

$i$

# Deep Learning Specialization

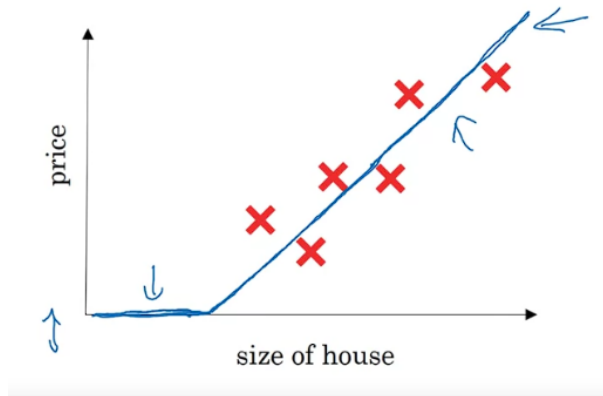
Declan Lim

July 13, 2022

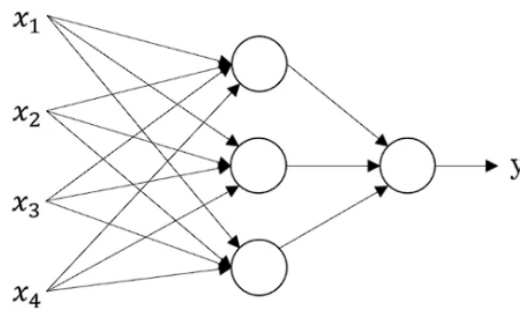
# 1 Neural Networks and Deep Learning

## 1.1 Introduction to Deep Learning

- Takes input  $x$  to a “neuron” and gives some output  $y$



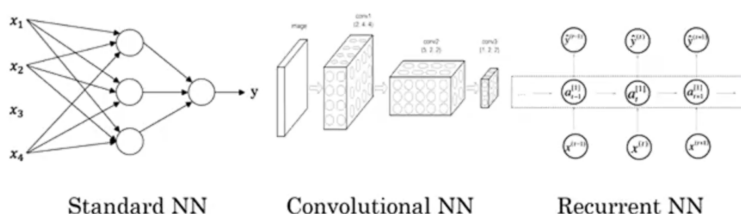
- Simple neural network has a single input, neuron and output
- $x$ : size of the house
- $y$ : price of the house
- Hypothesis (blue line) is a ReLU (Rectified Linear Unit)
- More complex neural networks can be formed by “stacking” neurons



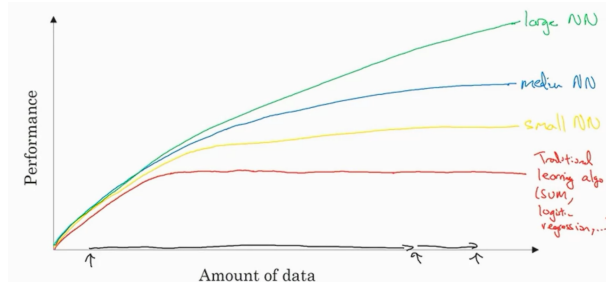
- Every input layer feature is interconnected with every hidden layer feature
  - The neural network will decide what the intermediate features will be
- Most useful in supervised learning settings

### 1.1.1 Supervised Learning

- Aims to learn a function to map an input  $x$  to an output  $y$ 
  - Real estate: predicting house prices from the house features
  - Online advertising: showing ads based on probability of user clicking on ad
  - Photo tagging: tagging images based on objects in the image
  - Speech recognition: generating a text transcript from audio
  - Machine translation: translating from one language to another
  - Autonomous driving: returning the positions of other cars from images and radar info
- Different types of neural network used for different tasks
  - Standard neural network: real estate and online advertising
  - Convolutional neural network (CNN): image data
  - Recurrent neural network (RNN): audio and language data (sequenced data)
  - Hybrid neural network: Autonomous driving (more complex input)



