Deep Learning Specialization : Course 1-2

Student
Riahi LOURIZ

Teacher
Andrew Ng

August 17, 2018

Contents

Introduction

This document is a summary of the deep learning specialization performed by Andrew Ng. It does not contain only my personal notes, but also others notes. You would find in the last section all resourcesI have used to perform this document. I wish you will enjoy it. If you enjoy it, do not forget it to share it!

About This Specialization (From the official Deep Learning Specialization page) :

If you want to break into AI, this Specialization will help you do so. Deep Learning is one of the most highly sought after skills in tech. We will help you become good at Deep Learning.

In five courses, you will learn the foundations of Deep Learning, understand how to build neural networks, and learn how to lead successful machine learning projects. You will learn about Convolutional networks, RNNs, LSTM, Adam, Dropout, BatchNorm, Xavier/He initialization, and more. You will work on case studies from healthcare, autonomous driving, sign language reading, music generation, and natural language processing. You will master not only the theory, but also see how it is applied in industry. You will practice all these ideas in Python and in TensorFlow, which we will teach.

You will also hear from many top leaders in Deep Learning, who will share with you their personal stories and give you career advice.

AI is transforming multiple industries. After finishing this specialization, you will likely find creative ways to apply it to your work.

We will help you master Deep Learning, understand how to apply it, and build a career in AI.

1 Course 1 : Neural Networks and Deep Learning

Course summary (taken from coursera) :
If you want to break into cutting-edge AI, this course will help you do so. Deep learning engineers are highly sought after, and mastering deep learning will give you numerous new career opportunities. Deep learning is also a new "superpower" that will let you build AI systems that just weren't possible a few years ago.

In this course, you will learn the foundations of deep learning. When you finish this class, you will:

- Understand the major technology trends driving Deep Learning
- Be able to build, train and apply fully connected deep neural networks
- Know how to implement efficient (vectorized) neural networks
- Understand the key parameters in a neural network's architecture

This course also teaches you how Deep Learning actually works, rather than presenting only a cursory or surface-level description. So after completing it, you will be able to apply deep learning to a your own applications. If you are looking for a job in AI, after this course you will also be able to answer basic interview questions.

1.1 Week 1 : Introduction to deep learning

Be able to explain the major trends driving the rise of deep learning, and understand where and how it is applied today.

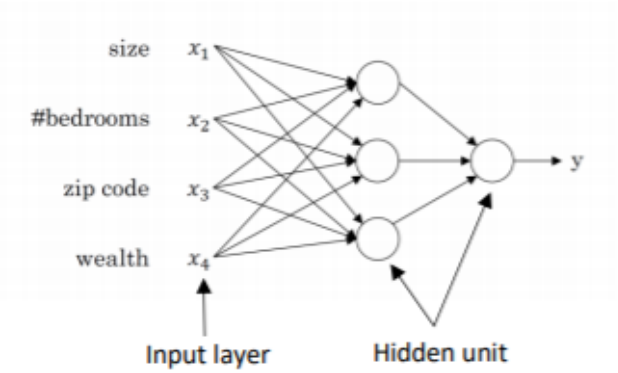
1.1.1 What is a Neural Netwrok (NN)?

It is a powerful learning algorithm inspired by how the brain works.

Example 1 - single neural network :
Given data about the size of houses on the real estate market and you want to fit a function that will predict their price. It is a linear regression problem because the price as a function of size is a continuous output.
We know the prices can never be negative so we are creating a function called Rectified Linear Unit (ReLU) which starts at zero.

1. The input is the size of the house (x)
2. The neuron implements the function ReLU[1] (blue line)
3. The output is the price (y).

Example 2 - Multiple neural network :
The price of a house can be affected by other features such as size, number of bedrooms, zip code and wealth. The role of the neural network is to predicted the price and it will automatically generate the hidden units. We only need to give the inputs x and the output y.



Multiple neural network are very efficient in feature engineering. For the abovre NN, the hidden layer would contain for example : familiy size (by combining size and no of bedrooms), walkability (by using Zipcode) and School Quality (by combining ZipCodes and wealth).

1.1.2 Supervised learning with neural networks

In supervised learning, we are given a data set and already know what our correct output should look like, having the idea that there is a relationship between the input and the output.

[1]RELU stands for rectified linear unit is the most popular activation function right now that makes deep NNs train faster.

Supervised learning problems are categorized into "regression" and "classification" problems. In a regression problem, we are trying to predict results within a continuous output, meaning that we are trying to map input variables to some continuous function. In a classification problem, we are instead trying to predict results in a discrete output. In other words, we are trying to map input variables into discrete categories.

Here are some examples of supervised learning :

	Input(x)	*Output*(y)	*Application*
(1)	Home features	Price	Real Estate
(2)	ad, user info	Click on ad ?	Online Advertsing
(3)	Image	Object(1,...,1000)	Photo tagging
(4)	Audio	Text transcript	Speech recognition
(5)	English	Chinese	Machine transation
(6)	Image, Radar info	Position of other cars	Autonomous driving

There are different types of neural network, for example Standard NN that us useful for Structured data and Convolution Neural Network (CNN) used often for image application and Recurrent Neural Network (RNN) used for one-dimensional sequence data such as translating English to Chinses or a temporal component such as text transcript. As for the autonomous driving, it is a hybrid neural network architecture.

Structured vs unstructured data :
Structured data refers to things that has a defined meaning such as price, age whereas unstructured data refers to thing like pixel, raw audio, text.

Structured Data

Size	#bedrooms	...	Price (1000$s)
2104	3		400
1600	3		330
2400	3		369
⋮	⋮		⋮
3000	4		540

