

# Deep Learning Specialization

Declan Lim

July 25, 2022

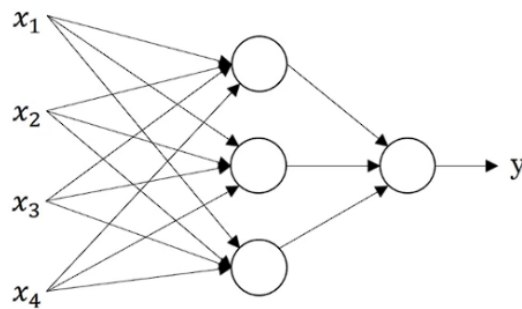
# 1 Neural Networks and Deep Learning

## 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  $m$ :

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

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

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

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

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

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

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

$J /= m$

$d\mathbf{w1} /= m$

$d\mathbf{w2} /= m$

$d\mathbf{b} /= m$

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

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

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

- 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**

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

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

$$dz = A - Y$$

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

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

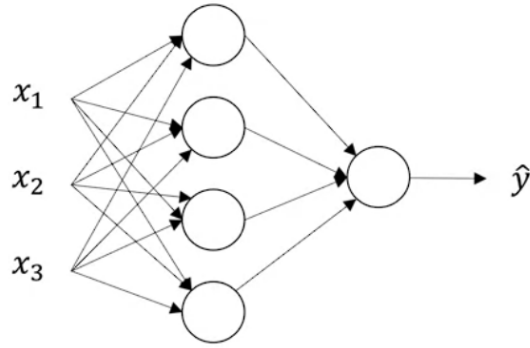
```
Z = np.dot(w.T,X) + b
A = sigmoid(Z)
dz = A - Y
dw = 1/m * np.dot(X, dz.T)
db = 1/m * np.sum(dz)

# 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)

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

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

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

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

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

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

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

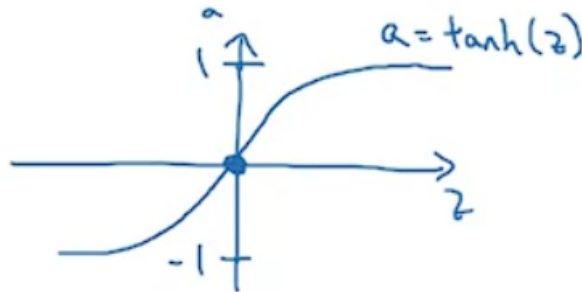
### 1.3.1 Activation Functions

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

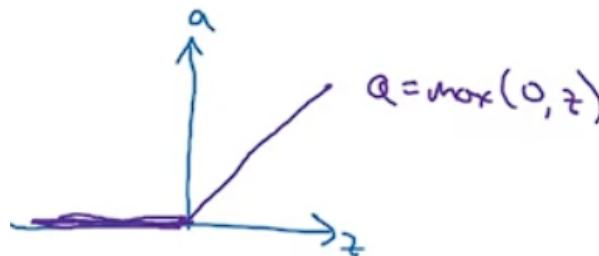
$$a_{sigmoid} = \frac{1}{1 + e^{-z}}$$

- 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

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

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

- Explicit for loop will be used to loop over the layers in the network
- $b$  will still be a column vector but will apply correctly due to broadcasting
- When working with  $W$  and  $A$  matrices,  $A$  will be for the previous layer so the dimensions will fit
- 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
- For each forward propagation step, the value of  $z^{[l]}$  should be cached for backpropagation
  - Values of  $w^{[l]}$  and  $b^{[l]}$  can also be stored in the cache so they can be accessed for backpropagation



- All forward propagation steps will be carried out until the hypothesis,  $\hat{y}$  is found
  - Using cached values, all backpropagation steps will be carried out until  $dz^{[1]}$
  - Parameters  $W^{[l]}$  and  $b^{[l]}$  can be updated accordingly

$$W^{[l]} := W^{[l]} - \alpha dw^{[l]}$$

$$b^{[l]} := b^{[l]} - \alpha db^{[l]}$$

- Backpropagation will also follow a pattern for all layers in the NN
  - $dz^{[l]} = da^{[l]} * g^{[l]'}(z^{[l]})$
  - $dW^{[l]} = dz^{[l]} a^{[l-1]T}$

- $db^{[l]} = dz^{[l]}$
- $da^{[l-1]} = W^{[l]T} dz^{[l]}$
- For a vectorized implementation:
  - $dZ^{[l]} = dA^{[l]} * g^{[l]'}(Z^{[l]})$
  - $dW^{[l]} = \frac{1}{m} dZ^{[l]} A^{[l-1]T}$