- Supervised learning can be applied to structured and unstructured data
  - Structured data has features with well defined meanings
  - Unstructured data has more abstract features (images, audio, text)
- Deep learning has only recently started to become more widespread
  - Given large amounts of data and a large NN, deep learning will outperform more traditional learning algorithms
  - For small amounts of data, any performance of the algorithm depends on specific implementation
- “Scale drives deep learning progress”
  - Both the scale of the data and the NN
- Recent algorithmic innovations with increase scale of computation



- Idea to switch from sigmoid activation function to ReLu function increased NN performance
- Ends of sigmoid function have close to 0 gradient so and therefore result in small changes in  $\theta$
- ReLu function has gradient of 1 for positive values
- Neural network process is iterative
  - Increasing speed at which a NN can be trained allows different ideas to be tried

## 1.2 Neural Network Basics

### 1.2.1 Logistic Regression as a Neural Network

- Logistic regression used for binary classification
- For a colour image, of  $64 \times 64$  pixels, will have total 12288 input features
  - Image is stored as 3 separate matrices for each colour channel
  - All pixel intensities should be unrolled into a single feature vector

$$n = 12288$$

$$x \in \mathbb{R}^{12288}$$

- For a matrix  $X$  of shape  $(a, b, c, d)$ , want a matrix `X_flatten` of shape  $(b * c * d, 1)$

```
X_flatten = X.reshape(X.shape[0], -1).T
```

### Notation

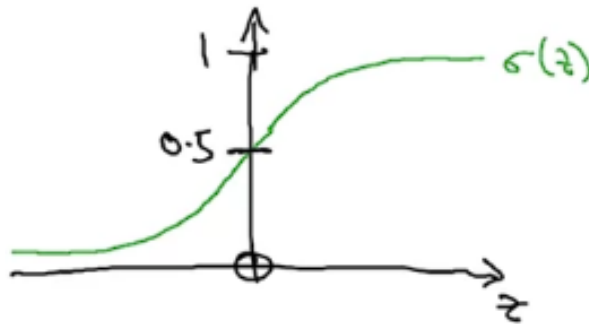
$$\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \dots, (x^{(m)}, y^{(m)})\}$$

- $(x, y)$ : single training example
  - $x \in \mathbb{R}^{n_x}$  ( $n_x$  = number of features)
  - $y \in \{0, 1\}$

- $(x^{(i)}, y^{(i)})$ :  $i^{th}$  training example
- $m = m_{train}$
- $m_{test} = \#$  of test examples
- $X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \dots & x^{(m)} \\ | & | & & | \end{bmatrix}$ 
  - $X \in \mathbb{R}^{n_x \times m}$
- $Y = [y^{(1)} \ y^{(2)} \ \dots \ y^{(m)}]$ 
  - $Y \in \mathbb{R}^{1 \times m}$

## Logistic Regression

- Given  $x$ , want  $\hat{y} = P(y = 1|x)$ 
  - Since  $\hat{y}$  is a probability, want  $0 \leq \hat{y} \leq 1$
- Parameters:  $w \in \mathbb{R}^{n_x}, b \in \mathbb{R}$
- Output:  $\hat{y} = \sigma(w^T x + b)$



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

$$z = w^T x + b$$

- Aim is to learn parameters  $w$  and  $b$  such that  $\hat{y}$  is a good estimate of the probability
- Previous convention had  $\theta$  vector with an additional  $\theta_0$  parameter
  - Keeping  $\theta_0$  ( $b$ ) separate from the rest of the parameters is easier to implement

## Cost Function

- Given  $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \dots, (x^{(m)}, y^{(m)})\}$ , want  $\hat{y}^{(i)} \approx y^{(i)}$
- Squared error function not used for logistic regression loss function

- Optimization problem becomes non convex and will have local optima

$$\mathcal{L}(\hat{y}, y) = -(y \log(\hat{y}) + (1 - y) \log(1 - \hat{y}))$$

- If  $y = 1$ :
  - $\mathcal{L}(\hat{y}, y) = -\log(\hat{y})$
  - Want large  $\log(\hat{y}) \therefore$  want large  $\hat{y}$
  - $\hat{y}$  has a max of 1  $\therefore$  want  $\hat{y} = 1$
- If  $y = 0$ :
  - $\mathcal{L}(\hat{y}, y) = -\log(1 - \hat{y})$
  - Want large  $\log(1 - \hat{y}) \therefore$  want small  $\hat{y}$
  - $\hat{y}$  has a min of 0  $\therefore$  want  $\hat{y} = 0$