User Age	Ad Id	...	Click
41	93242		1
80	93287		0
18	87312		1
⋮	⋮		⋮
27	71244		1

Unstructured Data

Audio

Image

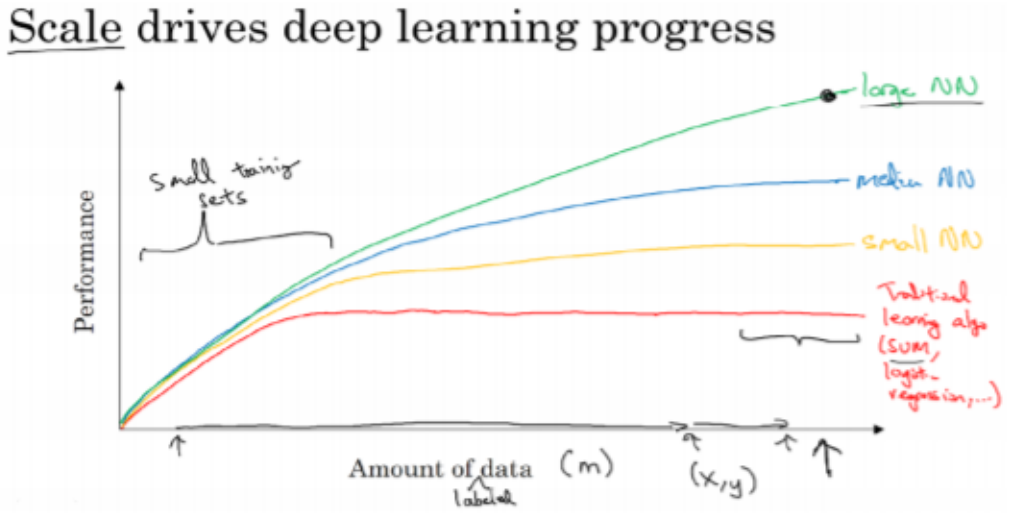
Four scores and seven years ago...

Text

1.1.3 Why is deep learning taking off ?

Deep learning is taking off due to a large amount of data available through the digitization of the society,faster computation and innovation in the development of neural network algorithm. As just described, Deep learning is taking off for 3 reasons :

1. Data :



   Using the above figure, we can conclude the following :

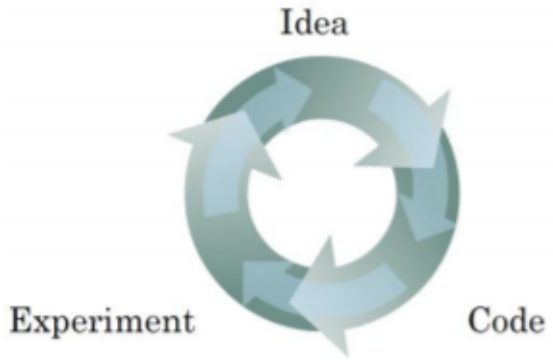
   - For small data NN can perform as Linear regression or SVM (Support vector machine)
   - For big data a small NN is better than SVM, etc
   - For big data a big NN is better than a medium NN that is better than small NN.
   - Hopefully we have a lot of data because the world is using the computer a little bit more (mobiles, IOT, etc).
   - Two things have to be considered to get to the high level of performance :

     (a) Being able to train a big enough neural network

     (b) Huge amount of labeled data

2. Computation : including GPU, Powerful CPU, Distributed computing(HDFS, Spark, MapReduce, multiprocessing etc).
3. Algorithms : Creative algorithms has appeared that changed the way NN works.
   For example using RELU function is so much better than using SIGMOID function in training a NN because it helps with the vanishing gradient problem.

The process of training a neural network is iterative :

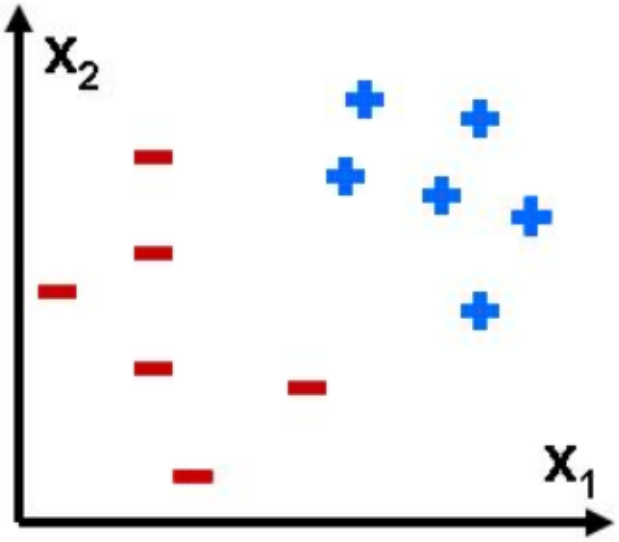


It could take a good amount of time to train a neural network, which affects your productivity. Faster computation helps to iterate and improve new algorithm.

1.2 Week 2 : Neural Networks Basics

Learn to set up a machine learning problem with a neural network mindset. Learn to use vectorization to speed up your models.

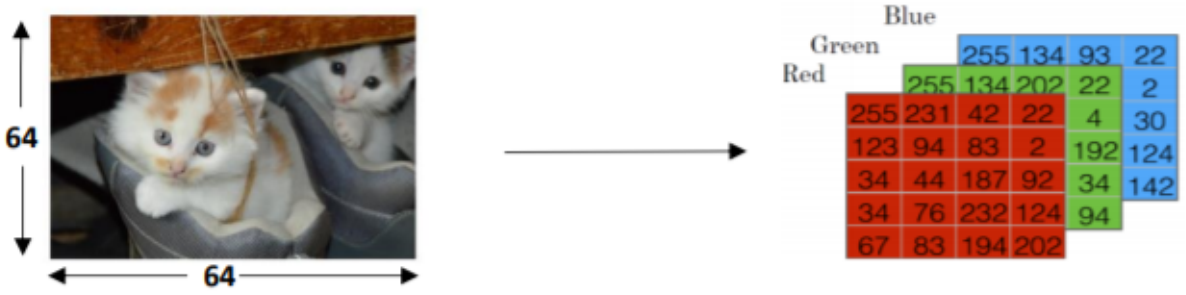
1.2.1 Binary Classification



In a binary classification problem, the result is a discret value output.
For example :

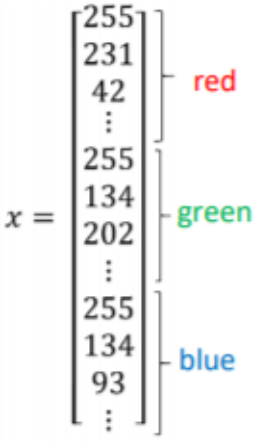
- account hacked (1) or compromised (0)
- a tumor malign (1) or benign (0)

Example - Cat vs Non Cat :
The goal is to train a classifier that the input is an image represented by a feature vector, x , and predicts whether the corresponding label is 1 or 0. In this case, whether this is a cat image (1) or a non-cat image (0).



An image is stored in the computer in three separate matrices corresponding to the Red, Green, and Blue color channels of the image (RGB). The three matrices have the same size as the image, for example, the resolution of the cat image is 64 pixels X 64 pixels, the three matrices (RGB) are 64 X 64 each.
The value in a cell represents the pixel intensity which will be used to create a feature vector of n-dimension. In pattern recognition and machine learning, a feature vector represents an object, in this case, a cat or no cat.