  - $db^{[l]} = \frac{1}{m} \text{np.sum}(dZ^{[l]}, \text{axis}=1, \text{keepdims}=\text{True})$
  - $dA^{[l-1]} = W^{[l]T} dZ^{[l]}$

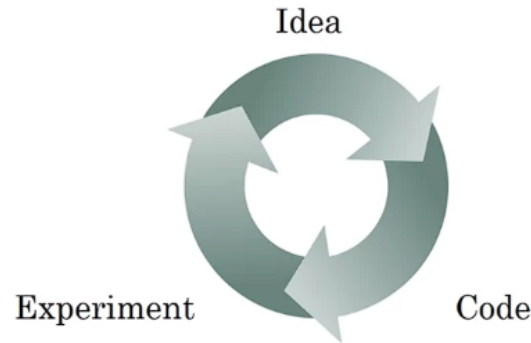
#### 1.4.1 Parameters vs Hyperparameters

- Parameters of the NN are the  $W$  and  $b$  matrices
- NN also has a number of associated hyperparameters:
  - Learning rate  $\alpha$
  - Number of iterations  $z'$
  - Number of layers in the network
  - Number of hidden units
  - Choice of activation function
- Hyperparameters will control the values of  $W$  and  $b$
- Deep learning has many more hyperparameters than earlier eras of machine learning
  - Applying deep learning becomes an empirical process
- Intuitions about hyperparameters may be different across different applications

## 2 Improving Deep Neural Networks: Hyperparameter Tuning, Regularization and Optimization

### 2.1 Practical Aspects of Deep Learning

- Applying ML is a highly iterative process
  - Very hard to choose “correct” values for hyperparameters



- Deep learning used in many different areas
  - NLP
  - Computer vision
  - Speech analysis
  - Structured data
    - \* Advertisement
    - \* Search engines
    - \* Computer security
    - \* Logistics
- Intuitions from one subject area often don't transfer to another application
- Success of deep learning can depend on speed of iteration
  - Choice of split of the data can influence speed of iteration
- Whole dataset should be split into training, development and test set
  - Dev set should be used rate performance of different models
  - Final model should be evaluated on the test set
  - Split will allow better evaluation of bias and variance of the model
- Previous eras of ML had a 60/20/20 split between dataset

- For the big data era, a smaller percentage of data is given to the dev and test sets
  - For 1,000,000 examples, can allocate just 10,000 examples each to dev and test set
  - 10,000 examples is enough to run the algorithm and get a good idea about the algorithm performance
- Recent trends also show mismatched training and test set distributions
  - For images, training set may have very high quality images while test set may have lower quality
  - Dev and test set should come from the same distribution
- Dataset might be split to not include a test set
  - Dev set can be used to get to a “good” model
  - Since data is fit to the dev set, there is no unbiased estimate of performance
  - When data doesn’t include a test set, dev set is usually referred to as “test” set
  - Resulting model may overfit to the dev set

### 2.1.1 Bias and Variance

- In the deep learning era, there tends to be less of a discussion about the bias/variance trade off
- In 2 dimensions, data can be plotted to look for high bias or variance
  - High bias classifiers underfit the data
  - High variance classifiers overfit the data
- For higher dimensions, training set error and dev set error can be used
  - High variance classifier has low training error and high dev set error
  - High bias classifier has high training error and high dev set error
  - Classifier with high bias and high variance will have high training error and even higher dev set error
- Above ideas only work with the assumption that the optimal error is 0%
  - Training and dev set must also come from the same distribution