- Cost function:

$$\begin{aligned} J(w, b) &= \frac{1}{m} \sum_{i=1}^m \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) \\ &= -\frac{1}{m} \sum_{i=1}^m [y^{(i)} \log(\hat{y}^{(i)}) + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)})] \end{aligned}$$

- Average loss function over all training examples

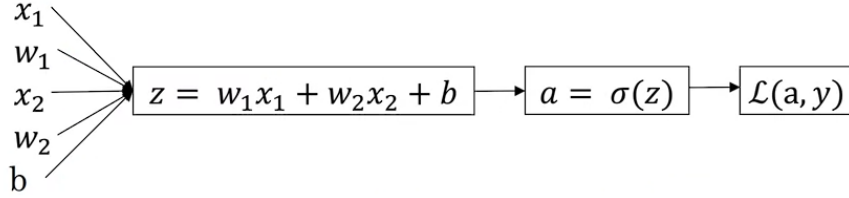
### ***Gradient Descent***

- Want to find values of  $w$  and  $b$  that minimize the cost function  $J(w, b)$ 
  - For logistic regression,  $w$  and  $b$  usually initialized to 0
- One iteration of gradient descent will take a step in the direction of steepest descent

```
Repeat {
  w := w -  $\alpha \frac{\partial J(w, b)}{\partial w}$ 
  b := b -  $\alpha \frac{\partial J(w, b)}{\partial b}$ 
}
```

- Using the computation graph:

$$\frac{\partial \mathcal{L}(a, y)}{\partial a} = -\frac{y}{a} + \frac{1 - y}{1 - a}$$



$$\begin{aligned}
 \frac{\partial \mathcal{L}(a, y)}{\partial z} &= \frac{\partial \mathcal{L}}{\partial a} \times \frac{\partial a}{\partial z} \\
 &= \left(-\frac{y}{a} + \frac{1-y}{1-a}\right) \times a(1-a) \\
 &= a - y
 \end{aligned}$$

$$\frac{\partial \mathcal{L}}{\partial w_1} = x_1 \times \frac{\partial \mathcal{L}}{\partial z}$$

$$\frac{\partial \mathcal{L}}{\partial w_2} = x_2 \times \frac{\partial \mathcal{L}}{\partial z}$$

$$\frac{\partial \mathcal{L}}{\partial b} = \frac{\partial \mathcal{L}}{\partial z}$$

- Partial derivative over all training examples calculated by taking the average  $\mathbf{dw1}$

$$\frac{\partial}{\partial w_1} J(w, b) = \frac{1}{m} \sum_{i=1}^m \frac{\partial}{\partial w_1} \mathcal{L}(a^{(i)}, y^{(i)})$$

Initialize  $J = 0$ ,  $\mathbf{dw1} = 0$ ,  $\mathbf{dw2} = 0$ ,  $\mathbf{db} = 0$

For  $i = 1$  to  $m$ :

$$\mathbf{z}^{(i)} = \mathbf{w}^T \mathbf{x}^{(i)} + \mathbf{b}$$

$$\mathbf{a}^{(i)} = \sigma(\mathbf{z}^{(i)})$$

$$J += -[y^{(i)} \log(a^{(i)}) + (1-y^{(i)}) \log(1-a^{(i)})]$$

$$d\mathbf{z}^{(i)} = \mathbf{a}^{(i)} - \mathbf{y}^{(i)}$$

$$d\mathbf{w1} += \mathbf{x}_1^{(i)} d\mathbf{z}^{(i)}$$

$$d\mathbf{w2} += \mathbf{x}_2^{(i)} d\mathbf{z}^{(i)}$$

$$d\mathbf{b} += d\mathbf{z}^{(i)}$$

$J /= m$

$\mathbf{dw1} /= m$

$\mathbf{dw2} /= m$

$\mathbf{db} /= m$

$$\mathbf{w1} := \mathbf{w1} - \alpha d\mathbf{w1}$$

$$\mathbf{w2} := \mathbf{w2} - \alpha d\mathbf{w2}$$

$$\mathbf{b} := \mathbf{b} - \alpha d\mathbf{b}$$



- Above implementation requires **for** loop over all features for all training examples
  - Vectorization can be used to remove explicit **for** loops
  - Vectorization required for deep learning to be efficient