To create a feature vector, , the pixel intensity values will be "unroll" or "reshape" for each color. The dimension of the input feature vector x is n_x = 64x64x3 = 12 288.



NB : for standard notations for Deep Learning, please check the last section of this document.

1.2.2 Logistic regression

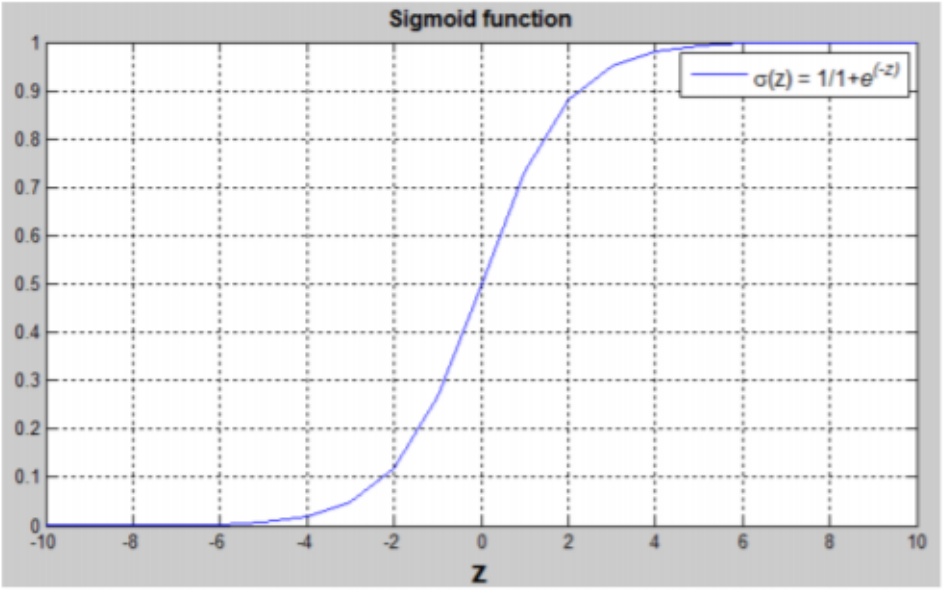
Logistic regression is a learning algorithm used in a supervised learning problem when the output are all either zero or one. The goal of logistic regression is to minimize the error between its predictions and training data.

Example: Cat vs No - cat :
Given an image represented by a feature vector , the algorithm will evaluate the probability of a cat being in that image.

$$Given\ x,\ \hat{y} = P(y = 1|x),\ where\ 0 \leq \hat{y} \leq 1$$

The parameters used in Logistic regression are :

- The input features vector: $x \in \mathbb{R}^{n_x}$, where n_x is the number of features
- The training label: $y \in 0, 1$
- The weights: $w \in \mathbb{R}^{n_x}$, where n_x is the number of features
- The bias: $b \in \mathbb{R}$
- The output: $\hat{y} = \sigma(W^T x + b)$
- Sigmoid function : $s = \sigma(W^T x + b) = \sigma(z) = \frac{1}{1+\exp(-z)}$



$\sigma(W^T x+b)$) is a linear function $(ax+b)$, but since we are looking for a probability constraint between

[0,1], the sigmoid function is used. The function is bounded between [0,1] as shown in the graph above.

Some observations from the graph:

- If z is a large positive number, then $\sigma(z) = 1$
- If z is small or large negative number, then $\sigma(z) = 0$
- If z =0, then $\sigma(z) = 0.5$

1.2.3 Logistic Regression : Cost Function

To train the parameters w and b, we need to define a cost function.
Recap :
$\hat{y}^{(i)} = \sigma(W^T x^{(i)} + b)$, where $\sigma(z^{(i)}) = \frac{1}{1+\exp(-z^{(i)})}$
Given $(x^{(1)}, y^{(1)}), ..., (x^{(m)}, y^{(m)})$

Loss (error) function :
The loss function measures the discrepancy between the prediction $\hat{y}^{(i)}$ and the desired output $y^{(i)}$. In other words, the loss function computes the error for a single training example.

$L(\hat{y}^{(i)}, y^{(i)}) = \frac{1}{2}(\hat{y}^{(i)} - y^{(i)})^2$: But we won't use this notation because it leads us to optimization problem which is non convex, means it contains local optimum points.
$L(\hat{y}^{(i)}, y^{(i)}) = -(y^{(i)} \log(\hat{y}^{(i)}) + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)}))$

- if $y^{(i)} = 1$: $L(\hat{y}^{(i)}, y^{(i)}) = -\log(\hat{y}^{(i)})$ where $\log(\hat{y}^{(i)})$ and $y^{(i)}$ should be close to 1.
- if $y^{(i)} = 0$: $L(\hat{y}^{(i)}, y^{(i)}) = -\log(1 - \hat{y}^{(i)})$ where $\log(1 - \hat{y}^{(i)})$ and $y^{(i)}$ should be close to 0.

Cost function :
The cost function is the average of the loss function of the entire training set. We are going to find the parameters w and b that minimize the overall cost function :

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

This cost function is convex. This fact will help us in gradient descent.

Math behind Logistic Regression :

We put the general definition :

$$p(y/x) = \hat{y}^y * (1-\hat{y})^{(1-y)} = \begin{cases} \hat{y} & \text{if } y = 1 \\ 1-\hat{y} & \text{if } y = 0 \end{cases}$$

Consequently :

$$\begin{aligned} \log(p(y/x) &= y * \log(\hat{y}) + (1-y) * log(1-\hat{y}) && (1) \\ &= -L(\hat{y}, y) && (2) \end{aligned}$$

Obviously, $\log(p(y/x))$ must be maximized. Indeed, in both cases $(y = 1, y = 0)$, The probability $p(y/x)$ must be close to $1 \Rightarrow L(\hat{y}, y)$ must be minimized

So, fo m examples which are independent and identically distributed (so verify $p(A \bigcap B) = p(A) * p(B)$) :

$$\begin{aligned} p(examples) &= \prod_{i=1}^{m} p(y^(i)/x^(i)) \\ log(p(examples)) &= \sum_{i=1}^{m} \log(p(y^(i)/x^(i))) \end{aligned}$$