### 2.1.2 Basic Recipe for Machine Learning

- Train initial algorithm and reduce bias of algorithm to an “acceptable value”
  - Use a larger network
  - Train algorithm for longer

- Reduce variance of the algorithm by getting more data
  - Add regularization terms to the cost function
- Bias and variance can also be reduced by using a more appropriate NN architecture
- In the big data era, bias and variance can be reduced without affecting each other
  - Training a bigger network typically reduce the bias
  - Getting more training data will typically reduce the variance
- Using regularization will have a bias variance trade off

### 2.1.3 Regularization

- Adding regularization will usually help in reducing variance and prevent overfitting
  - Regularization will only affect how the weights change during backpropagation
  - For forward propagation, regularization has no effect
- For logistic regression:

$$J(w, b) = \frac{1}{m} \sum_{i=1}^m \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m} ||w||_2^2$$

$$\begin{aligned} ||w||_2^2 &= \sum_{j=1}^{n_x} w_j^2 \\ &= w^T w \end{aligned}$$

- Above method is  $L_2$  regularization after the  $L_2$  norm (Euclidean norm) of  $w$
- $b$  values can also be regularized but will have a much smaller effect than  $w$
- $L_1$  regularization adds the term:

$$\frac{\lambda}{m} \sum_{i=1}^{n_x} |w| = \frac{\lambda}{m} ||w||_1$$

- Using  $L_1$  regularization will result in  $w$  being sparse
- Can be seen to compressing the model
- $L_2$  regularization is more common for deep learning
- Regularization parameter  $\lambda$  will be set using the cross validation set
  - `lambda` is a reserved keyword in Python

- For a neural network:

$$J(w^{[1]}, b^{[1]}, \dots, w^{[L]}, b^{[L]}) = \frac{1}{m} \sum_{i=1}^m \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m} \sum_{l=1}^L \|w^{[l]}\|^2$$

$$\|w^{[L]}\|^2 = \sum_{i=1}^{n^{[L]}} \sum_{j=1}^{n^{[L-1]}} (w_{i,j}^{[L]})^2$$

–  $\|W^{[l]}\|_F^2$  known as the Frobenius norm of the matrix

- Since new term added to cost function,  $\frac{\partial J}{\partial W^{[l]}}$  will be different

$$dW^{[l]} = \dots + \frac{\lambda}{m} W^{[l]}$$

$$W^{[l]} = W^{[l]} - \frac{\alpha \lambda}{m} W^{[l]} - \alpha(\dots)$$

– Also known as weight decay as value of  $W$  will decrease on every iteration

$$W^{[l]} - \frac{\alpha \lambda}{m} W^{[l]} = \left(1 - \frac{\alpha \lambda}{m}\right) W^{[l]}$$

– Value of  $\left(1 - \frac{\alpha \lambda}{m}\right)$  will be slightly less than 1

- Adding regularization term will penalize the weight matrix from being too large
  - As the value of  $\lambda$  is increased, the weights in  $w$  will get closer to 0
  - Each hidden unit will have a smaller effect and the resulting NN will be simpler
- When using the tanh function, penalizing  $w$  will make  $z^{[l]}$  smaller
  - For a small  $z^{[l]}$ , tanh function is roughly linear
  - If all hidden units in the network are roughly linear, the result of the NN will also be roughly linear

## Dropout Regularization

- Each layer in the NN has a probability of eliminating a node
  - When a node is eliminated, all outgoing links from the node are also deleted
  - Each example will be trained on a smaller network so will have less chance of overfitting
- For each different training example, the NN is reset and randomly eliminates nodes again
- Inverted dropout:

