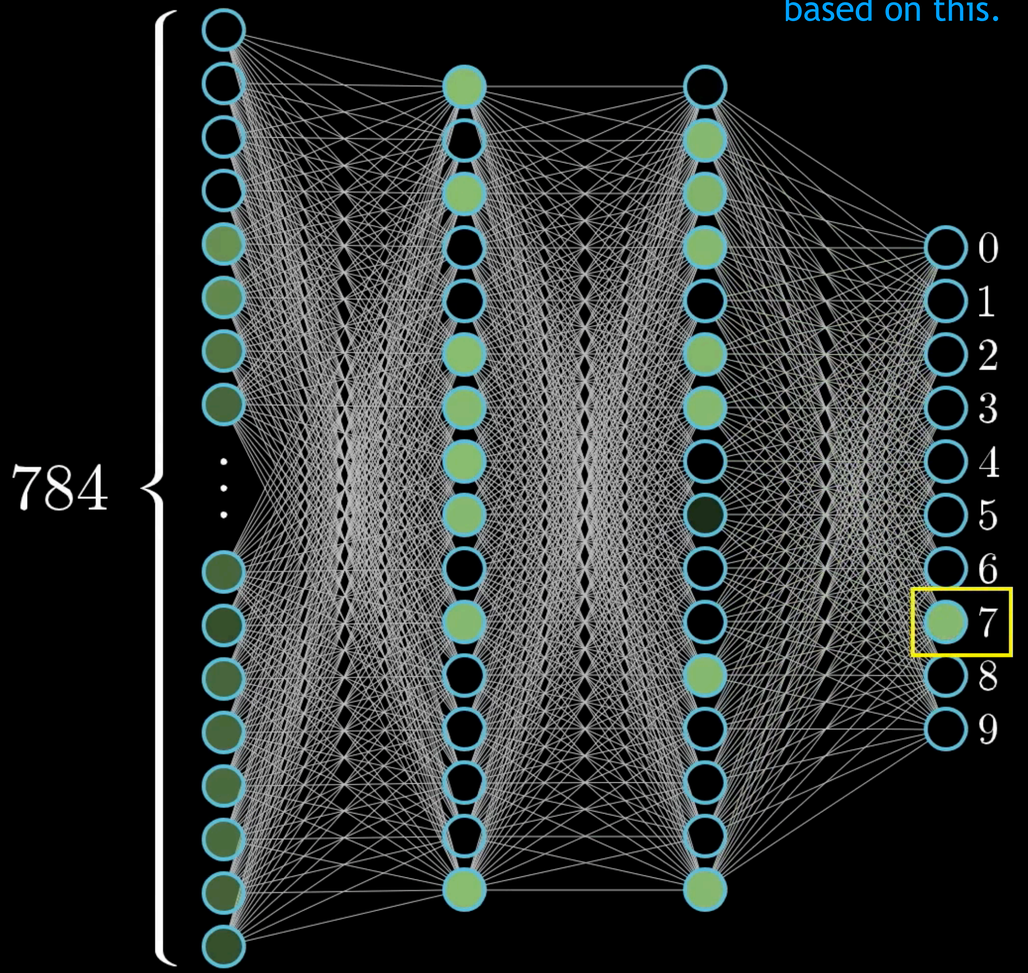
Whilst the Universal Approximation Theorem states that there always exists a NN with enough nodes to approximate a function within some tolerance, it doesn't actually tell us how to construct such a NN.

Clearly, the number of hidden layers and nodes are key parameters for a NN. Stathakis* states that the "near-optimal" number of total nodes should be $2\sqrt{(m+2)\,N}$, where m is the number of output nodes and N is the number of training samples.

In this example, we have used a total of 192 nodes (significantly less than what is suggested by Stathakis*), with 128 nodes in the first hidden layer and 64 nodes in the second.

The first layer in this NN can be thought of as identifying lines or curves in the image - with certain neurons being "activated" based on this.

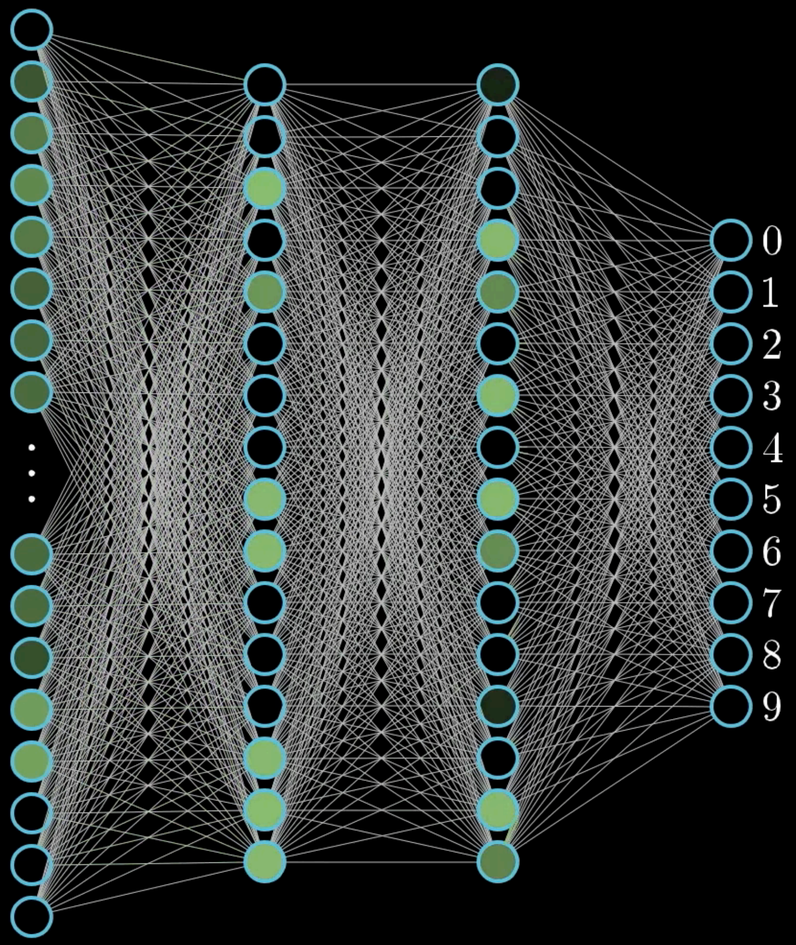
The second layer in this NN can be thought of as identifying patterns of lines or curves in the image - with certain neurons being "activated" based on this.



*http://dstath.users.uth.gr/papers/IJRS2009_Stathakis.pdf

MNIST NN

Forward Propagation



Non-Linear Activation Function for layer n

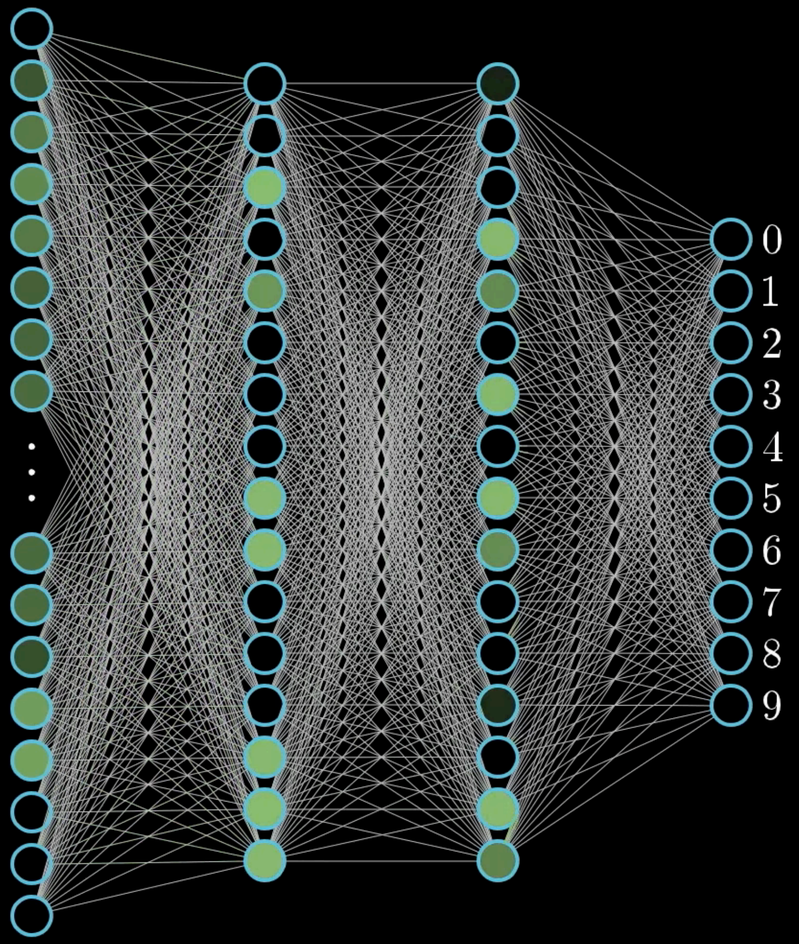
$$\mathbf{a}^{(n)} = f^{(n)}\left(\mathbf{W}'^{(n)}\, a^{(n-1)} + \mathbf{b}^{(n)}\right)$$

$$\begin{bmatrix} a_0^{(n)} \\ a_1^{(n)} \\ \vdots \\ a_m^{(n)} \end{bmatrix} \quad \begin{bmatrix} w_{00}^{(n)} & w_{01}^{(n)} & \cdots & w_{0k}^{(n)} \\ w_{10}^{(n)} & w_{11}^{(n)} & \cdots & w_{1k}^{(n)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m0}^{(n)} & w_{m1}^{(n)} & \cdots & w_{mk}^{(n)} \end{bmatrix} \quad \begin{bmatrix} a_0^{(n)} \\ a_1^{(n)} \\ \vdots \\ a_k^{(n)} \end{bmatrix} \quad \begin{bmatrix} b_0^{(n)} \\ b_1^{(n)} \\ \vdots \\ b_m^{(n)} \end{bmatrix}$$

n here denotes the n-th layer of the neural network
m is the number of nodes in the n-th layer
k is the number of nodes in the (n-1)-th layer

MNIST NN

Cost Function

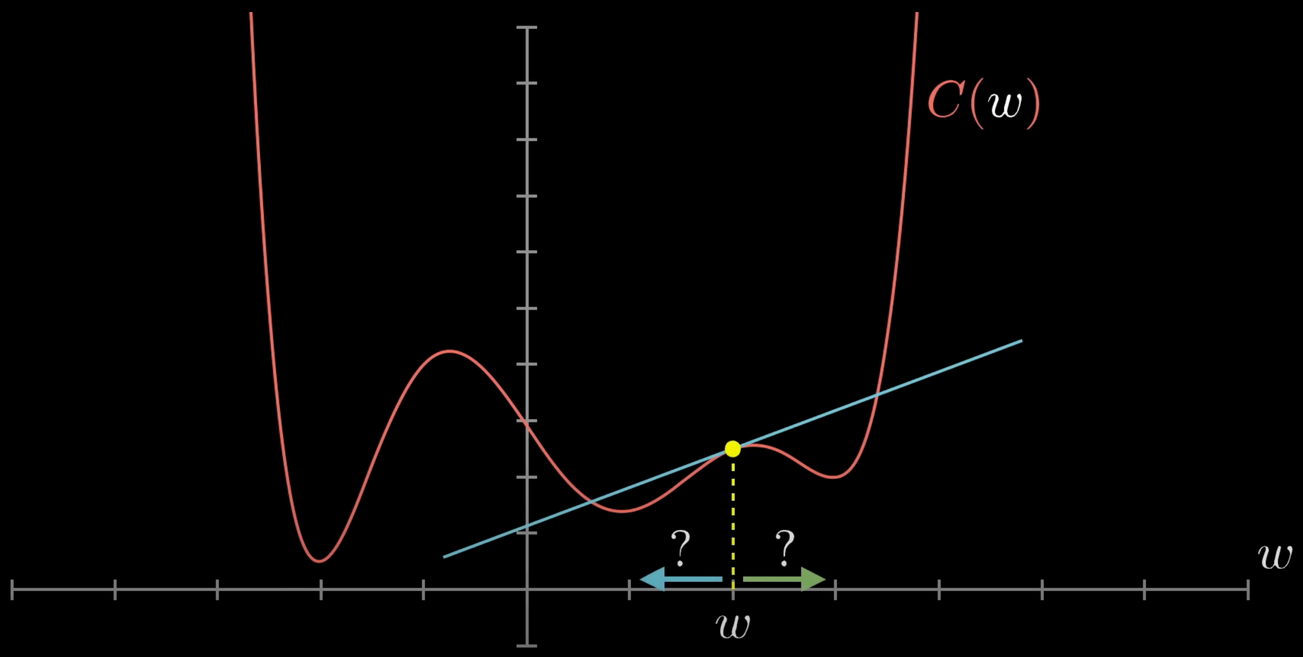


- For the neural network to work, we need to find the weights and biases which will correctly predict a digit
- This means we need to first define a cost function to minimise with respect to these parameters
- In this case, it's the squared difference of the output ($\mathbf{a}^{(3)}$) vs what is expected ($\mathbf{y}$) for each training example:

$$C\left(\mathbf{W}^{(1)}, \mathbf{W}^{(2)}, \mathbf{W}^{(3)}, \mathbf{b}^{(1)}, \mathbf{b}^{(2)}, \mathbf{b}^{(3)}\right) = \left(\mathbf{a}^{(3)} - \mathbf{y}\right)^2$$