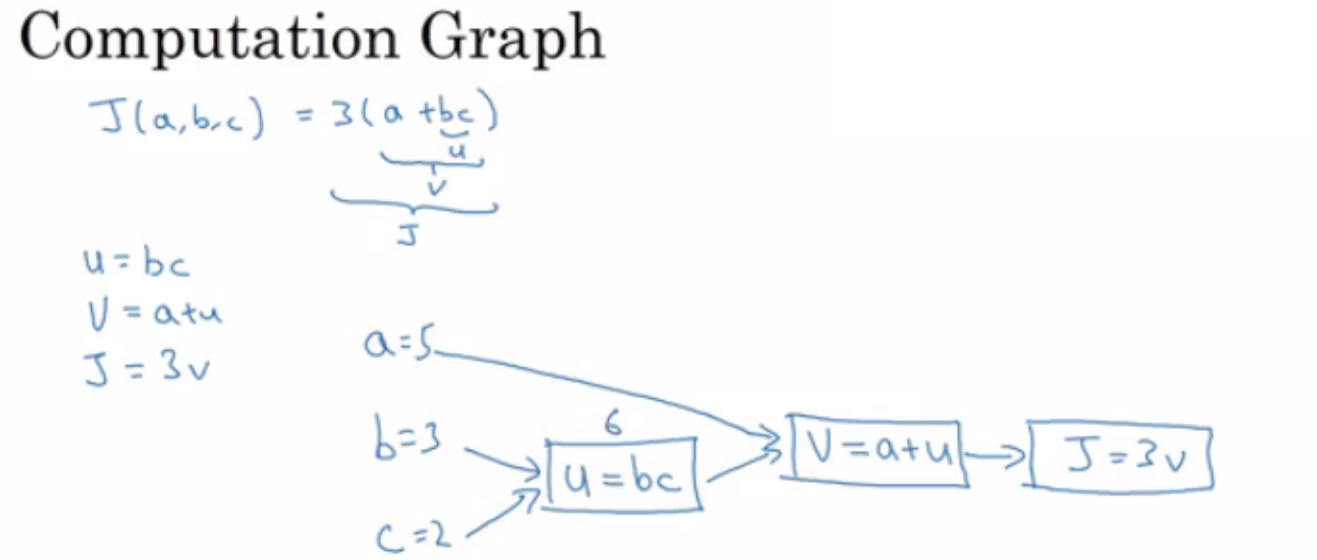
Finally :

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

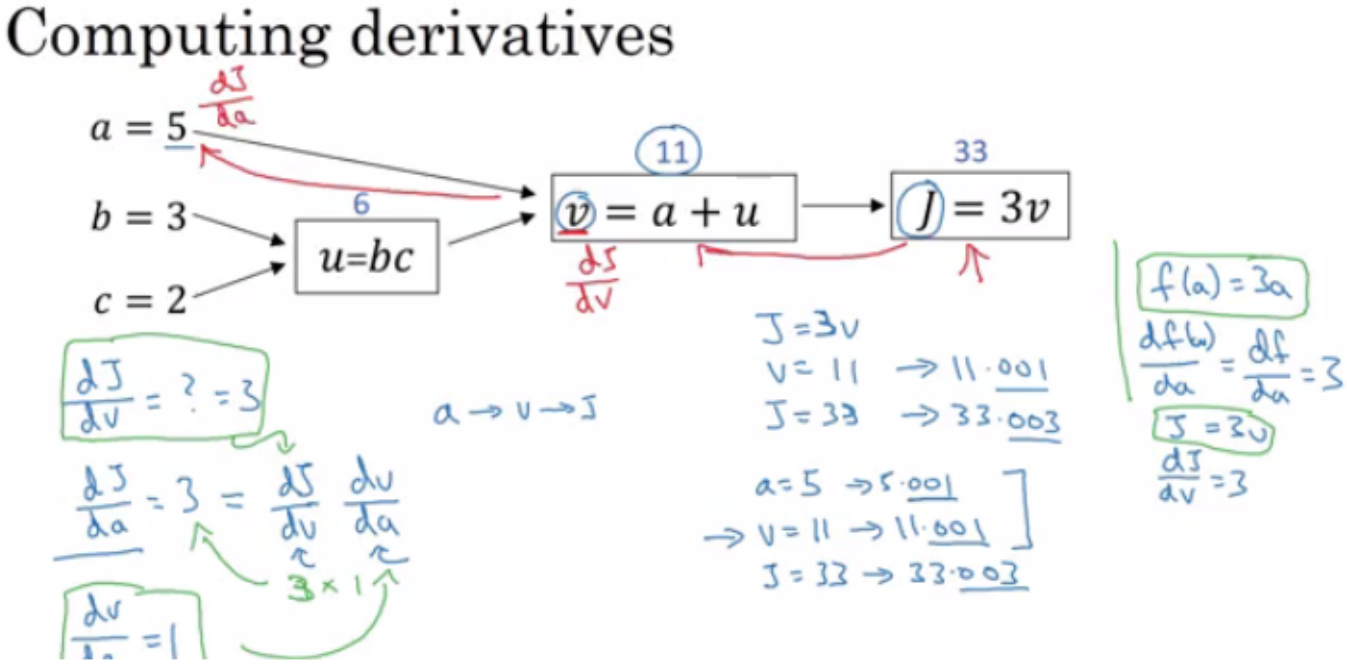
Gradient Descent :

- We want to find w and b that minimize the cost function.
- First we initialize w and b to 0,0 or initialize them to a random value in the convex function and then try to improve the values the reach minimum value.
- In Logistic regression people always use 0,0 instead of random.
- The gradient decent algorithm repeats:$w = w - \alpha * dw$ where α is the learning rate and dw is the derivative of w (Change to w) The derivative is also the slope of w
- The actual equations we will implement:
  - $w = w - \alpha * \frac{\partial J(w,b)}{\partial w}$: (how much the function slopes in the w direction)
  - $b = b - \alpha * \frac{\partial J(w,b)}{\partial b}$: (how much the function slopes in the b direction)

Computation graph Its a graph that organizes the computation from left to right.