### 1.2.2 Vectorisation in Python

- Deep learning performs best on large data sets
  - Code must be able to run quickly to be effective on large data sets

$$z = w^T x + b$$

$$w \in \mathbb{R}^{n_x} \quad x \in \mathbb{R}^{n_x}$$

- Non vectorized implementation:

```
z = 0
for i in range(n_x):
    z += w[i] * x[i]
z += b
```

- GPUs and CPUs both have parallelization instructions (SIMD: Single Instruction Multiple Data)
  - If built in functions are used, **numpy** will use parallelism to perform computations faster
- For logistic regression, need to calculate  $z$  and  $a$  values for each training example

$$z^{(i)} = w^T x^{(i)} + b$$

$$a^{(i)} = \sigma(z^{(i)})$$

$$X = \begin{bmatrix} | & | & \dots & | \\ x^{(1)} & x^{(2)} & \dots & x^{(m)} \\ | & | & & | \end{bmatrix}$$

$$w \in \mathbb{R}^{n_x} \quad X \in \mathbb{R}^{n_x \times m}$$

$$\begin{aligned} [z^{(1)} \quad z^{(2)} \quad \dots \quad z^{(m)}] &= w^T X + [b \quad b \quad \dots \quad b] \\ &= [w^T x^{(1)} + b \quad w^T x^{(2)} + b \quad \dots \quad w^T x^{(m)} + b] \end{aligned}$$

- In Python:

```
Z = np.dot(w.T, X) + b
```

- Python will broadcast the value **b** so it can be added to the matrix
- Vectorized implementation of sigmoid function can be used on **Z** to calculate **A**

$$A = [a^{(1)} \quad a^{(2)} \quad \dots \quad a^{(m)}]$$

$$dz^{(i)} = a^{(i)} - y^{(i)}$$

$$dz = A - Y$$

$$db = \frac{1}{m} \sum_{i=1}^m dz^{(i)}$$

$$dw = \frac{1}{m} X(dz)^T$$

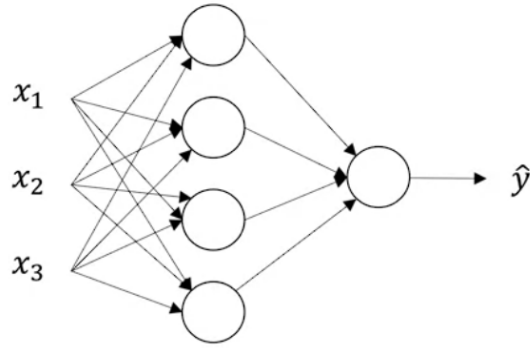
```
Z = np.dot(w.T,X) + b
A = sigmoid(Z)
dz = A - Y
dw = 1/m * np.dot(X, dz.T)
db = 1/m * np.sum(dz)

# Gradient descent update
w = w - alpha * dw
b = b - alpha * db
```

- for loop is required to run multiple iterations of gradient descent

### 1.3 Shallow Neural Networks

- A neural network will have stacked logistic regression units in each layer
  - Logistic regression output from one layer will be fed to another layer