Minimising Cost Function

Gradient Descent



$$w \leftarrow w - \alpha \frac{\partial C}{\partial w}$$

- Gradient Descent is a quick and easy way to find a *local* minima of a function
- The chart on the left shows how this works for a simple cost function with one weight
- We essentially change the weight proportionally to the negative gradient of the cost function
- The farther we are from the minima, the bigger the change in the opposite direction to the gradient

Simplified NN

Backward Propagation - Final Two Layers

$a^{(2)}$ $a^{(3)}$ y

Desired Output

Cost of one training example

$$C_0(\dots) = \left(a^{(3)} - y\right)^2$$

$$C_0 = \left(a^{(3)} - y\right)^2$$

$$z^{(3)} = w^{(3)} a^{(2)} + b^{(3)}$$

$$a^{(3)} = f\left(z^{(3)}\right)$$

$$\frac{\partial C_0}{\partial w^{(3)}} = \frac{\partial z^{(3)}}{\partial w^{(3)}} \frac{\partial a^{(3)}}{\partial z^{(3)}} \frac{\partial C_0}{\partial a^{(3)}}$$

$a^{(2)}$ $f'\left(z^{(3)}\right)$ $2\left(a^{(3)} - y\right)$

$$\frac{\partial C_0}{\partial b^{(3)}} = \frac{\partial z^{(3)}}{\partial b^{(3)}} \frac{\partial a^{(3)}}{\partial z^{(3)}} \frac{\partial C_0}{\partial a^{(3)}}$$

1 $f'\left(z^{(3)}\right)$ $2\left(a^{(3)} - y\right)$

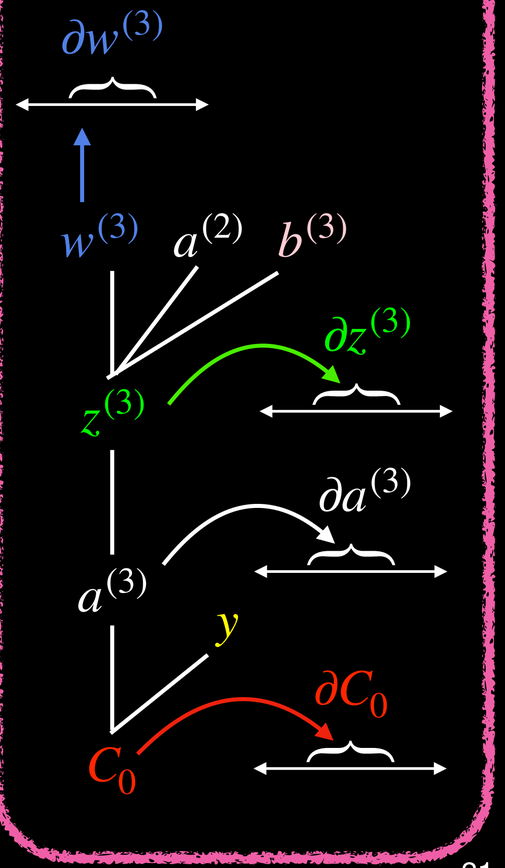
Average over all training examples

$$\frac{\partial C}{\partial w^{(3)}} = \overbrace{\frac{1}{N} \sum_{k=0}^{N-1} \frac{\partial C_k}{\partial w^{(3)}}}$$

Average over all training examples

$$\frac{\partial C}{\partial b^{(3)}} = \overbrace{\frac{1}{N} \sum_{k=0}^{N-1} \frac{\partial C_k}{\partial b^{(3)}}}$$

Chain Rule



Simplified NN

Backward Propagation - Previous Layers

$a^{(1)}$ $a^{(2)}$ $a^{(3)}$

Desired Output

y

Cost of one training example

$$C_0(\dots) = \left(a^{(3)} - y\right)^2$$

$$z^{(2)} = w^{(2)}a^{(1)} + b^{(2)}$$
$$a^{(2)} = f\left(z^{(2)}\right)$$

$$C_0 = \left(a^{(3)} - y\right)^2$$
$$z^{(3)} = w^{(3)}a^{(2)} + b^{(3)}$$
$$a^{(3)} = f\left(z^{(3)}\right)$$

$$\frac{\partial C_0}{\partial w^{(2)}} = \frac{\partial z^{(2)}}{\partial w^{(2)}} \frac{\partial a^{(2)}}{\partial z^{(2)}} \frac{\partial C_0}{\partial a^{(2)}}$$

$a^{(1)}$ $f'\left(z^{(2)}\right)$

$$\frac{\partial C_0}{\partial a^{(2)}} = \frac{\partial z^{(3)}}{\partial a^{(2)}} \frac{\partial a^{(3)}}{\partial z^{(3)}} \frac{\partial C_0}{\partial a^{(3)}}$$

$w^{(3)}$ $f'\left(z^{(3)}\right)$ $2\left(a^{(3)} - y\right)$

$$\frac{\partial C_0}{\partial b^{(2)}} = \frac{\partial z^{(2)}}{\partial b^{(2)}} \frac{\partial a^{(2)}}{\partial z^{(2)}} \frac{\partial C_0}{\partial a^{(2)}}$$

1 $f'\left(z^{(2)}\right)$

Average over all training examples

$$\frac{\partial C}{\partial w^{(2)}} = \overbrace{\frac{1}{N}\sum_{k=0}^{N-1} \frac{\partial C_k}{\partial w^{(2)}}}$$

Average over all training examples

$$\frac{\partial C}{\partial b^{(2)}} = \overbrace{\frac{1}{N}\sum_{k=0}^{N-1} \frac{\partial C_k}{\partial b^{(2)}}}$$

Simplified NN to MNIST NN

Backward Propagation

$a^{(2)}$ $a^{(3)}$

$$C_0 = \left(a^{(3)} - y\right)^2$$

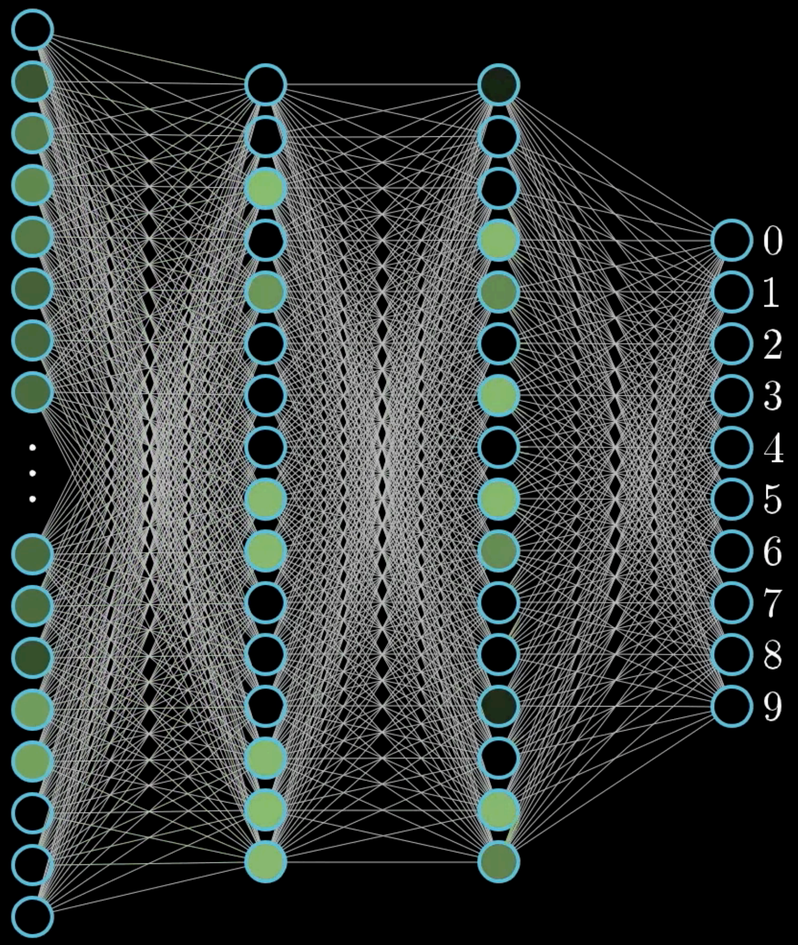
$$z^{(3)} = w^{(3)} a^{(2)} + b^{(3)}$$

$$a^{(3)} = f\left(z^{(3)}\right)$$

$$\frac{\partial C_0}{\partial w^{(3)}} = \frac{\partial z^{(3)}}{\partial w^{(3)}} \frac{\partial a^{(3)}}{\partial z^{(3)}} \frac{\partial C_0}{\partial a^{(3)}}$$

$$\frac{\partial C_0}{\partial b^{(3)}} = \frac{\partial z^{(3)}}{\partial b^{(3)}} \frac{\partial a^{(3)}}{\partial z^{(3)}} \frac{\partial C_0}{\partial a^{(3)}}$$

$$\frac{\partial C_0}{\partial a^{(2)}} = \frac{\partial z^{(3)}}{\partial a^{(2)}} \frac{\partial a^{(3)}}{\partial z^{(3)}} \frac{\partial C_0}{\partial a^{(3)}}$$



$$C_0 = \sum_{j=0}^{n_3-1} \left(a_j^{(3)} - y_j\right)^2$$

$$z_j^{(3)} = \sum_{k=0}^{n_2-1} w_{jk}^{(3)} a_k^{(2)} + b_j^{(3)}$$

$$a_j^{(3)} = f\left(z_j^{(3)}\right)$$

$$\frac{\partial C_0}{\partial w_{jk}^{(3)}} = \frac{\partial z_j^{(3)}}{\partial w_{jk}^{(3)}} \frac{\partial a_j^{(3)}}{\partial z_j^{(3)}} \frac{\partial C_0}{\partial a_j^{(3)}}$$

$$\frac{\partial C_0}{\partial b_j^{(3)}} = \frac{\partial z_j^{(3)}}{\partial b_j^{(3)}} \frac{\partial a_j^{(3)}}{\partial z_j^{(3)}} \frac{\partial C_0}{\partial a_j^{(3)}}$$

$$\frac{\partial C_0}{\partial a_k^{(2)}} = \sum_{j=0}^{n_3-1} \frac{\partial z_j^{(3)}}{\partial a_k^{(2)}} \frac{\partial a_j^{(3)}}{\partial z_j^{(3)}} \frac{\partial C_0}{\partial a_j^{(3)}}$$

MNIST NN

Implementation - Initialise the NN (No Bias Terms)

```python
# For the neural network, we need 784 inputs and two hidden layers
# Hidden layer 1 will have 128 nodes
# Hidden layer 2 will have 64 nodes

class DNN:

    def __init__(self, sizes, epochs, lr):
        self.sizes = sizes
        self.epochs = epochs
        self.lr = lr
        input_layer = sizes[0]
        hidden_1 = sizes[1]
        hidden_2 = sizes[2]
        output_layer = sizes[3]
        self.params = {
            'W1': np.random.randn(hidden_1, input_layer) * np.sqrt(1./hidden_1),       # 128 x 784
            'W2': np.random.randn(hidden_2, hidden_1) * np.sqrt(1./hidden_2),          # 64 x 128
            'W3': np.random.randn(output_layer, hidden_2) * np.sqrt(1./output_layer)   # 10 x 64
        }
```