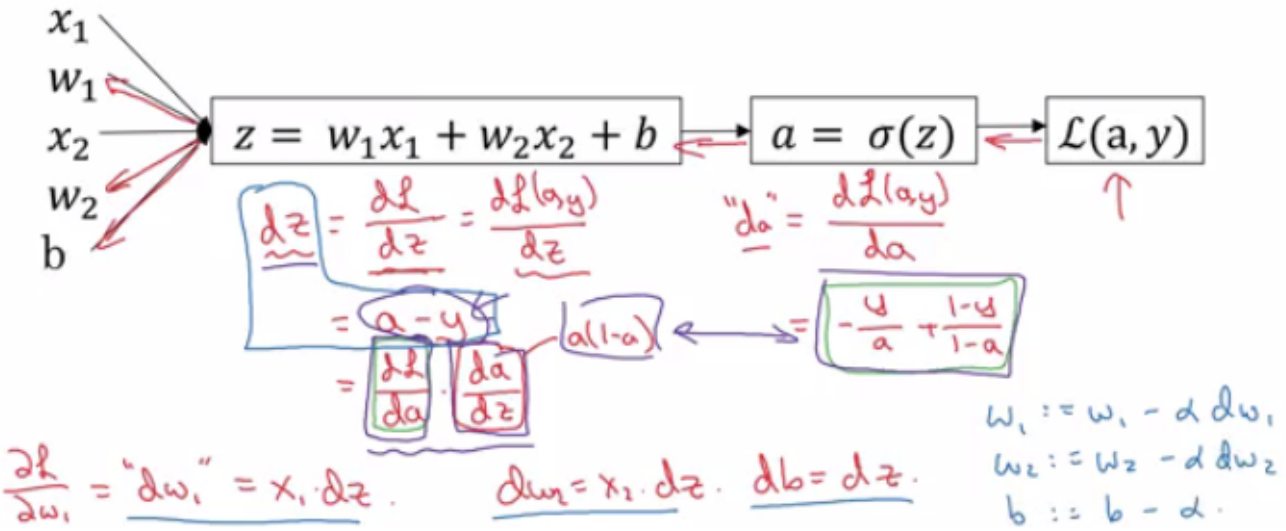


Derivatives with a computation graph :
Here is an example from the videos :



- We compute the derivatives on a graph from right to left and it will be a lot more easier.
- *dvar* means the derivatives of a final output (here it is the cost function) variable with respect to various intermediate quantities ($w1$,$w2$, etc)

Logistic Regression Gradient Descent :



Gradient Descent on m Examples :
Let's consider the given NN :

- $x1$ and $x2$: two features
- $w1$ and $w2$: weights for $x1$ and $x2$
- B : Bias term
- m : the number of training examples
- $y(i)$: expected output for i

So we have :
$z(i) = x1 * w1 + x2 * w2 + b \Longrightarrow a(i) = \sigma(z(i)) \Longrightarrow l(a(i), y(i)) = -(y(i) * \log(a(i)) + (1 - y(i)) * \log(1 - a(i)))$

Then from the right to left we will calculate derivations compared to the result :

- $\frac{\partial l}{\partial a} = da = -(\frac{y}{a}) + (\frac{1-y}{1-a})$
- $\frac{\partial l}{\partial z} = dz = \frac{\partial l}{\partial a} * \frac{\partial a}{\partial z} = [-(\frac{y}{a}) + (\frac{1-y}{1-a})] * [a(1 - a)]$ (see 1.2.3 for sigmoid derivatives)
- $\frac{\partial l}{\partial w1} = dw1 = x1 * dz$
- $\frac{\partial l}{\partial w2} = dw2 = x2 * dz$

- $\frac{\partial l}{\partial b} = db = dz$

Details about sigmoid derivatives :

$$\begin{aligned}
\frac{d}{dz}\sigma(z) &= \frac{d}{dz}\left[\frac{1}{1+e^{-z}}\right] && (3)\\
&= \frac{d}{dz}\left(1+\mathrm{e}^{-z}\right)^{-1} && (4)\\
&= -(1+e^{-z})^{-2}(-e^{-z}) && (5)\\
&= \frac{e^{-z}}{(1+e^{-z})^2} && (6)\\
&= \frac{1}{1+e^{-z}}\cdot\frac{e^{-z}}{1+e^{-z}} && (7)\\
&= \frac{1}{1+e^{-z}}\cdot\frac{(1+e^{-z})-1}{1+e^{-z}} && (8)\\
&= \frac{1}{1+e^{-z}}\cdot\left(\frac{1+e^{-z}}{1+e^{-z}}-\frac{1}{1+e^{-z}}\right) && (9)\\
&= \frac{1}{1+e^{-z}}\cdot\left(1-\frac{1}{1+e^{-x}}\right) && (10)\\
&= \sigma(z)\cdot(1-\sigma(z)) && (11)
\end{aligned}$$

Finally, the pseudo code (taken from videos (PS : the for loop is from 1 to m (not n))) :

$J = 0,\ dw1 = 0,\ dw2 = 0,\ db = 0$

for i = 1 to n:

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

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

$J \mathrel{+}= -\left[y^{(i)} \log \hat{y}^{(i)} + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)})\right]$

$dz^{(i)} = a^{(i)}(1 - a^{(i)})$

$dw_1 \mathrel{+}= x_1^{(i)} dz^{(i)}$

$dw_2 \mathrel{+}= x_2^{(i)} dz^{(i)}$

$db \mathrel{+}= dz^{(i)}$

$J = J/m,\ dw_1 = dw_1/m,\ dw_2 = dw_2/m,\ db = db/m$

- The above code should run for some iterations to minimize the error.
- So there will be two inner loops to implement the logistic regression.
- Vectorization is so important on deep learning to reduce loops. In the last code we can make the whole loop in one step using vectorization !

Vectorization :

- Deep learning shines when the dataset are big. However for loops will make you wait a lot for a result. Thats why we need vectorization to get rid of some of our for loops.
- NumPy library (dot) function is using vectorization by default.
- The vectorization can be done on CPU or GPU thought the SIMD operation. But its faster on GPU.
- Whenever possible avoid for loops.
- Most of the NumPy library methods are vectorized version.