- Input layer of the neural network contains the feature  $x_1, x_2, x_3$ 
  - $a^{[0]} = X$
- Intermediate layers in the network are hidden layers
  - Hidden layers do not have “true” values in the training set
- Final layer in the network is the output layer
  - Generates the predicted value  $\hat{y}$
- Above diagram is a 2 layer NN
  - Input layer is layer 0
- Each layer will have parameters  $w$  and  $b$  associated with them
- Each node in the NN will perform logistic regression with its inputs

$$z_i^{[l]} = w_i^{[l]T} x + b_i^{[l]} \rightarrow a_i^{[l]} = \sigma(z_i^{[l]})$$

$$W^{[1]} = \begin{bmatrix} - & w_1^{[1]T} & - \\ - & w_2^{[1]T} & - \\ - & w_3^{[1]T} & - \\ - & w_4^{[1]T} & - \end{bmatrix}$$

$$a^{[0]} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

$$b^{[1]} = \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \\ b_4^{[1]} \end{bmatrix}$$

$$\begin{aligned}
z^{[1]} &= \begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \\ z_4^{[1]} \end{bmatrix} \\
&= \begin{bmatrix} w_1^{[1]T} a^{[0]} + b_1^{[1]} \\ w_2^{[1]T} a^{[0]} + b_1^{[1]} \\ w_3^{[1]T} a^{[0]} + b_1^{[1]} \\ w_4^{[1]T} a^{[0]} + b_1^{[1]} \end{bmatrix} \\
&= w^{[1]} a^{[0]} + b^{[1]}
\end{aligned}$$

$$\begin{aligned}
a^{[1]} &= \begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \\ a_4^{[1]} \end{bmatrix} \\
&= \sigma(z^{[1]})
\end{aligned}$$

$$z^{[2]} = W^{[2]} a^{[1]} + b^{[2]} \rightarrow a^{[2]} = \sigma(z^{[2]})$$

- Vectorized method should be able to work on all training examples at one time
  - Vector for each training example can be stacked horizontally in a matrix
  - Vertical dimension will be the number of units in a layer ( $n_x$  for the input layer)

$$X = \begin{bmatrix} \begin{matrix} | \\ x^{(1)} \\ | \end{matrix} & \begin{matrix} | \\ x^{(2)} \\ | \end{matrix} & \begin{matrix} | \\ x^{(m)} \\ | \end{matrix} \end{bmatrix}$$

$$Z^{[1]} = \begin{bmatrix} \begin{matrix} | \\ z^{[1](1)} \\ | \end{matrix} & \begin{matrix} | \\ z^{[1](2)} \\ | \end{matrix} & \dots & \begin{matrix} | \\ z^{[1](m)} \\ | \end{matrix} \end{bmatrix}$$

$$A^{[1]} = \begin{bmatrix} \begin{matrix} | \\ a^{[1](1)} \\ | \end{matrix} & \begin{matrix} | \\ a^{[1](2)} \\ | \end{matrix} & \dots & \begin{matrix} | \\ a^{[1](m)} \\ | \end{matrix} \end{bmatrix}$$

$$Z^{[1]} = W^{[1]} X + b^{[1]}$$

$$A^{[1]} = \sigma(Z^{[1]})$$

$$Z^{[2]} = W^{[2]} A^{[1]} + b^{[2]}$$

$$A^{[2]} = \sigma(Z^{[2]})$$

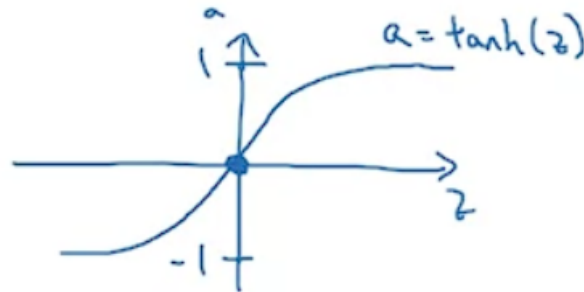
### 1.3.1 Activation Functions

- After  $z$  values are calculated, activation function must be run to get the activation value  $a$

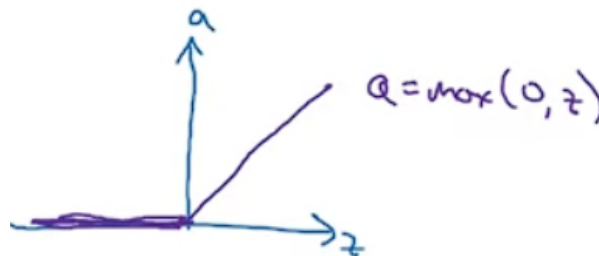
$$a_{sigmoid} = \frac{1}{1 + e^{-z}}$$

- Alternatively  $a^{[1]} = g(z^{[1]})$  where  $g$  is a non linear function
- tanh function almost always performs better than the sigmoid function
  - Equivalent to a transformed version of the sigmoid function
  - tanh function is odd and is “centered” around the origin
  - The mean of the data will be closer to 0 and will help with learning in the next layer