MNIST NN

Implementation - Define Activation Functions

```python
def activation_function(self, x, func_type, derivative=False):
    if derivative:
        if func_type == 'sigmoid':
            return (np.exp(-x)) / ((np.exp(-x) + 1) ** 2)
        elif func_type == 'tanh':
            return 1 - (((np.exp(x) - np.exp(-x)) ** 2) / ((np.exp(x) + np.exp(-x)) ** 2))
        elif func_type == 'softmax':
            exps = np.exp(x-x.max())
            return exps / np.sum(exps, axis=0) * (1 - exps / np.sum(exps, axis=0))
        elif func_type == 'relu':
            return (x > 0) * 1
        else:
            print('Unrecognised activation function!')
    else:
        if func_type == 'sigmoid':
            return 1 / (1 + np.exp(-x))
        elif func_type == 'tanh':
            return (np.exp(x) - np.exp(-x)) / (np.exp(x) + np.exp(-x))
        elif func_type == 'softmax':
            exps = np.exp(x-x.max())
            return exps / np.sum(exps, axis=0)
        elif func_type == 'relu':
            return np.maximum(0.0, x)
        else:
            print('Unrecognised activation function!')
```

We have the option to use any of these activation functions for our neural network

It allows us to use a different activation function for each layer

MNIST NN

Implementation - Forward & Backward Propagation

```python
def forward_pass(self, x_train, func_type):
    params = self.params
    params['A0'] = x_train                              # 784 x 1
    # input_layer to hidden_1
    params['Z1'] = np.dot(params['W1'], params['A0'])   # 128 x 1
    params['A1'] = self.activation_function(params['Z1'], func_type)
    # hidden_1 to hidden_2
    params['Z2'] = np.dot(params['W2'], params['A1'])   # 64 x 1
    params['A2'] = self.activation_function(params['Z2'], func_type)
    # hidden_2 to output_layer
    params['Z3'] = np.dot(params['W3'], params['A2'])   # 10 x 1
    params['A3'] = self.activation_function(params['Z3'], 'softmax')
    return params['Z3']

def backward_pass(self, y_train, output, func_type):
    params = self.params
    change_w = {}
    # calculate W3 update
    error = 2 * (output - y_train) / output.shape[0] * self.activation_function(params['Z3'],
    'softmax', derivative=True)
    change_w['W3'] = np.outer(error, params['A2'])
    # calculate W2 update
    error = np.dot(params['W3'].T, error) * self.activation_function(params['Z2'], func_type,
    derivative=True)
    change_w['W2'] = np.outer(error, params['A1'])
    # calculate W1 update
    error = np.dot(params['W2'].T, error) * self.activation_function(params['Z1'], func_type,
    derivative=True)
    change_w['W1'] = np.outer(error, params['A0'])
    return change_w
```

We have used the softmax activation function for the final layer, which lends itself to the multi-classification scenario

For the previous layers, we can use sigmoid, tanh or relu - depending on the use case and the learning rate

MNIST NN

Implementation - Train! Using 60k Images…

```python
def train(self, train_list, test_list, func_type):
    output_data = []
    for i in range(self.epochs):
        start_time = time.time()
        for x in train_list:
            values = x.split(",")
            inputs = (np.asfarray(values[1:]) / 255.0 * 0.99) + 0.01
            targets = np.zeros(10) + 0.01
            targets[int(values[0])] = 0.99
            output = self.forward_pass(inputs, func_type)
            change_w = self.backward_pass(targets, output, func_type)
            self.update_weights(change_w)
        accuracy = self.compute_accuracy(test_list, func_type)
        print('Epoch: {0}, Time Spent: {1:.02f}s, Accuracy: {2:.2f}%, Learning Rate: {3}, Activation
        Function: {4}'.format(i + 1, time.time() - start_time, accuracy * 100, self.lr, func_type))
        output_data.append([i + 1, time.time() - start_time, accuracy * 100, self.lr, func_type])
    return output_data
```

✓ 0.1s Python

```python
dnn = DNN(sizes=[784, 128, 64, 10], epochs=10, lr=1)
output_data = dnn.train(train_list, test_list, 'tanh')
```

✓ 11m 2.8s Python

```
Epoch: 1, Time Spent: 72.23s, Accuracy: 92.74%, Learning Rate: 1, Activation Function: tanh
Epoch: 2, Time Spent: 69.05s, Accuracy: 95.78%, Learning Rate: 1, Activation Function: tanh
Epoch: 3, Time Spent: 73.28s, Accuracy: 96.48%, Learning Rate: 1, Activation Function: tanh
Epoch: 4, Time Spent: 63.49s, Accuracy: 96.51%, Learning Rate: 1, Activation Function: tanh
Epoch: 5, Time Spent: 61.30s, Accuracy: 96.76%, Learning Rate: 1, Activation Function: tanh
Epoch: 6, Time Spent: 60.86s, Accuracy: 96.96%, Learning Rate: 1, Activation Function: tanh
Epoch: 7, Time Spent: 64.80s, Accuracy: 96.98%, Learning Rate: 1, Activation Function: tanh
Epoch: 8, Time Spent: 62.96s, Accuracy: 97.14%, Learning Rate: 1, Activation Function: tanh
Epoch: 9, Time Spent: 62.10s, Accuracy: 97.16%, Learning Rate: 1, Activation Function: tanh
Epoch: 10, Time Spent: 72.69s, Accuracy: 97.17%, Learning Rate: 1, Activation Function: tanh
```

Epoch here refers to one trip of forward and backward propogation for an entire training sample

The learning rate is a measure of how much you adjust the weights based on gradient descent - a higher value is not necessarily a better approach as you can overshoot

There is a relationship between the learning rate, the initial accuracy and the activation function

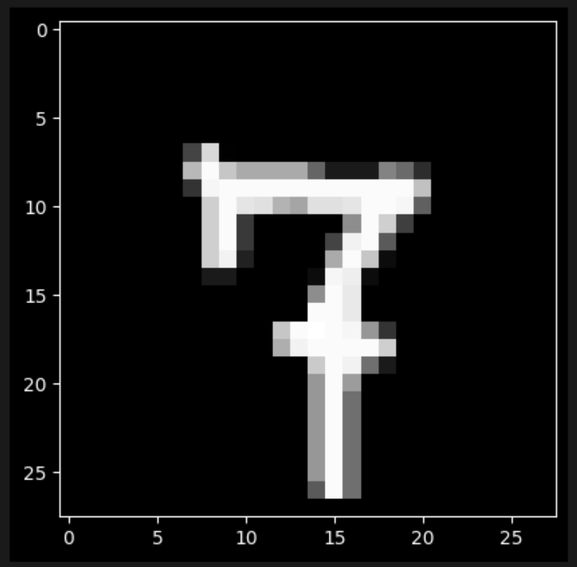
MNIST NN

Implementation - Test the Trained NN

```
# Test random image
idx = 987
values = test_list[idx].split(",")
image_array = np.asfarray(values[1:]).reshape(28,28)
matplotlib.pyplot.style.use('dark_background')
matplotlib.pyplot.imshow(image_array, cmap="Greys_r", interpolation="None")
```

✓ 0.1s

```
<matplotlib.image.AxesImage at 0x1178b2a90>
```



```
dnn.predict_image(test_list[idx], 'tanh')
```

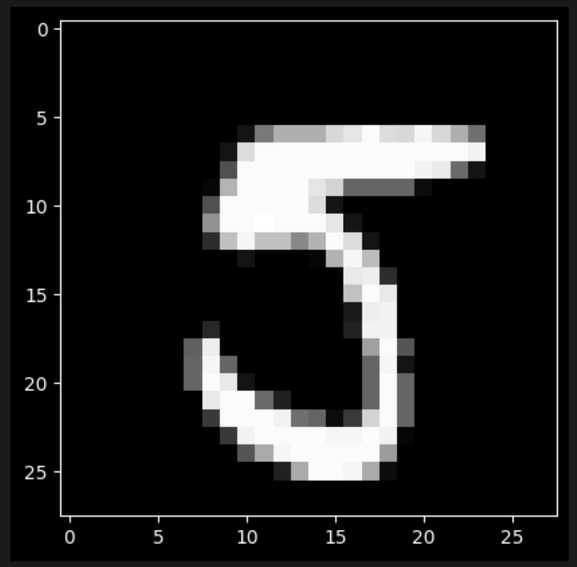
✓ 0.0s

```
7
```

```
# Test random image
idx = 570
values = test_list[idx].split(",")
image_array = np.asfarray(values[1:]).reshape(28,28)
matplotlib.pyplot.style.use('dark_background')
matplotlib.pyplot.imshow(image_array, cmap="Greys_r", interpolation="None")
```

✓ 0.6s

```
<matplotlib.image.AxesImage at 0x117790310>
```



```
dnn.predict_image(test_list[idx], 'tanh')
```

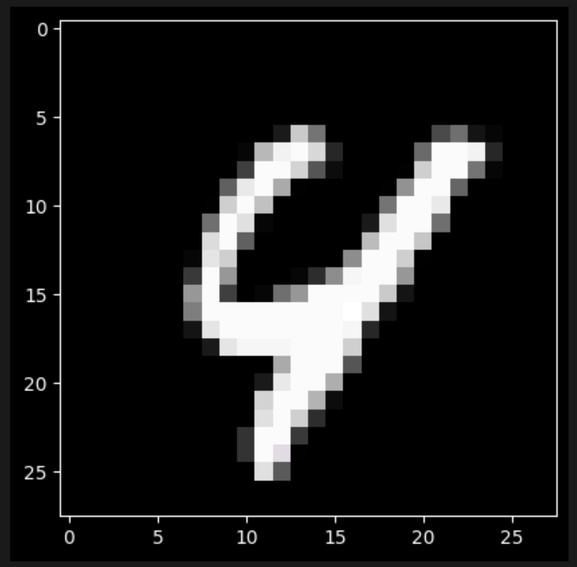
✓ 0.0s

```
5
```

```
# Test random image
idx = 139
values = test_list[idx].split(",")
image_array = np.asfarray(values[1:]).reshape(28,28)
matplotlib.pyplot.style.use('dark_background')
matplotlib.pyplot.imshow(image_array, cmap="Greys_r", interpolation="None")
```

✓ 0.2s

```
<matplotlib.image.AxesImage at 0x117ddaf10>
```



```
dnn.predict_image(test_list[idx], 'tanh')
```

✓ 0.0s

```
4
```

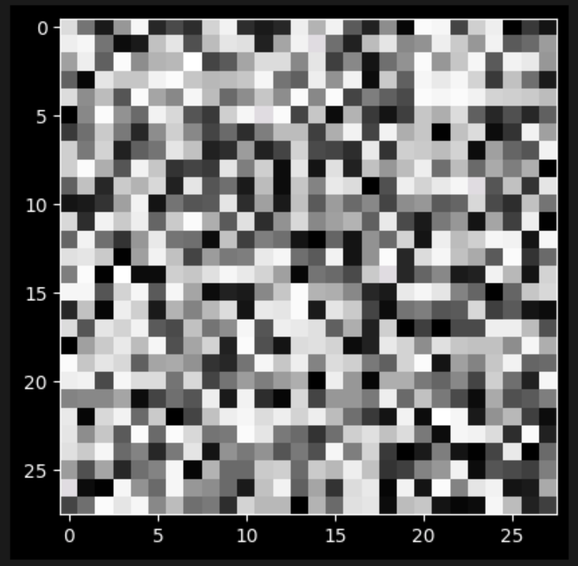
MNIST NN

A neural network's arrogance knows no bounds…

```
# Create gibberish digit
rubbish_vector = np.random.randint(0, 256, (785))
rubbish_matrix = rubbish_vector[1:].reshape(28,28)
matplotlib.pyplot.style.use('dark_background')
matplotlib.pyplot.imshow(rubbish_matrix, cmap="Greys_r", interpolation="None")
```

✓ 0.1s

```
<matplotlib.image.AxesImage at 0x1177e8310>
```




```
rubbish_str = ','.join(map(str, rubbish_vector))
dnn.predict_image(rubbish_str, 'tanh')
```

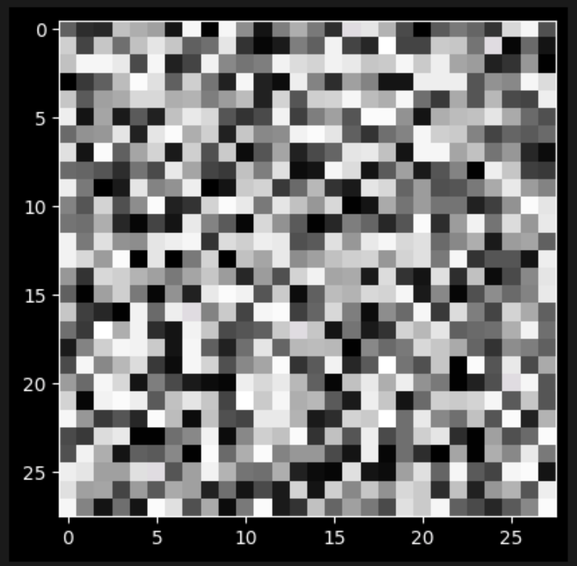
✓ 0.0s

```
8
```

```
# Create gibberish digit
rubbish_vector = np.random.randint(0, 256, (785))
rubbish_matrix = rubbish_vector[1:].reshape(28,28)
matplotlib.pyplot.style.use('dark_background')
matplotlib.pyplot.imshow(rubbish_matrix, cmap="Greys_r", interpolation="None")
```