```
d3 = np.random.rand(a3.shape[0], a3.shape[1]) < keep_prob
a3 = np.multiply(a3, d3)
a3 /= keep_prob
```

- For **keep\_prob** = 0.8 each node has a 0.2 chance of being removed
- Activation values should be scaled by **keep\_prob** so the expected value of  $z$  can stay constant
- On each pass through the training set, a different set of units should be zeroed out
- At test time, dropout should not be used as it will create noise in the predictions
- A single hidden unit cannot rely on a specific feature as it may not be used on each iteration
  - Weights for the unit will be spread out between the units
  - Has the same effect as shrinking the weights like L2 regularization
  - The equivalent L2 penalty on different weights depends on the size of the activations being used for the weight
- **keep\_prob** can be varied between the layers
  - Larger layers may be more prone to overfitting and can have a larger **keep\_prob**
  - For small layers with a very small chance of overfitting, **keep\_prob** can be set to 1
- Many dropout implementations started with computer vision
  - Input size for computer vision is extremely large
- Cost function is not well defined when dropout is used
  - Can set **keep\_prob** to 1 and check for monotonically decreasing  $J$
  - When  $J$  is decreasing, then can reduce the value of **keep\_prob** to use dropout

## Other Regularization Methods

- Getting more training data will almost always help overfitting
  - May not be possible to get more training data or very expensive
- Data augmentation will create new examples and can help reduce overfitting
- For an image dataset:
  - Flipping the image horizontally

- Randomly cropping and distorting the image
- Magnitude of image transformation depends on classifier
  - For a cat dataset, image should not be flipped vertically
  - For OCR, distortions and rotations can be slightly more extreme



- Early stopping can be used to prevent overfitting from happening
  - If the NN is overfitting the data, the dev set error will initially decrease before increasing
  - Training of the NN can be stopped when the dev set error is lowest and the data has not been overfit
- Using early stopping links the task of optimizing  $J$  and not overfitting the data
  - Early stopping will prevent the cost function from being optimized
- L2 regularization is a better method to prevent overfitting
  - Requires a choice for the value of  $\lambda$  and is much more computationally expensive

#### 2.1.4 Setting up the Optimization Problem

- Normalization can be used to speed up the training of a NN
  - Subtract the mean:

$$\mu = \frac{1}{m} \sum_{i=1}^m x^{(i)}$$

$$x := x - \mu$$

- Normalize the variance:

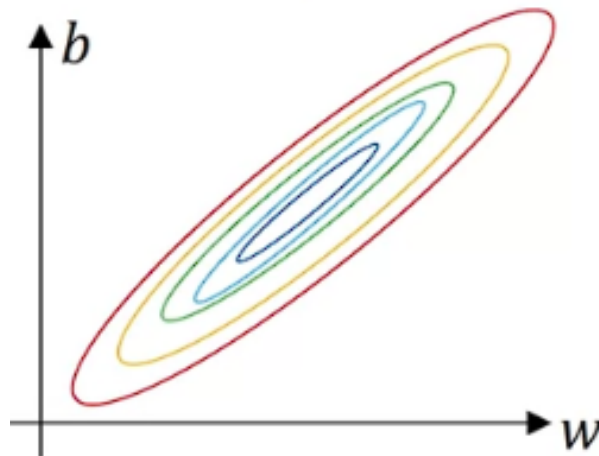
$$\sigma^2 = \frac{1}{m} \sum_{i=1}^m (x^{(i)} - \mu)^2$$

$$x := (x - \mu) / \sigma$$

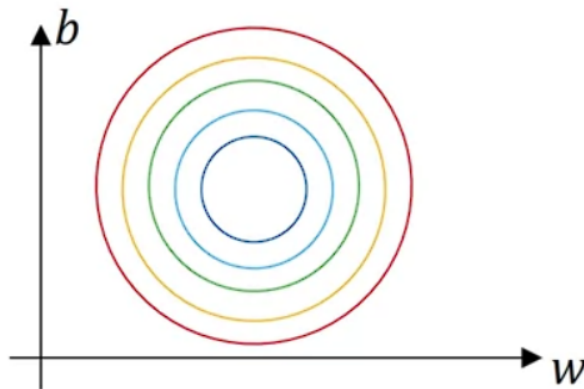
- When normalizing a training set, test set and training set should be processed together



- All the data must go through the same transformation
- For data that is not normalized, the cost function will be very elongated
  - The gradient will be quite shallow and will take longer to converge
  - Algorithm will require a smaller learning rate



- On average, normalized data will have a cost function that is more symmetric
  - Gradient descent will converge faster and can use a larger learning rate



### Vanishing/Exploding Gradients

- For very deep neural networks, the derivatives can get exponentially big or small
- If the weights of a NN are all the same, the prediction  $\hat{y}$  will  $x$  to the  $L$ th power
  - For  $W^{[l]} > I$  the gradient will explode
  - For  $W^{[l]} < I$  the gradient will vanish

- Some modern applications use 152 layer NN
  - Require careful initialization of the weights to ensure correct training
- For a single neuron:
  - The output  $\hat{y}$  will be the sum of all  $w_i x_i$

$$z = w_1 x_1 + w_2 x_2 + \dots + w_n x_n$$

- For a large  $n$ , want a smaller  $w_i$
- Want  $\text{Var}(w_i) = \frac{1}{n}$

$$W^{[l]} = \text{np.random.randn(shape)} * \text{np.sqrt}\left(\frac{1}{n^{[l-1]}}\right)$$

- Variance of Gaussian random variable can be set by multiplying by sqrt tem
- For ReLU activation function, the variance should be set to  $\frac{2}{n}$ 
  - tanh activation uses Xavier initialization  $\frac{1}{n^{[l-1]}}$
  - Yoshua Bengio multiplied random variable by  $\sqrt{\frac{2}{n^{[l-1]} + n^{[l]}}}$
- Initialization of weights aims to set weight matrices close to 1
  - Helps to prevent  $\hat{y}$  from vanishing or exploding too quickly
- Variance parameter can be tuned as another hyperparameter

## Gradient Checking

- Can be used to ensure implementation of backpropagation is correct
- Requires numerical approximations of gradients