$$a_{tanh} = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

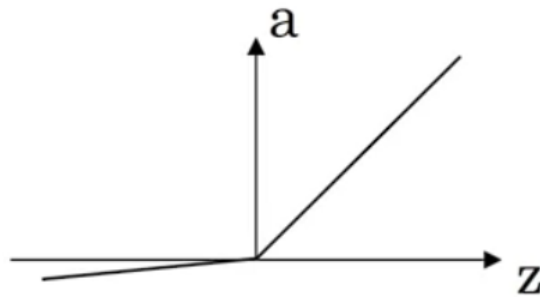


- For binary classification, the final output layer can use the sigmoid function
  - Want the value of  $\hat{y}$  to be between 0 and 1
- For both the sigmoid and tanh functions, when  $z$  is large, the gradient is very small
  - Results in a slower gradient descent
- ReLU function has a gradient of 1 when  $z$  is positive



- Gradient is 0 when  $z$  is negative
- For majority of the ReLU function, gradient is very different from 0
  - Will typically allow NN to learn much faster than sigmoid or tanh function
- ReLU function should be used as the default activation function
- The leaky ReLU function has a slight positive gradient when  $z$  is negative

$$a_{leakyReLU} = \max(0.01z, z)$$



- For a NN to compute more complex functions, activation function must be non linear
  - If a linear activation function is used, final output of the NN can only be a linear function
  - Multiple linear activation neurons with a sigmoid as the output neuron is equivalent to standard logistic regression
- Linear activation function can be used in the output layer if output is a real number
- Derivative of the activation function must be calculated for backpropagation
  - Sigmoid function

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

$$\begin{aligned} \frac{d}{dz}g(z) &= \frac{1}{1 + e^{-z}} \left( 1 - \frac{1}{1 + e^{-z}} \right) \\ &= g(z)(1 - g(z)) \end{aligned}$$

- tanh function

$$\begin{aligned} g(z) &= \tanh(z) \\ &= \frac{e^z - e^{-z}}{e^z + e^{-z}} \end{aligned}$$

$$\begin{aligned}\frac{d}{dz}g(z) &= 1 - \left(\frac{e^z - e^{-z}}{e^z + e^{-z}}\right)^2 \\ &= 1 - g(z)^2\end{aligned}$$

– ReLU function

$$\begin{aligned}g(z) &= \max(0, z) \\ \frac{d}{dz}g(z) &= \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases}\end{aligned}$$

– Leaky ReLU function

$$\begin{aligned}g(z) &= \max(0.01z, z) \\ \frac{d}{dz}g(z) &= \begin{cases} 0.01 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases}\end{aligned}$$

### 1.3.2 Gradient Descent for Neural Networks

- For a single hidden layer NN, parameters are:  $w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]}$ 
  - $w^{[1]} \in \mathbb{R}^{n_1 \times n_0}$
  - $b^{[1]} \in \mathbb{R}^{n_1 \times 1}$
  - $w^{[2]} \in \mathbb{R}^{n_2 \times n_1}$
  - $b^{[2]} \in \mathbb{R}^{n_2 \times 1}$
- Cost function:  $J(w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]}) = \frac{1}{m} \sum_{i=1}^n \mathcal{L}(\hat{y}, y)$
- For one iteration of gradient descent:

$$w^{[1]} := w^{[1]} - \alpha dw^{[1]}, \quad b^{[1]} := b^{[1]} - \alpha db^{[1]}$$

$$w^{[2]} := w^{[2]} - \alpha dw^{[2]}, \quad b^{[2]} := b^{[2]} - \alpha db^{[2]}$$

– Gradient descent step will take place after backpropagation calculates the derivatives