✓ 0.4s

```
<matplotlib.image.AxesImage at 0x117e39990>
```

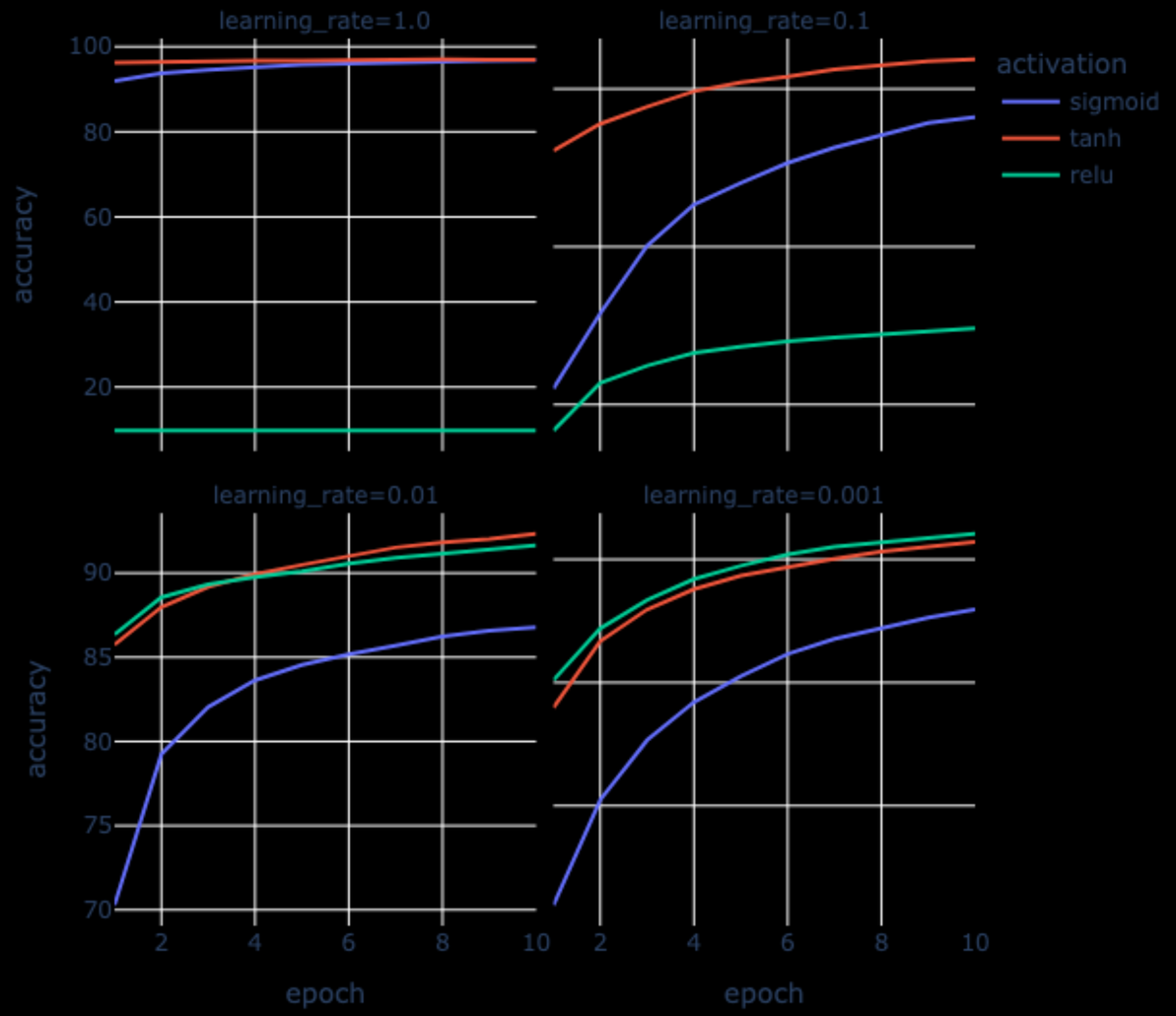


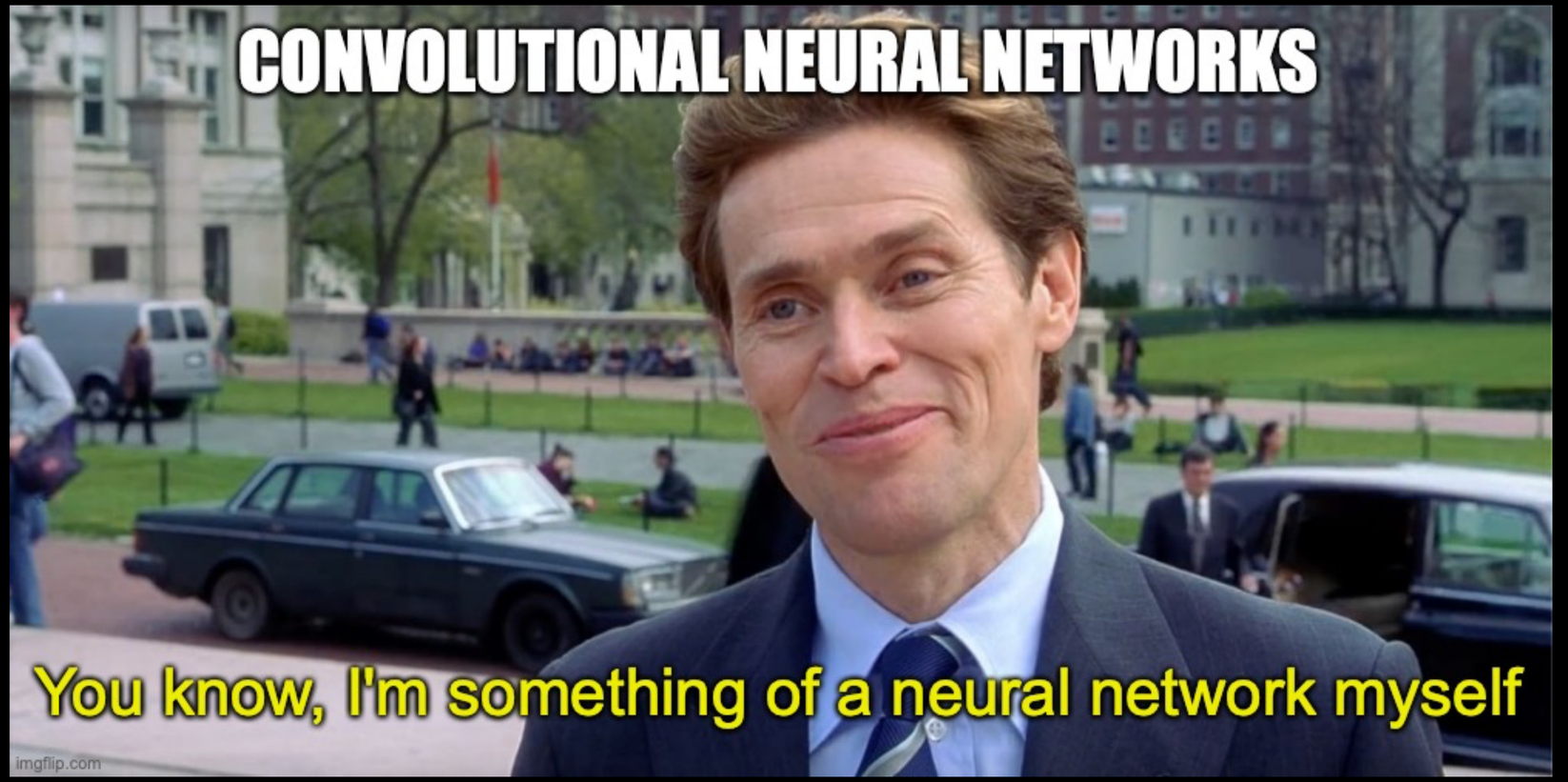

```
rubbish_str = ','.join(map(str, rubbish_vector))
dnn.predict_image(rubbish_str, 'tanh')
```

✓ 0.0s

```
5
```

MNIST NN - Learning Rate & Activation Function




CONVOLUTIONAL NEURAL NETWORKS
You know, I'm something of a neural network myself
imgflip.com

Convolutional Neural Networks (CNN)

Why do we need them?

- In the previous chapter we discussed an artificial neural network which was able to identify a digit from a grayscale image with a high level of accuracy
- Whilst this is a very useful tool already, using this type of neural network can be challenging when dealing with real-life or coloured images
- This is because we now have 3 numerical values (RGB) describing each pixel and each image is usually made up of millions (1000 x 1000) pixels
- Building and training a neural network with these many inputs ends up with billions of parameters - something we want to avoid
- This is where the use of convolution and pooling comes in handy

What is a Convolution

Discrete Convolution

$$(a \circledast b)_n = \sum_{\substack{i,\, j \\ i+j=n}} a_i \, b_j$$

$a = [1, 2, 3]$

$b = [4, 5, 6]$

$a \circledast b = [4, 13, 28, 27, 18]$

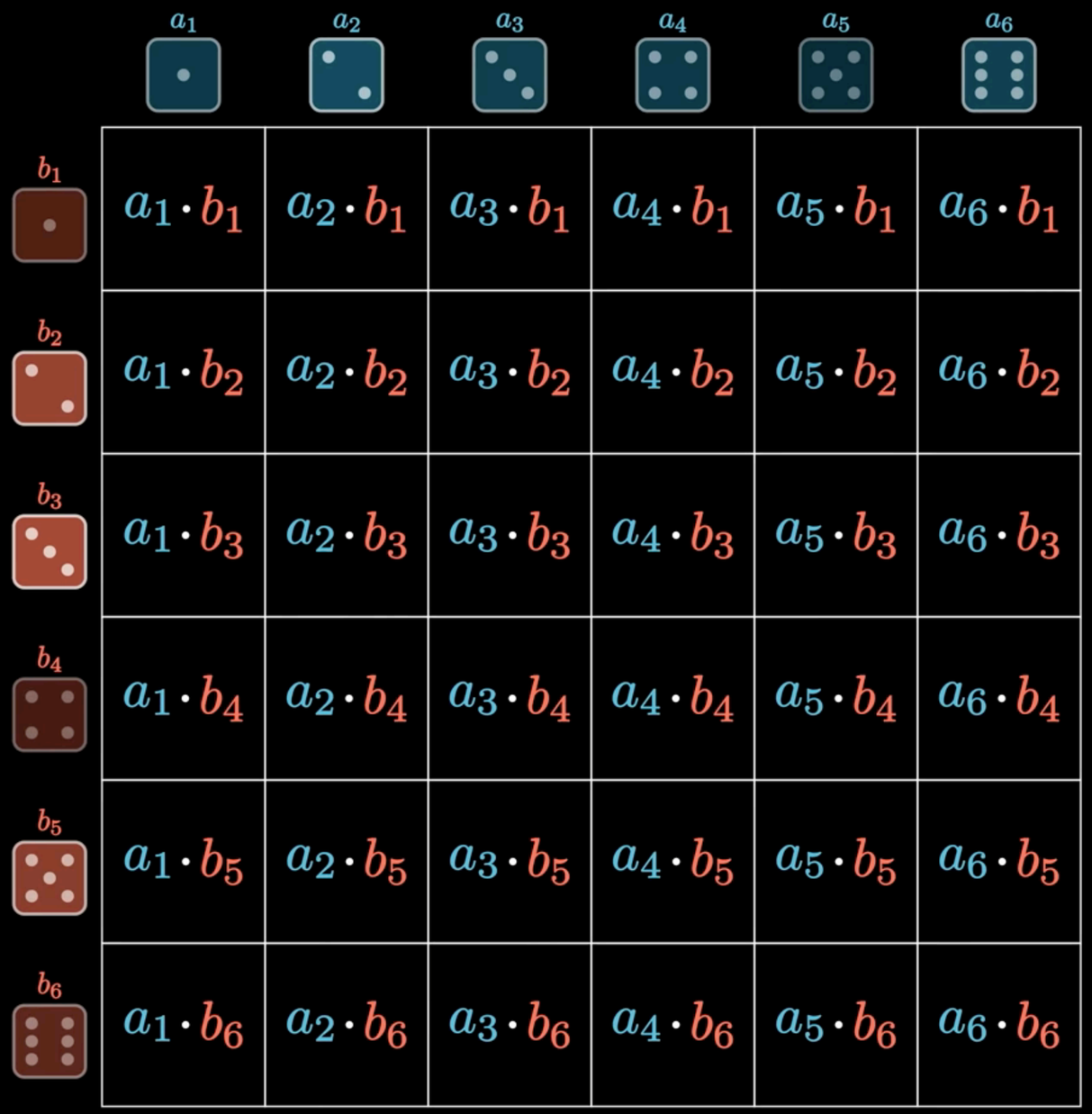
$1(4)$

$1(5) + 2(4)$

$1(6) + 2(5) + 3(4)$

$1(5) + 2(4)$

$3(6)$



What is a Convolution

A link to Fast Fourier Transforms

$$\mathbf{a} = [a_0, a_1, a_2, \ldots, a_{n-1}] \xrightarrow{\text{FFT}} \hat{\mathbf{a}} = [\hat{a}_0, \hat{a}_1, \hat{a}_2, \ldots, \hat{a}_{m+n-1}]$$