Vectorizing Logistic Regression :
Given a matrix X that its shape is $[n_x, m]$ and a matrix Y that its shape is $[n_y, m]$, we can have the vectorized implementation as following:

```
Z = np.dot(W.T,X) + b      # Vectorization, then broadcasting,
                           # Z shape is (1, m)
A = 1 / 1 + np.exp(-Z)     # Vectorization, A shape is (1, m)
```

```
dZ = A - Y                      # Vectorization, dZ shape is (1, m)
dw = np.dot(X, dz.T) / m        # Vectorization, dw shape is (n_x, 1)
db = dz.sum() / m               # Vectorization, dz shape is (1, 1)
```

1.2.4 All in All

The main steps for building a Neural Network are :

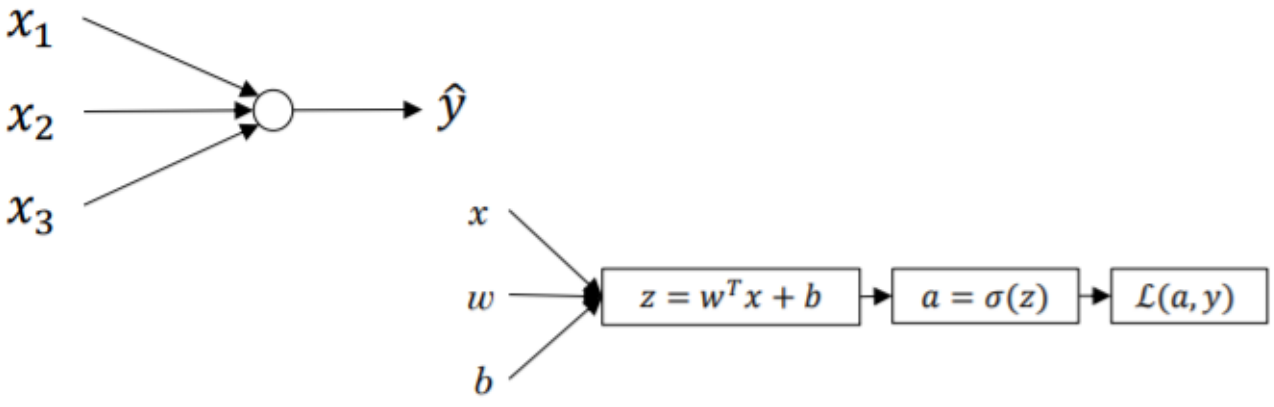
1. Define the model structure (such as number of input features)
2. Initialize the model's parameters
3. Loop:
    - Calculate current loss (forward propagation)
    - Calculate current gradient (backward propagation)
    - Update parameters (gradient descent)

1.3 Week 3 : Shallow neural networks

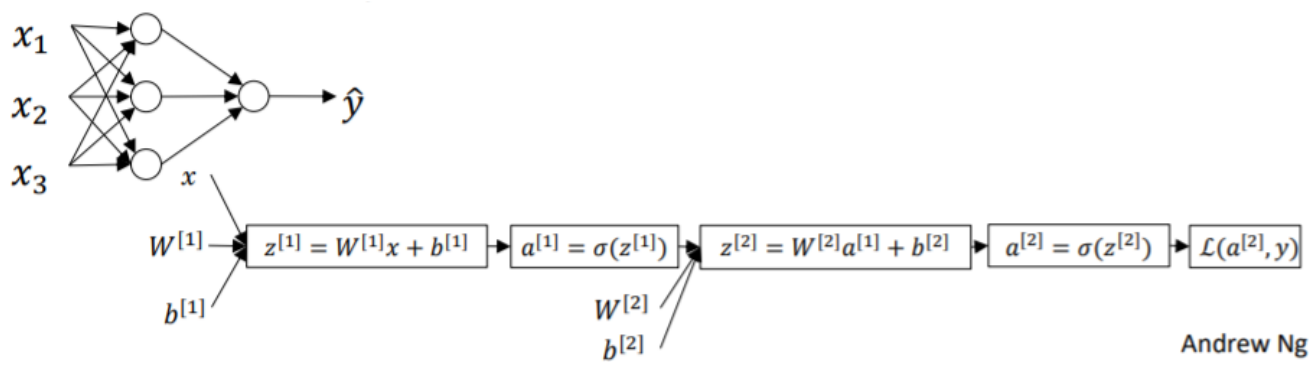
This week will help us to learn to build a neural network with one hidden layer, using forward propagation and backpropagation.

1.3.1 Neural Networks Overview

In logistic regression we had (taken from videos):



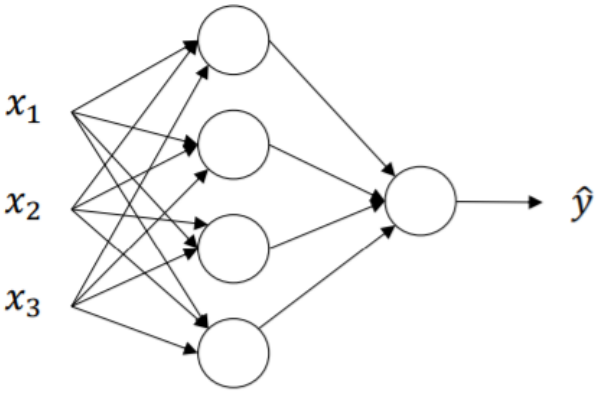
In neural networks with one layer we will have the following architecture : ((taken from videos))



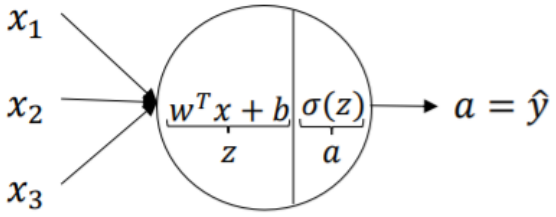
$\Rightarrow$ a Neural network is stack of logistic regression objects.

1.3.2 Neural Network Representation

Neural Network Representation :
Thie figure below is an overview of a Neural Network with one hidden layer.

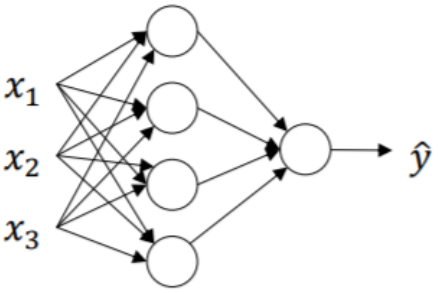


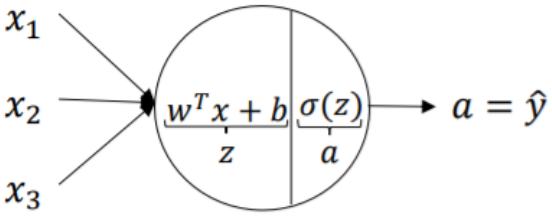
Computing a Neural Network's Output :
As we have said before, a NN is a stack of logistic regression objects, the following figures give all details about computing the output of a neural network :