- Forward propagation:

$$Z^{[1]} = W^{[1]}X + b^{[1]}$$

$$A^{[1]} = g^{[1]}(Z^{[1]})$$

$$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$$

$$A^{[2]} = g^{[2]}(Z^{[2]})$$

- Backpropagation:

$$\begin{aligned}
 dz^{[2]} &= A^{[2]} - Y \\
 dw^{[2]} &= \frac{1}{m} dz^{[2]} A^{[1]T} \\
 db^{[2]} &= \frac{1}{m} \text{np.sum}(dz^{[2]}, \text{axis} = 1, \text{keepdims} = \text{True}) \\
 dz^{[1]} &= w^{[2]T} dz^{[2]} \times g^{[1]'}(z^{[1]}) \\
 dw^{[1]} &= \frac{1}{m} dz^{[1]} X^T \\
 db^{[1]} &= \frac{1}{m} \text{np.sum}(dz^{[1]}, \text{axis} = 1, \text{keepdims} = \text{True})
 \end{aligned}$$

### 1.3.3 Random Initialization

- Weights must be initialized randomly for a NN
  - Weights can be initialized to 0 for logistic regression
  - The bias terms  $b$  can be initialized
- If weights are initialized to 0, all neurons in a layer will compute the same hypothesis

```

W1 = np.random.randn((2,2)) * 0.01
b1 = np.zeros((2,1))

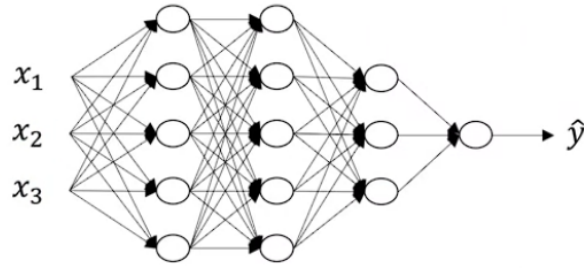
```

- Weights should be initialized to small random values
  - If weight is too large, activation value  $z^{[1]}$  will be large
  - If sigmoid or tanh function is used, derivative will be very small and learning will be very slow
- Different constant for `np.random.randn` should be used for deeper neural networks

## 1.4 Deep Neural Networks

- Logistic regression is equivalent to a 1-layer NN
- Deep NN have more hidden layers
  - Number of hidden layers in the network can be a parameter for the ML problem





- Above network has 4 layers,  $L = 4$
- $n^{[l]}$  = number of units in layer  $l$
- $a^{[l]}$  = activations in layer  $l$
- The inputs  $x$  are the activations of the first layer,  $x = a^{[0]}$ 
  - Prediction  $\hat{y}$  will be the activations of the last layer,  $\hat{y} = a^{[L]}$
- Forward propagation for a deep NN will follow the same pattern for all layers

$$z^{[l]} = w^{[l]}a^{[l-1]} + b^{[l]}$$

$$a^{[l]} = g^{[l]}(z^{[l]})$$

- For a vectorized implementation

$$Z^{[l]} = W^{[l]}X + b^{[l]}$$

$$A^{[l]} = g^{[l]}(z^{[l]})$$

- Explicit for loop will be used to loop over the layers in the network
- When debugging NN, can look at dimensions of all the matrices
- For a non vectorized implementation:
  - $W^{[l]} : (n^{[l]}, n^{[l-1]})$
  - $b^{[l]} : (n^{[l]}, 1)$
  - Dimensions of  $dw$  and  $db$  should be the same as the dimensions of  $W$  and  $b$
  - $a^{[l]}, z^{[l]} : (n^{[l]}, 1)$
- For a vectorized implementation,  $z$  vectors and  $a$  vectors will be stacked horizontally for all training examples
  - $Z^{[l]}, A^{[l]} : (n^{[l]}, m)$
- Deep NN tend to work better as each layer can compute increasingly complex functions
  - Face recognition: edge detection  $\rightarrow$  individual features  $\rightarrow$  large parts of the face

- Audio: low level waveforms  $\rightarrow$  phonemes  $\rightarrow$  words  $\rightarrow$  sentences
- Functions that can be computed with a “small” deep neural network require exponentially more hidden units in a shallower network