$$f(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1}$$

$$\mathbf{b} = [b_0, b_1, b_2, \ldots, b_{m-1}] \xrightarrow{\text{FFT}} \hat{\mathbf{b}} = [\hat{b}_0, \hat{b}_1, \hat{b}_2, \ldots, \hat{b}_{m+n-1}]$$

$$g(x) = b_0 + b_1 x + b_2 x^2 + \cdots + b_{m-1} x^{m-1}$$

$$\hat{\mathbf{a}} \cdot \hat{\mathbf{b}} = [\hat{a}_0 \hat{b}_0, \hat{a}_1 \hat{b}_1, \hat{a}_2 \hat{b}_2, \ldots,] \xrightarrow{\text{Inverse FFT}} \mathbf{a} * \mathbf{b} = \begin{bmatrix} a_0 b_0, \\ a_0 b_1 + a_1 b_0, \\ a_0 b_2 + a_1 b_1 + a_2 b_0, \\ \vdots \\ a_{n-1} b_{m-1} \end{bmatrix}$$

Multiply (pointwise)

Comparing Convolution with FFT

```
import numpy as np
import scipy.signal as sig
a = np.random.random(100000)
b = np.random.random(100000)
```

[2] ✓ 0.0s

```
%%timeit
np.convolve(a,b)
```

[3] ✓ 10.8s

```
1.34 s ± 66.2 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
%%timeit
sig.fftconvolve(a,b)
```

[4] ✓ 7.7s

```
9.53 ms ± 225 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

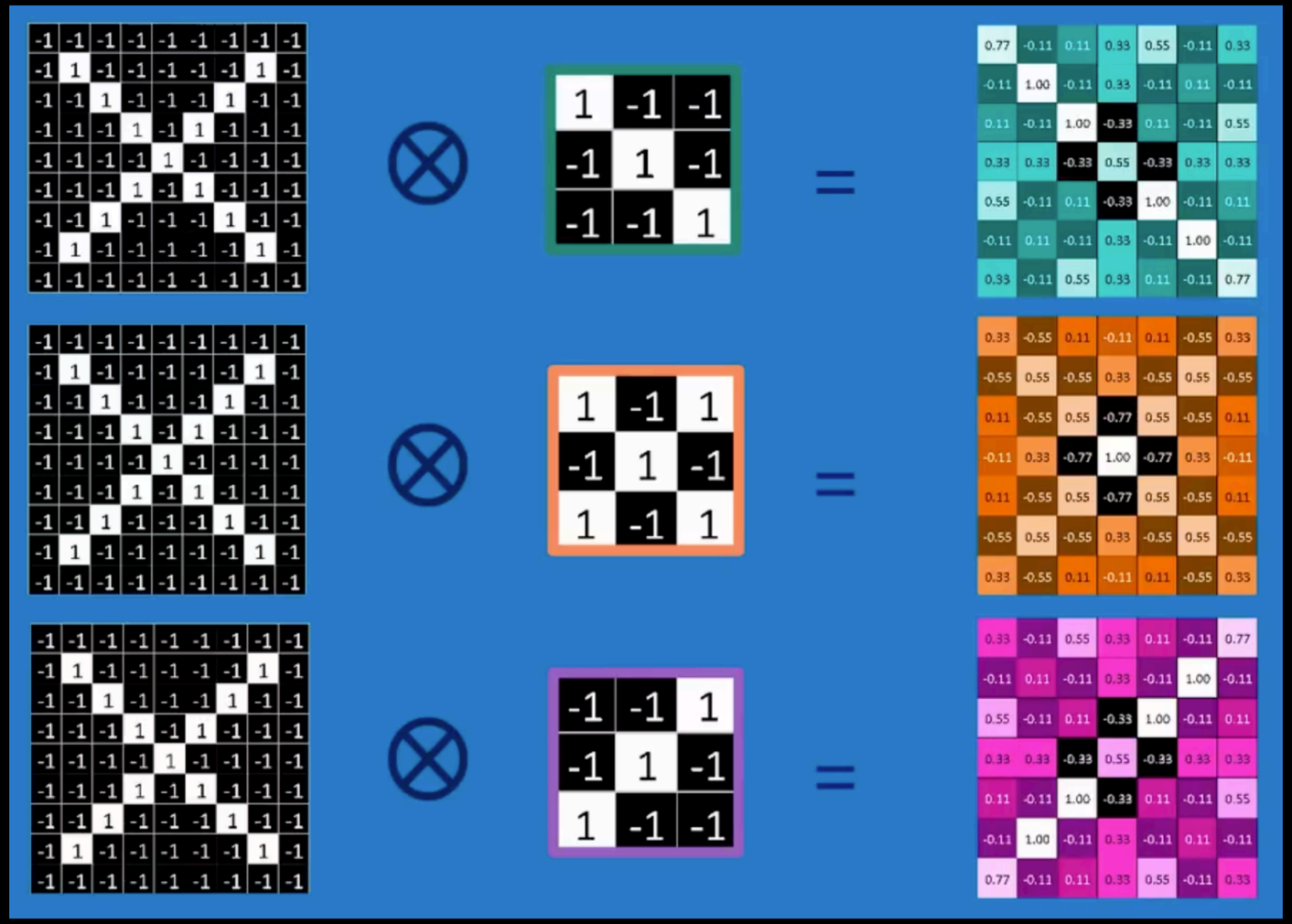
```
a = [1,2,3]
b = [4,5,6]
c = np.convolve(a,b)
d = sig.fftconvolve(a,b)
print(c, d)
```

✓ 0.0s