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

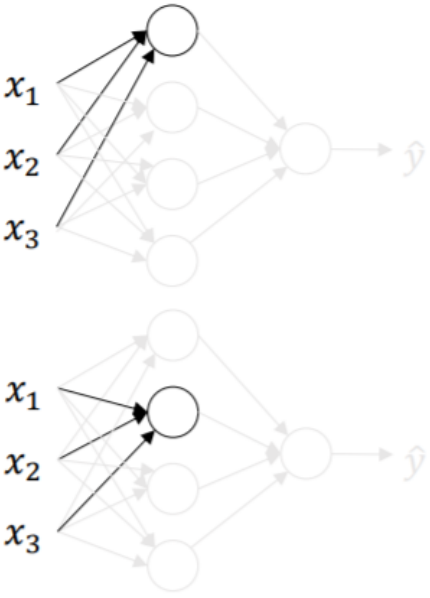
$$a = \sigma(z)$$

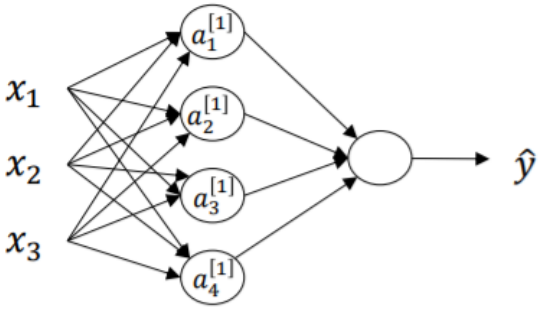




$$z = w^T x + b$$

$$a = \sigma(z)$$





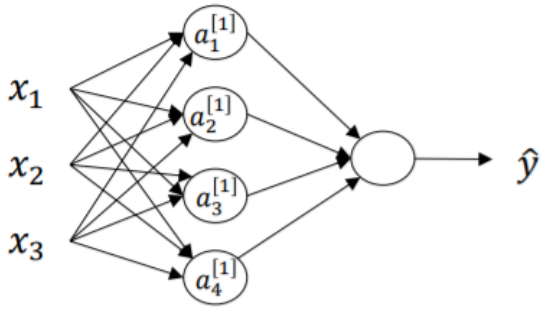
$$z_1^{[1]} = w_1^{[1]T} x + b_1^{[1]},\ a_1^{[1]} = \sigma(z_1^{[1]})$$

$$z_2^{[1]} = w_2^{[1]T} x + b_2^{[1]},\ a_2^{[1]} = \sigma(z_2^{[1]})$$

$$z_3^{[1]} = w_3^{[1]T} x + b_3^{[1]},\ a_3^{[1]} = \sigma(z_3^{[1]})$$

$$z_4^{[1]} = w_4^{[1]T} x + b_4^{[1]},\ a_4^{[1]} = \sigma(z_4^{[1]})$$

Learning Neural Network :
The process of learning a Neural Network is described as the following :



Given input x:

$$z^{[1]} = W^{[1]}x + b^{[1]}$$

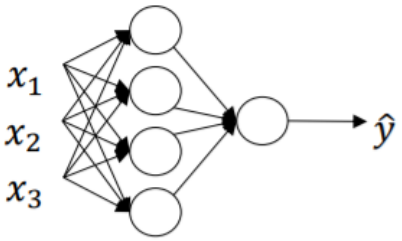
$$a^{[1]} = \sigma(z^{[1]})$$

$$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$$

$$a^{[2]} = \sigma(z^{[2]})$$

1.3.3 Vectorizing across multiple examples

For one example we perform the following calculations :

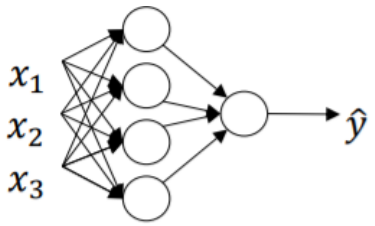


$$z^{[1]} = W^{[1]}x + b^{[1]}$$
$$a^{[1]} = \sigma(z^{[1]})$$
$$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$$
$$a^{[2]} = \sigma(z^{[2]})$$

Then, to do the same thing across all examples we will use a for loop :

```
for i = 1 to m:
```
$$z^{[1](i)} = W^{[1]}x^{(i)} + b^{[1]}$$
$$a^{[1](i)} = \sigma(z^{[1](i)})$$
$$z^{[2](i)} = W^{[2]}a^{[1](i)} + b^{[2]}$$
$$a^{[2](i)} = \sigma(z^{[2](i)})$$

But to do it efficiently, we will use a vectorized version :



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

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

```
for i = 1 to m
```

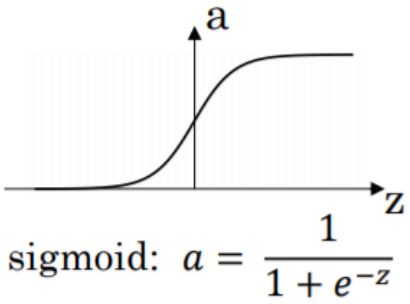
$$z^{[1](i)} = W^{[1]}x^{(i)} + b^{[1]}$$
$$a^{[1](i)} = \sigma(z^{[1](i)})$$
$$z^{[2](i)} = W^{[2]}a^{[1](i)} + b^{[2]}$$
$$a^{[2](i)} = \sigma(z^{[2](i)})$$

$$Z^{[1]} = W^{[1]}X + b^{[1]}$$
$$A^{[1]} = \sigma(Z^{[1]})$$
$$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$$
$$A^{[2]} = \sigma(Z^{[2]})$$

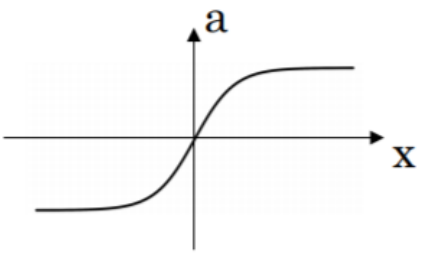
Andrew Ng

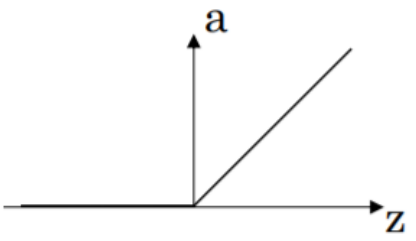
1.3.4 Activation functions

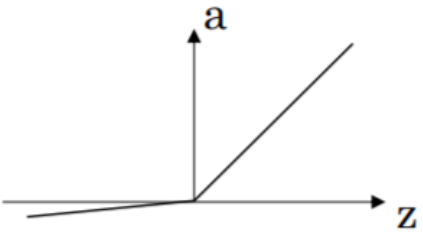
The most known activation function are : *sigmoid*, *tanh*, *ReLU* and *LeakyReLu*.