  - For a function  $f$  at a point  $\theta$ , gradient can be approximated by looking at  $\theta + \epsilon$  and  $\theta - \epsilon$
  - Approximation is closer when double sided estimate is used
- If  $g$  is the derivative of  $f$ :

$$g(\theta) \approx \frac{f(\theta + \epsilon) - f(\theta - \epsilon)}{2\epsilon}$$

- Using the 2 sided difference will give a much better estimate but is more computationally expensive

- The derivative of a function at a point is the limit of the numerical approximation

$$f'(\theta) = \lim_{\epsilon \rightarrow 0} \frac{f(\theta + \epsilon) - f(\theta - \epsilon)}{2\epsilon}$$

- For a non 0 value of  $\epsilon$ , the error of the approximation is  $O(\epsilon^2)$
- For the single sided numerical approximation, the error is  $O(\epsilon)$
- To perform gradient checking on a NN:
  1. Reshape  $W^{[1]}, b^{[1]}, \dots, W^{[L]}, b^{[L]}$  into a single vector  $\theta$
  2. Reshape  $dW^{[1]}, db^{[1]}, \dots, dW^{[L]}, db^{[L]}$  into a single vector  $d\theta$
  3. For every  $i$  in  $\theta$ , calculate:

$$d\theta_{approx}[i] = \frac{J(\theta_1, \theta_2, \dots, \theta_i + \epsilon) - J(\theta_1, \theta_2, \dots, \theta_i - \epsilon)}{2\epsilon}$$

4. Check if  $d\theta_{approx}$  and  $d\theta$  are reasonably close to each other

For  $\epsilon = 10^{-7}$ :

$$\frac{\|d\theta_{approx} - d\theta\|_2}{\|d\theta_{approx}\|_2 + \|d\theta\|_2} \approx 10^{-7}$$

- Grad check should be only be used when debugging
  - Calculating  $d\theta_{approx}$  is very computationally expensive
- If regularization is used, correct cost function must be used to calculate the gradient
- If dropout is used,  $J$  is not well defined and cannot use grad check
  - Cost function  $J$  that is optimized by dropout is defined by summing over all subsets of nodes that could be eliminated on each iteration
  - Can implement grad check with a `keep_prob` of 1 before turning on dropout
- Implementation of gradient descent may be correct when  $W$  and  $b$  are close to 0
  - Can run grad check just after random initialization
  - After training the network for a number of iterations, can run grad check again

## 2.2 Optimization Algorithms

### 2.2.1 Mini Batch Gradient Descent

- For gradient descent, vectorization will allow computation over all  $m$  training examples
  - If  $m$  is very large, then vectorization will still be very slow
- Gradient descent requires the whole training set to be processed for a single step of gradient descent

- Data from training set can be split into mini batches

$$X^{\{1\}} = [x^{(1)}, x^{(2)}, \dots, x^{(1000)}]$$

$$Y^{\{1\}} = [y^{(1)}, y^{(2)}, \dots, y^{(1000)}]$$

- Mini batch gradient descent looks at one mini batch on each iteration of gradient descent
- For each mini batch in the training set:

- Run forward propagation on  $X^{\{t\}}$

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

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

...

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

- Compute cost:  $J^{\{t\}} = \frac{1}{1000} \sum_{i=1}^l \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2 \times 1000} \sum_l \|W^{[l]}\|_F^2$
- Use backpropagation to calculate gradients wrt  $J^{\{t\}}$
- Update weights

$$W^{[l]} := W^{[l]} - \alpha dW^{[l]}$$

$$b^{[l]} := b^{[l]} - \alpha db^{[l]}$$

- A single pass through the training set is known as an epoch
- Algorithm can continue to run for multiple passes through the training set until an optimal solution is found
- For batch gradient descent, the cost should decrease on each iteration
  - If the cost doesn't decrease per iteration, then the algorithm has a bug
- For mini batch gradient descent, the cost will trend downwards but will be more noisy
  - Algorithm is being trained on a different batch of results on each iteration
- When running mini batch gradient descent, must choose the size of the mini batch
  - For mini batch size =  $m$ : Batch gradient descent
  - For mini batch size = 1: Stochastic gradient descent
- For stochastic gradient descent, each example may be good or bad for gradient descent
  - On average the cost function will be minimized for gradient descent
  - Path taken by gradient descent will be very noisy

- Stochastic gradient descent will never converge and just oscillate around the minimum
- Choice of mini batch size should be between 1 and  $m$ 
  - Batch gradient descent will take very long for a single iteration
  - Stochastic gradient descent will lose all the speed from vectorization
- For a small training set ( $m \leq 2000$ ), can just use gradient descent
- Otherwise can try a mini batch size from 64-512
  - Code may run faster if the mini batch size is a power of 2
- A single mini batch should be able to fit in the whole CPU/GPU memory

### **Advanced Optimization Algorithms**

- Some advanced algorithms require the use of exponentially weighted averages