```
[ 4 13 28 27 18] [ 4. 13. 28. 27. 18.]
```

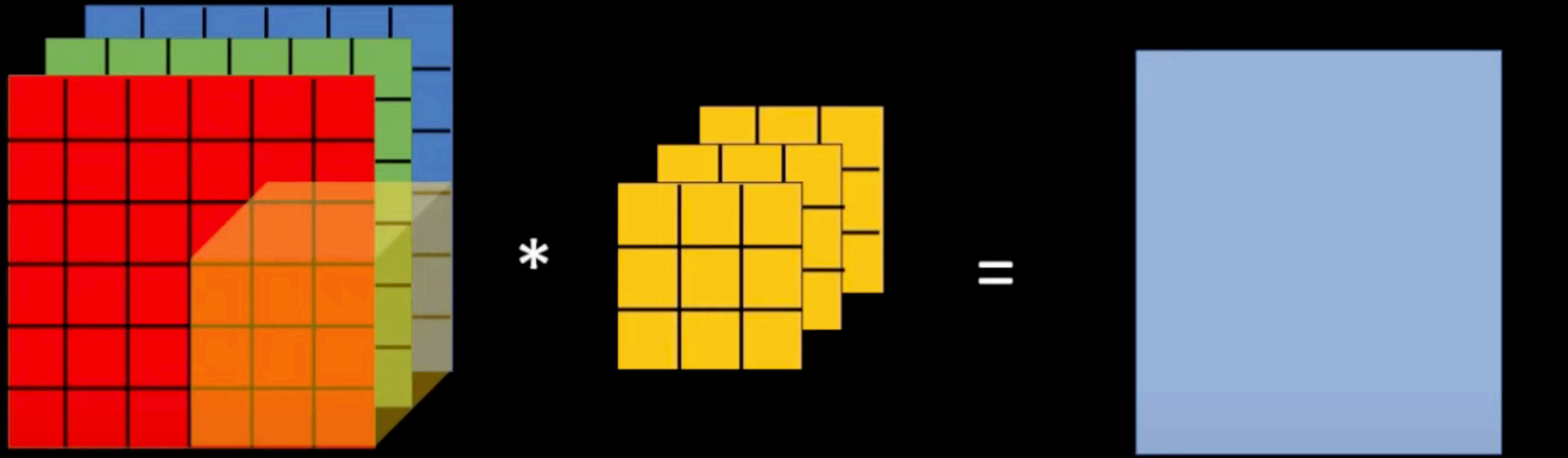
Convolution in Image Processing

Feature Recognition



Convolution in Image Processing

Applying a Filter of size (f x f) with Stride of 1



n x n x 3 * f x f x 3 = (n-f+1) x (n-f+1) x 1

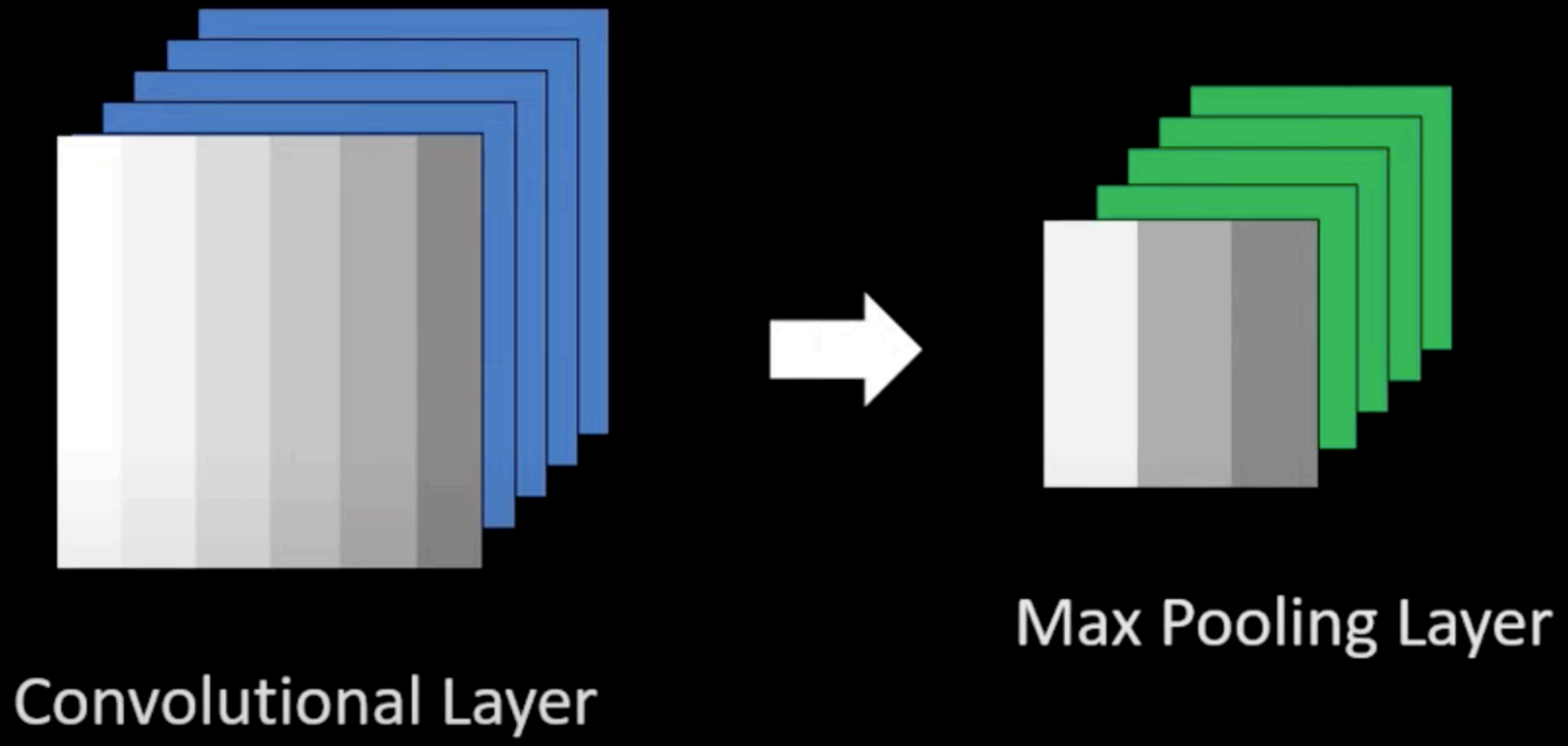
Max Pooling in Image Processing

Reducing Dimensionality and Enhancing Features

Filter of size 2x2

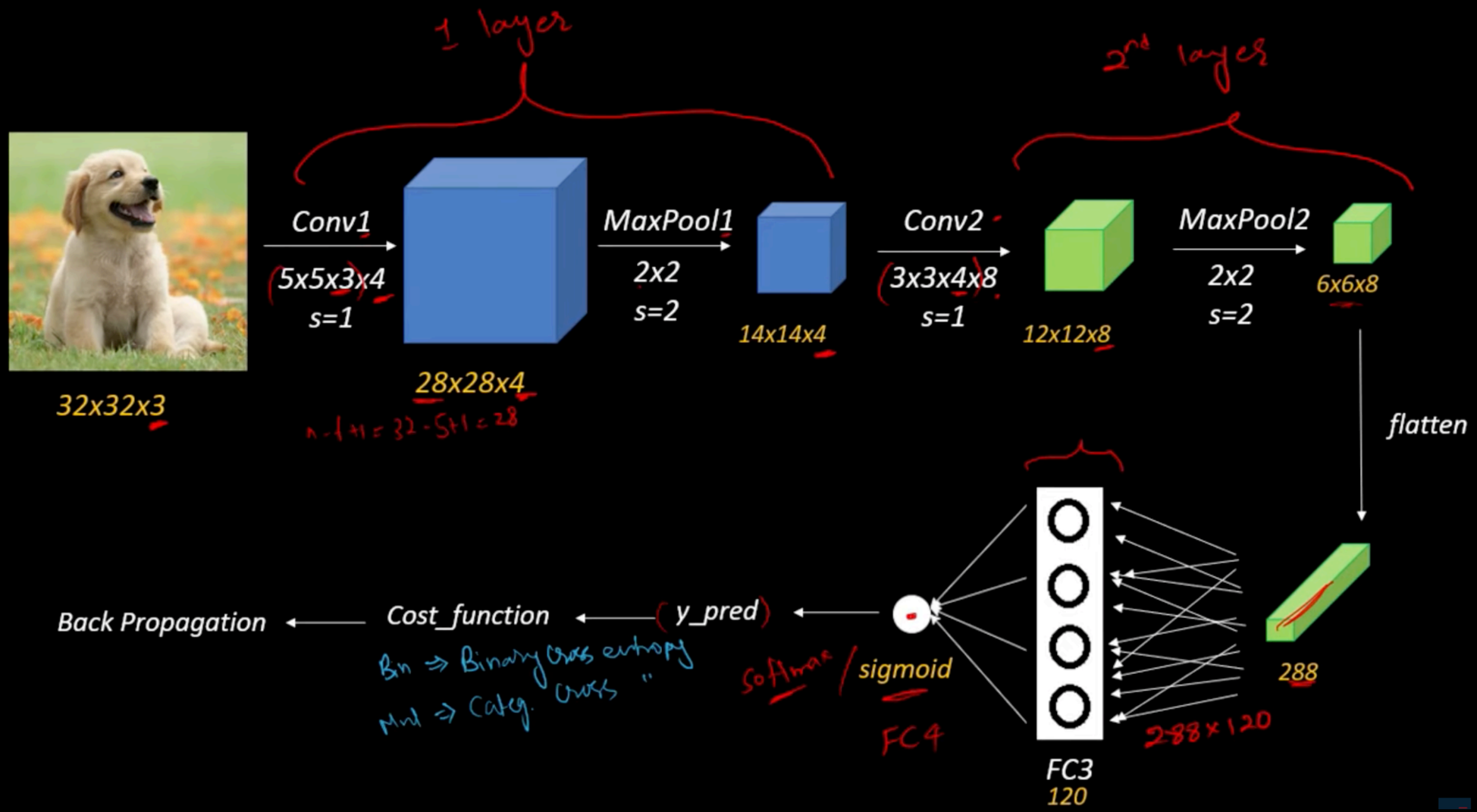
Stride = 2

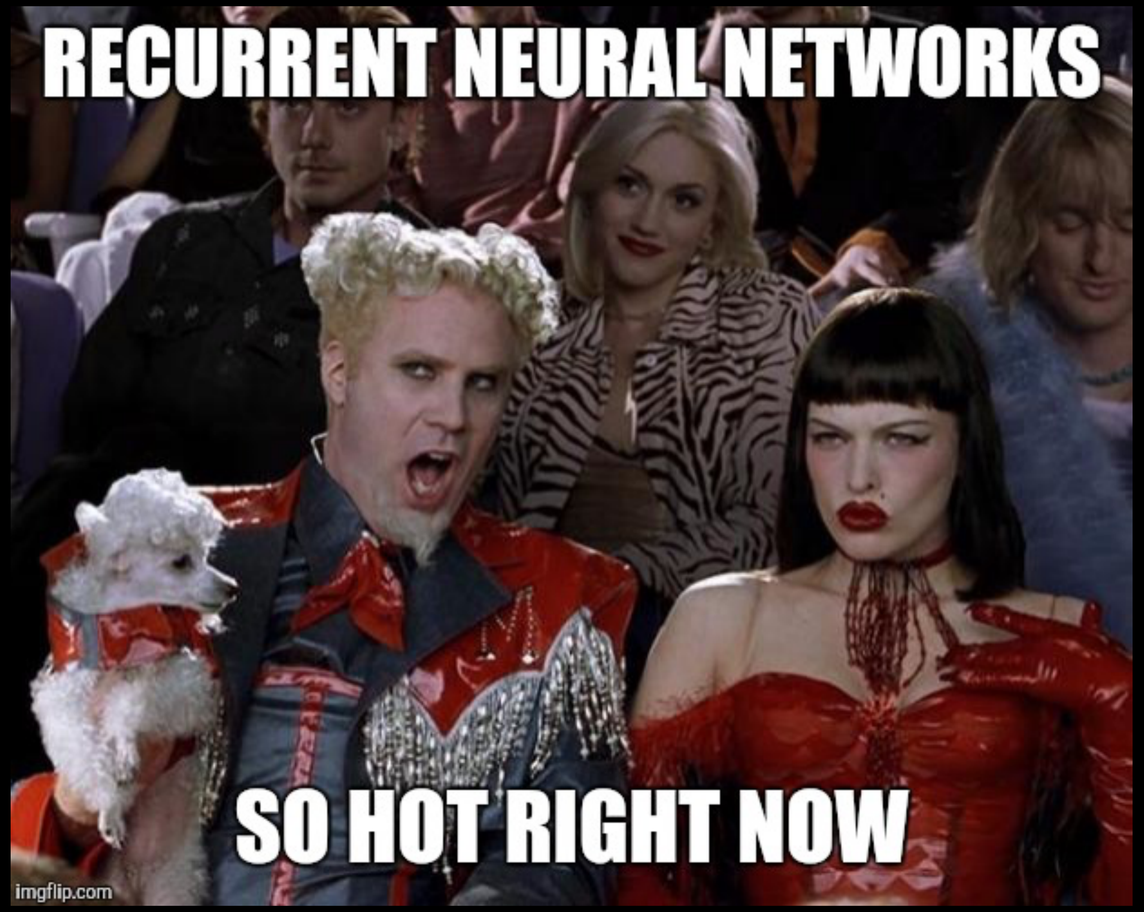
Convolutional Layer & Max Pooling Layer



Multi-Layer CNN

Example





Siri

Alexa

Google Translate

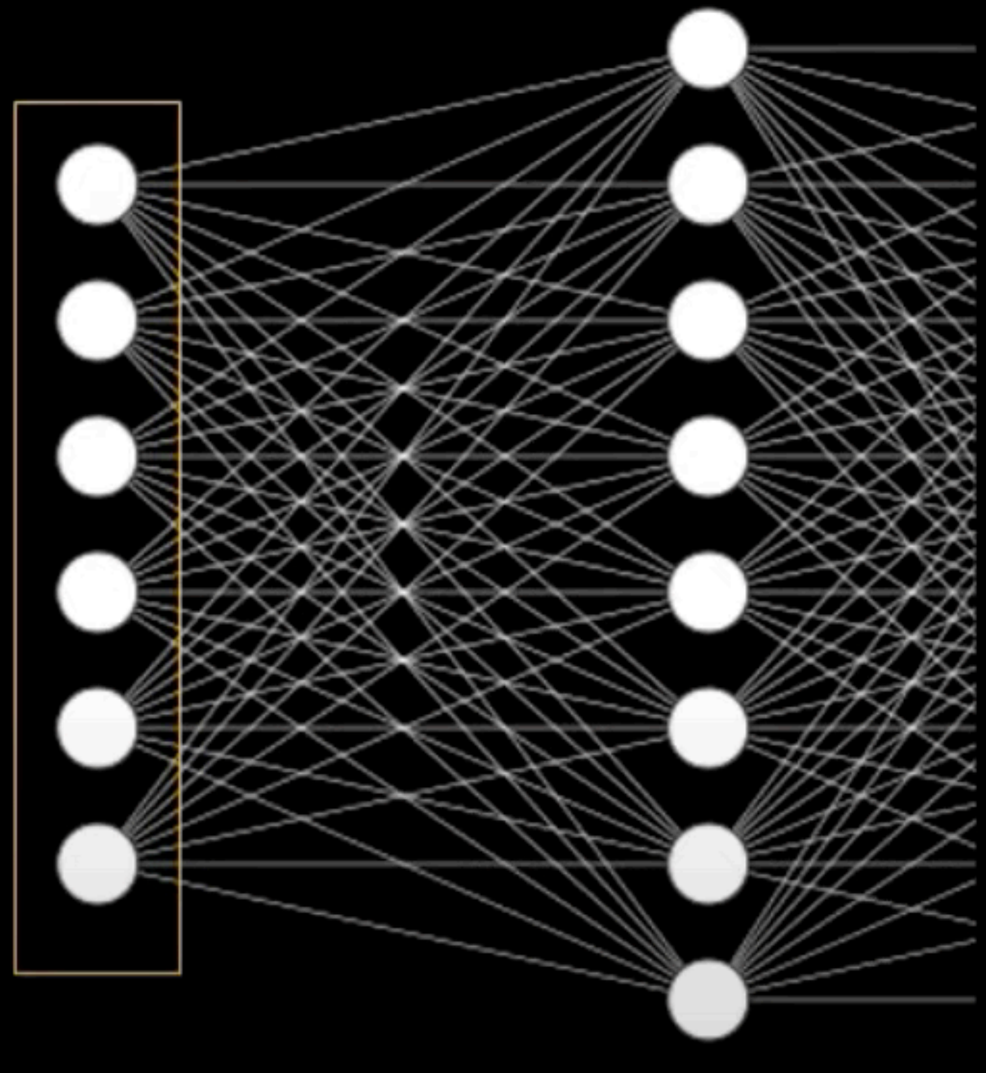
Autocomplete

ChatGPT

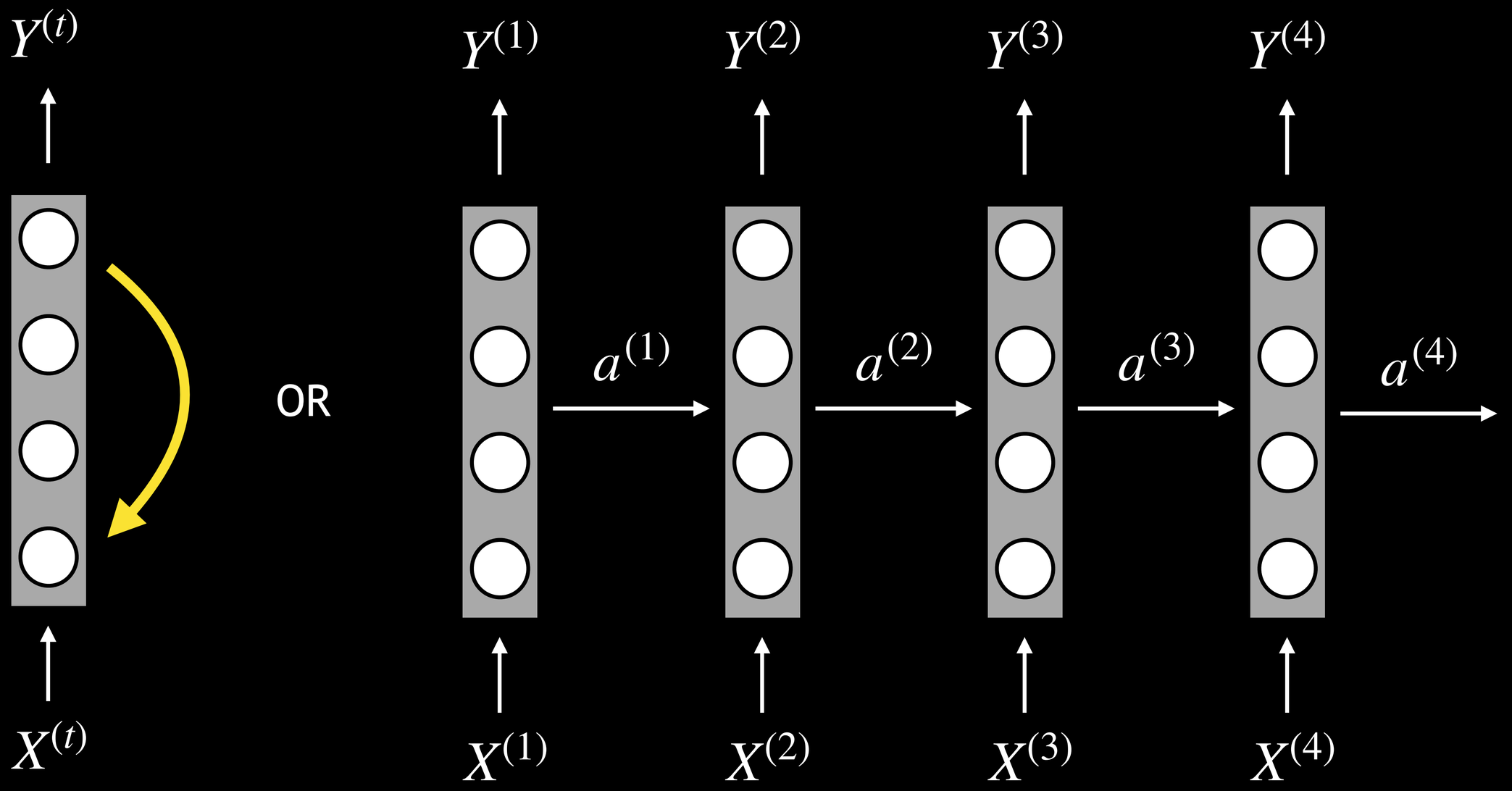
Recurrent Neural Networks (RNN)

Why do we need them?

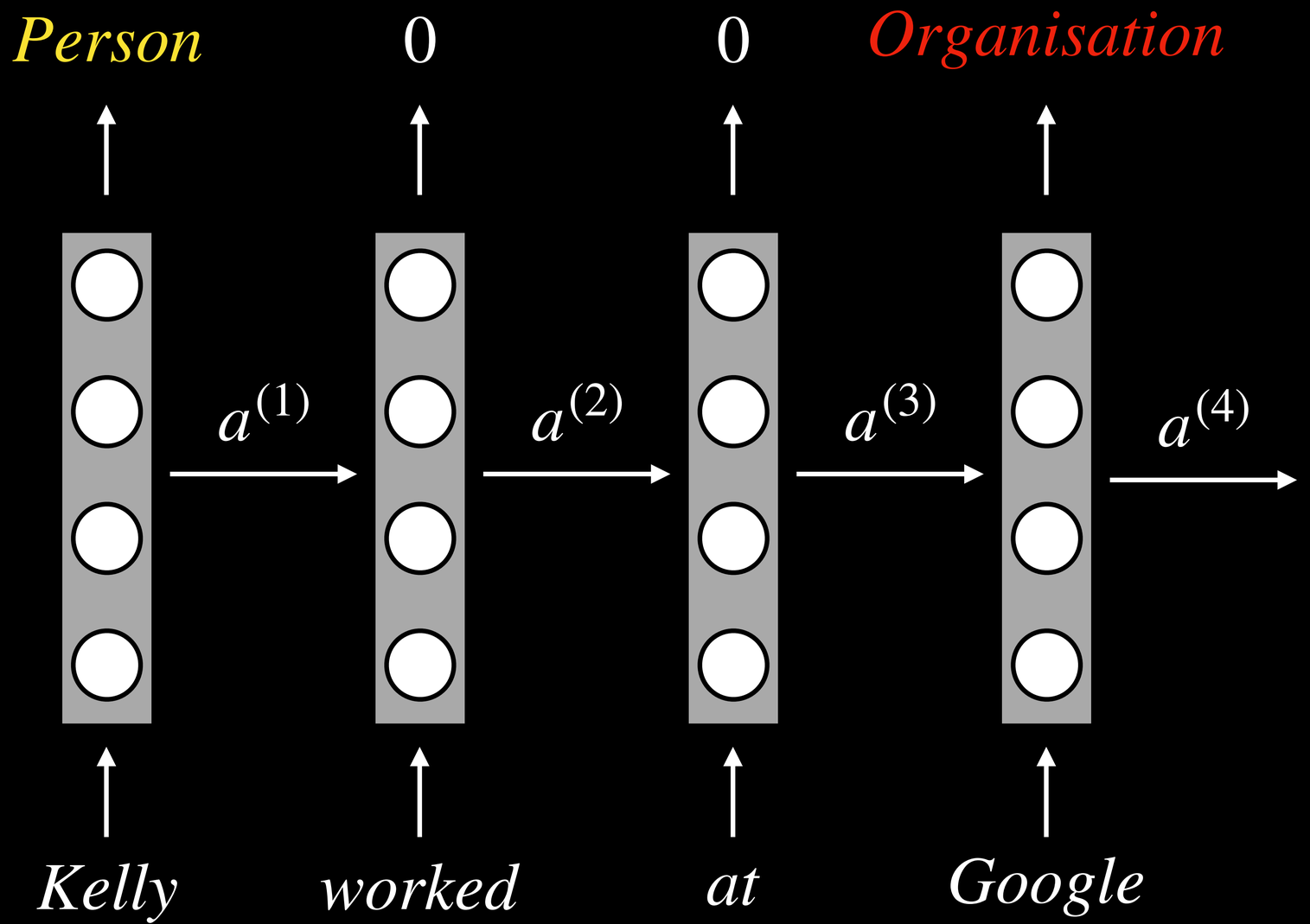
- A simple neural network has a fixed number of inputs and outputs - this creates a challenge if we want to train a neural network for translation
- Another issue is that a simple neural network does not take into account the input sequence into consideration - e.g. “Joy is clever, not an idiot” vs “Joy is an idiot, not clever”



RNN - Introduction



RNN - Entity Recognition Example



RNNs

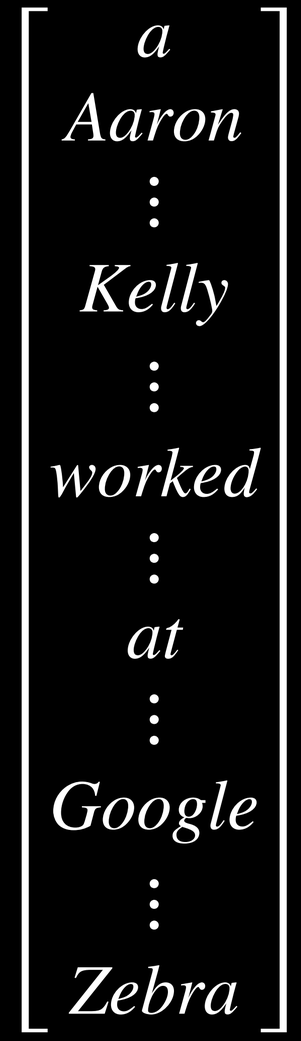
3 Types

- Many-to-one: Many inputs to one output, e.g. converting a text movie review to star rating
- One-to-many: One input to many outputs, e.g. baby name generator which takes in the biological sex as an input
- Many-to-many: Many inputs to many outputs, and there are two main sub-types:
  - Same input and output length, e.g. entity recognition
  - Different input and output lengths, e.g. translation

RNNs

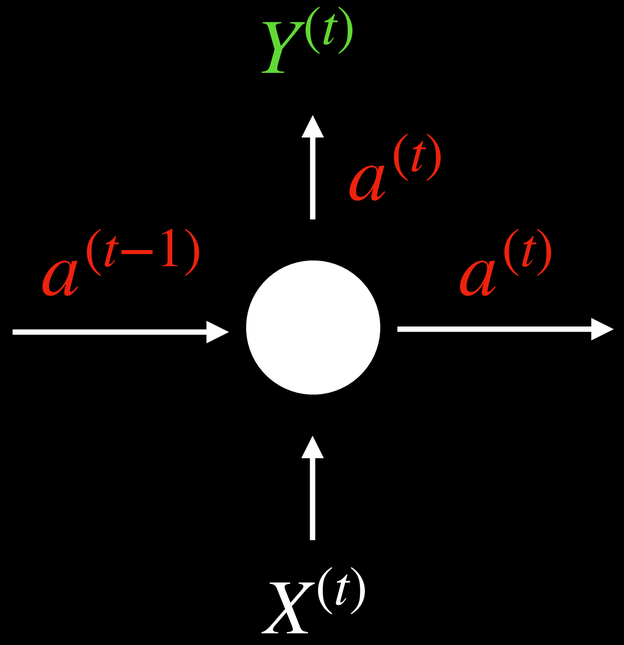
Converting Language into Numbers

- We know that neural networks only take in numerical inputs and this could be a challenge when working with words and languages
- But just as we did with the image, we can convert language into numbers
- We do this by creating a dictionary of vocabulary in a language and then assigning a number to each word
- Then each word can be translated into a vector of 0s and 1s, with the 1 denoting the location of the word in the dictionary



RNNs

What happens in each cell?

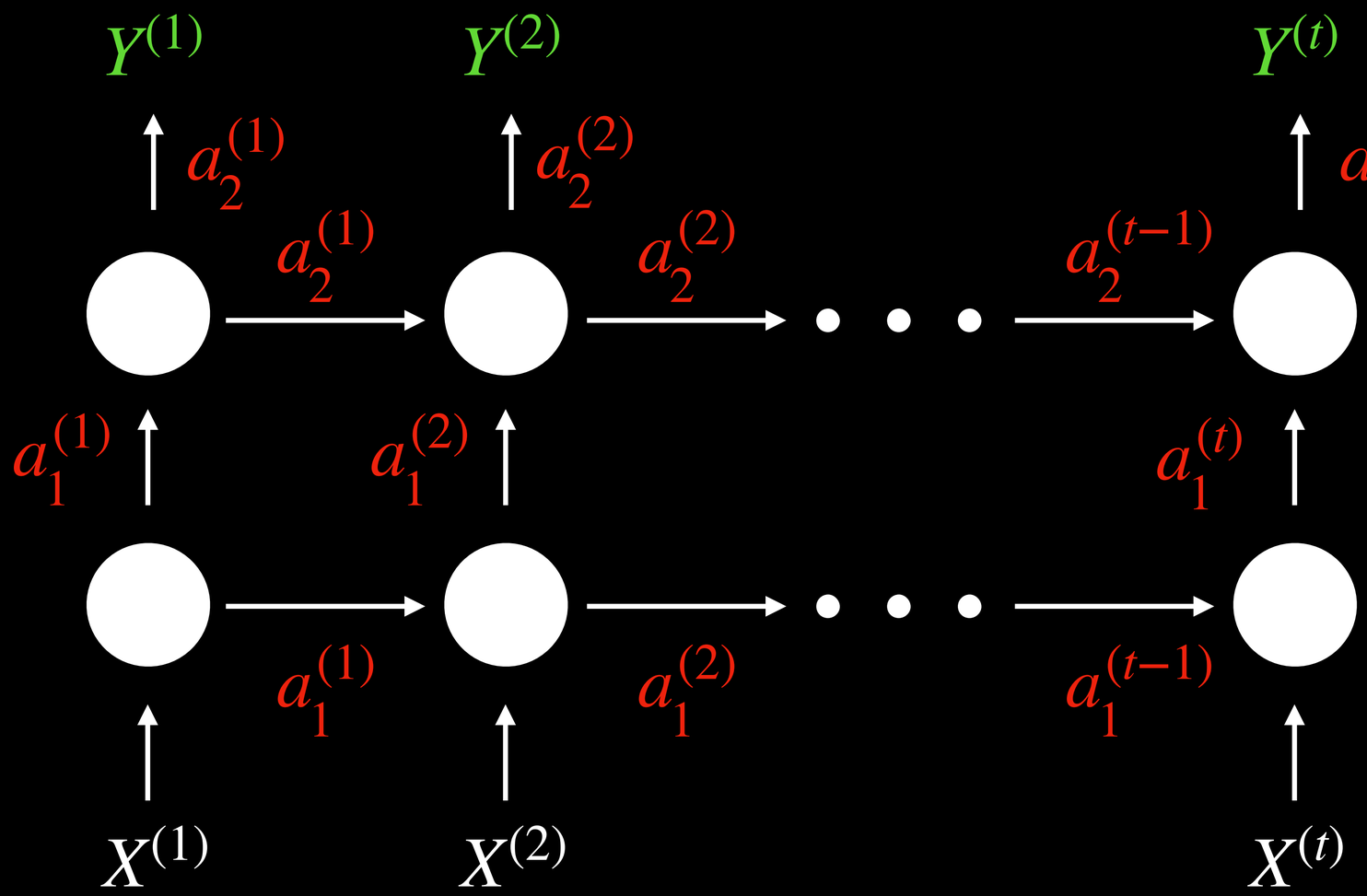


$$a^{(t)} = f\left(W_{aa}\ a^{(t-1)} + W_{ax}\ X^{(t)} + b_a\right)$$

$$Y^{(t)} = g\left(W_{ya}\ a^{(t)} + b_y\right)$$

RNNs

Forward Propagation



$$a_1^{(t)} = f\left(W_{aa}\ a_1^{(t-1)} + W_{ax}\ X^{(t)} + b_a\right)$$

$$a_2^{(t)} = f\left(W_{aa}\ a_2^{(t-1)} + W_{ax}\ a_1^{(t)} + b_a\right)$$

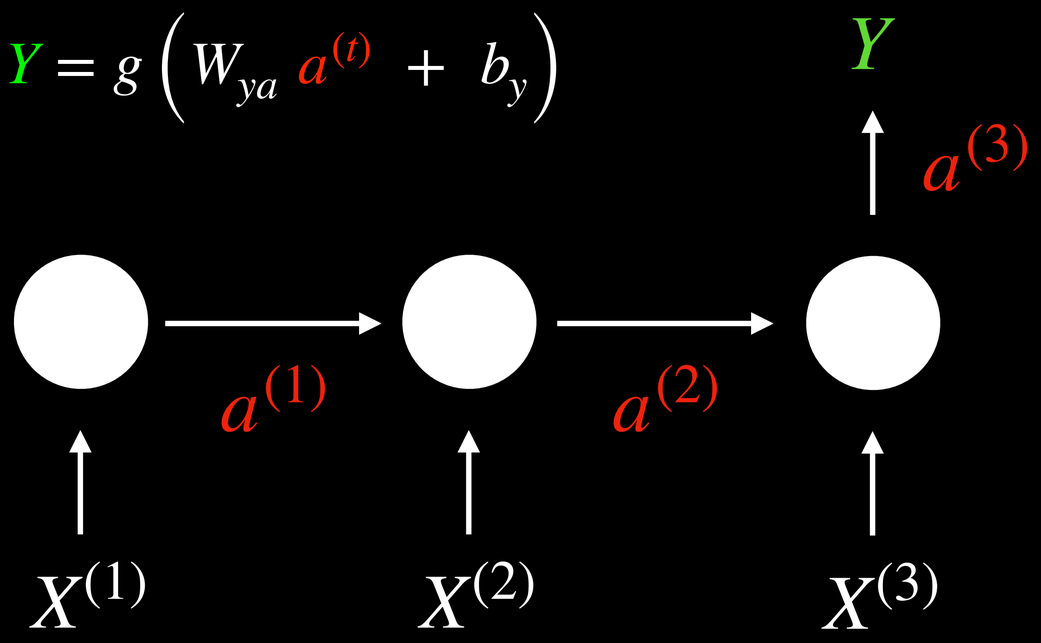
$$Y^{(t)} = g\left(W_{ya}\ a_2^{(t)} + b_y\right)$$

RNNs

Backward Propagation: Many-to-One Example

$$a^{(t)} = f\left(W_{aa}\, a^{(t-1)} + W_{ax}\, X^{(t)} + b_a\right)$$

$$Y = g\left(W_{ya}\, a^{(t)} + b_y\right)$$



Assume some cost function L, which allows us to calculate the accuracy of the RNN.

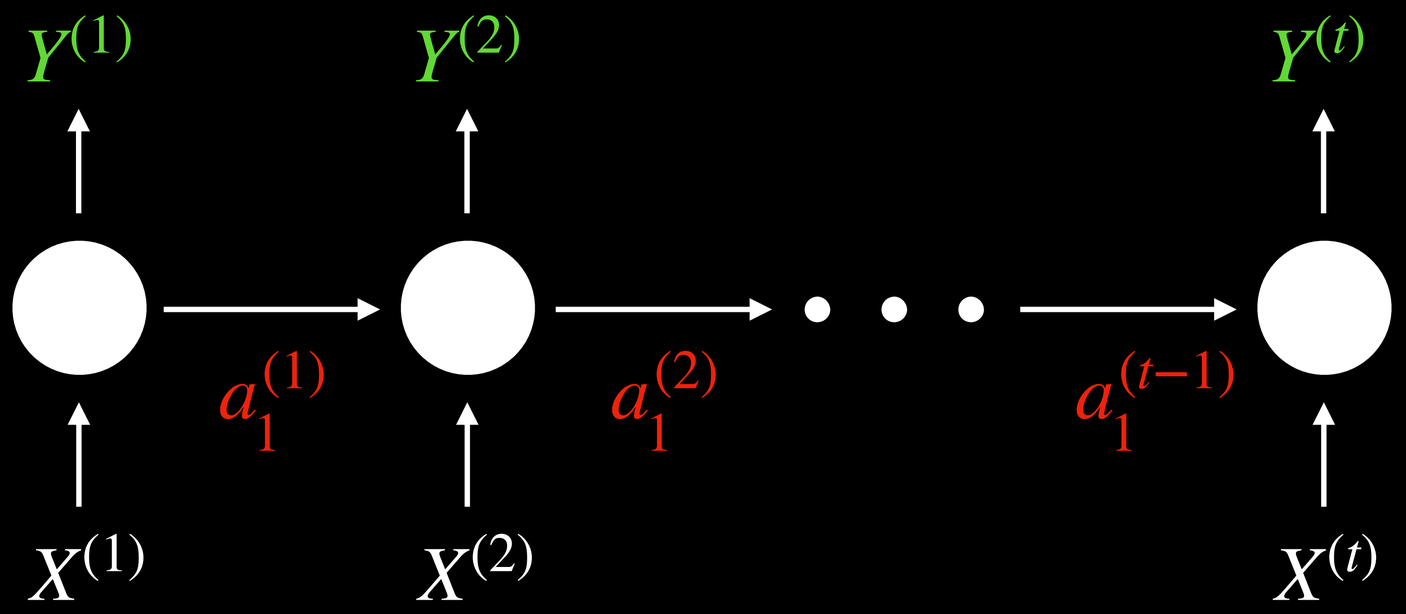
$$\frac{\partial L}{\partial W_{ya}} = \frac{\partial L}{\partial Y}\frac{\partial Y}{\partial W_{ya}}$$

$$\frac{\partial L}{\partial W_{aa}} = \frac{\partial L}{\partial Y}\frac{\partial Y}{\partial a_3}\frac{\partial a_3}{\partial W_{aa}} + \frac{\partial L}{\partial Y}\frac{\partial Y}{\partial a_3}\frac{\partial a_3}{\partial a_2}\frac{\partial a_2}{\partial W_{aa}} + \frac{\partial L}{\partial Y}\frac{\partial Y}{\partial a_3}\frac{\partial a_3}{\partial a_2}\frac{\partial a_2}{\partial a_1}\frac{\partial a_1}{\partial W_{aa}}$$

$$\frac{\partial L}{\partial W_{ax}} = \frac{\partial L}{\partial Y}\frac{\partial Y}{\partial a_3}\frac{\partial a_3}{\partial W_{ax}} + \frac{\partial L}{\partial Y}\frac{\partial Y}{\partial a_3}\frac{\partial a_3}{\partial a_2}\frac{\partial a_2}{\partial W_{ax}} + \frac{\partial L}{\partial Y}\frac{\partial Y}{\partial a_3}\frac{\partial a_3}{\partial a_2}\frac{\partial a_2}{\partial a_1}\frac{\partial a_1}{\partial W_{ax}}$$

RNNs

Information Loss & Vanishing Gradient



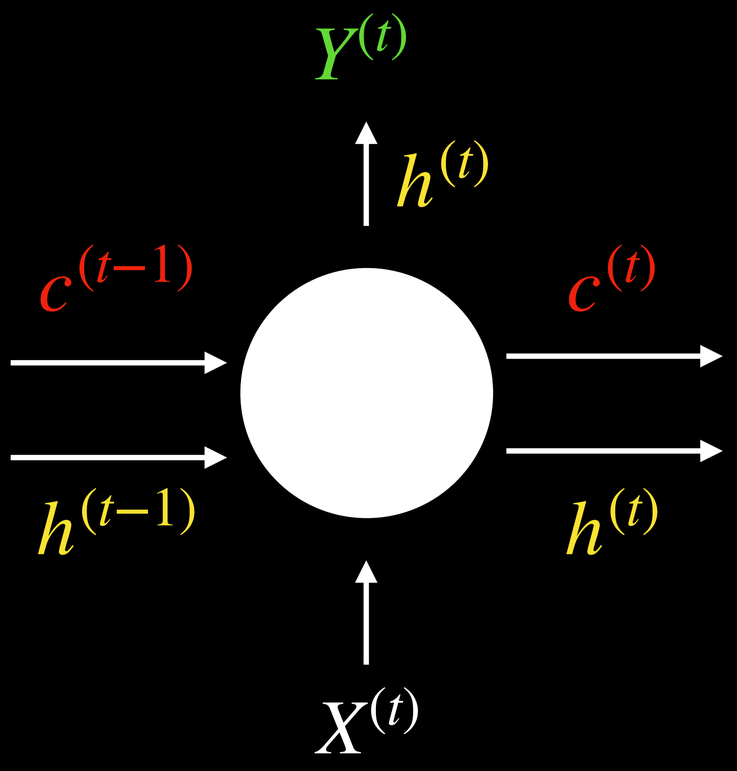
Mr. Watson goes on a business trip every Wednesday, and one day during a summer break, her wife was very angry…

By the time the RNN reaches her, the subject (Mr. Watson) will have gone through so many iterations that the information about the subject will be lost.

This means a grammatical correction algorithm will not work for this example.

You can also see this from the backward propagation equation in the previous slide - giving it the name “Vanishing Gradient”

Long Short Term Memory (LSTM)



$$C^{(t)} = f_t * C^{(t-1)} + i_t * \tilde{C}_t$$

$$\tilde{C}^{(t)} = g\left(W_{ch}\, h^{(t-1)} + W_{cx}\, X^{(t)} + b_c\right)$$

$$f_t = \sigma\left(W_{fh}\, h^{(t-1)} + W_{fx}\, X^{(t)} + b_f\right)$$

$$i_t = \sigma\left(W_{ih}\, h^{(t-1)} + W_{ix}\, X^{(t)} + b_i\right)$$

$$o_t = \sigma\left(W_{oh}\, h^{(t-1)} + W_{ox}\, X^{(t)} + b_o\right)$$

$$h^{(t)} = o_t\; g(C_t)$$

$$Y^{(t)} = g\left(W_{yh}\, h^{(t)} + b_y\right)$$

Forget Gate - Tells the network to remember or forget something (values 0 to 1)

Input Gate - Tells the network there is new information (values 0 to 1)

Output Gate - Tells the network it needs to output something (values 0 to 1)

Long Short Term Memory (LSTM)

The Intuition behind Gates

Forget Gate:

Mr Watson goes on a business trip every Wednesday, and one day during a summer break, her wife was very angry.
Ms. Mary is Mr Watson’s friend, and she is a business woman himself.

Input Gate:

Mr Watson goes on a business trip every Wednesday, and one day during a summer break, her wife was very angry.
And tomorrow, he was need to go on a trip again.