$$\text{sigmoid: } a = \frac{1}{1 + e^{-z}}$$







- It turns out that the tanh activation usually works better than sigmoid activation function for

hidden units because the mean of its output is closer to zero, and so it centers the data better for the next layer.
- Downsides of *tanh* and *sigmoid* functions is that when z is too large or too small their derivatives (slop) is very small which can make gradient descent very slow.
- One of the popular activation functions that solved the slow gradient decent is the *ReLU* function. $ReLU = max(0, z)$ so if z is negative the slope is 0 and if z is positive the slope remains linear.
- So here is some basic rule for choosing activation functions, if your classification is between 0 and 1, use the output activation as *sigmoid* and the others as *ReLU*.
- Leaky *ReLU* activation function different from *ReLU*. It works as *ReLU* but most people uses *ReLU*. $Leaky_{ReLU} = max(0.01z, z)$ the 0.01 can be a parameter for your algorithm.

Why non-linear activation functions ?

- If we have for all hidden layers only linear function[2] and for the output sigmoid function $\Rightarrow$ our NN is juts a logistic regression
- You might use linear activation function in one place - in the output layer if the output is real numbers (regression problem). But even in this case if the output value is non-negative you could use *ReLU* instead.

1.3.5 Derivatives of activation functions

The following derivations will be used in back propagation :

- Derivation of Sigmoid activation function : I have already done the derivative for sigmoid function in 1.2.3.
- Derivation of Tanh activation function :

$$g(z) = \frac{\exp^z - \exp^{-z}}{\exp^z + \exp^{-z}}$$
$$g'(z) = 1 - g(z)^2$$

- Derivation of ReLU activation function :

$$g(z) = np.maximum(0, z)$$
$$g'(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases}$$

- Derivation of leaky ReLU activation function :

$$g(z) = np.maximum(0.01 * z, z)$$
$$g'(z) = \begin{cases} 0.01 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases}$$

[2]linear activation functions will output linear activations

1.3.6 Gradient descent for Neural Networks

Architecture :
Before going deep into details about the gradient descent, let's recall the architecture of our Neural Network :

- $X(n_x, m) = A^0$: input data with m examples
- $Y(n_y, m)$: target variable
- $n[0] = n_x$: input features
- $n[1]$: number of hidden neurons
- $n[2]$: number of output neurons (in our case : $n[2] = 1$)
- $W^{[1]}(n[1], n[0])$: matrix of weights for the hidden layer
- $b^{[1]}(n[1], 1)$: bias term for the hidden layer
- $W^{[2]}(n[2], n[1])$: matrix of weights for the output layer
- $b^{[2]}(n[2], 1)$: bias term for the ouput layer
- $(g[i])$: activation function for the i^{th} layer

Forward propagation :
Let's recall the forward pass computation :

$$\begin{cases} Z^{[1]} &= W^{[1]}X + b^{[1]} \\ A^{[1]} &= \sigma(Z^{[1]}) \\ Z^{[2]} &= W^{[2]}A^{[1]} + b^{[2]} \\ A^{[2]} &= \sigma(Z^{[2]}) \end{cases}$$

$\Rightarrow J = J(W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}) = \frac{1}{m}\sum_{i=1}^{m} L(Y, A^{[2]})$, where $L(Y, A^{[2]})$ is the loss function.

Back propagation :

- $\frac{\partial J}{\partial Z^{[2]}} = dZ^{[2]} = A^{[2]} - Y$
- $\frac{\partial J}{\partial W^{[2]}} = dW^{[2]} = \frac{1}{m} dZ^{[2]} (A^{[2]})^T$
- $\frac{\partial J}{\partial b^{[2]}} = db^{[2]} = \frac{1}{m} np.sum(dZ^{[2]}, axis = 1, keepdims = True)$
- $\frac{\partial J}{\partial Z^{[1]}} = dZ^{[1]} = (W^{[2]})^T dZ^{[2]} * (g^{[1]})'(Z^{[1]})$
- $\frac{\partial J}{\partial W^{[1]}} = dW^{[1]} = \frac{1}{m} dZ^{[1]} X^T$
- $\frac{\partial J}{\partial b^{[1]}} = db^{[1]} = \frac{1}{m} np.sum(dZ^{[1]}, axis = 1, keepdims = True)$

Gradient Descent :
For a certain number of iterations :

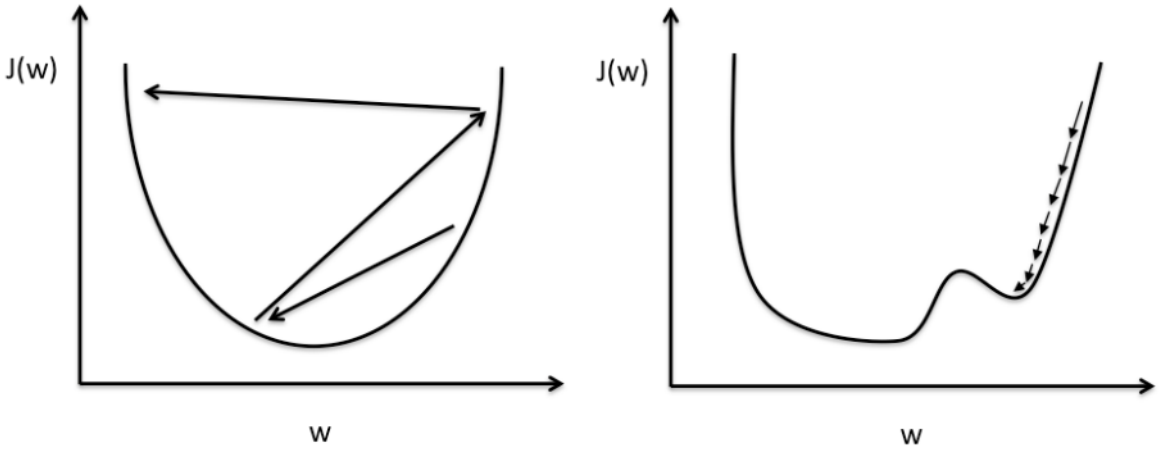
- (1). Forward propagation ⇒ Cost Function
- (2). Back propagation ⇒ Neural Network parameters ($W^{[1]}$, $b^{[1]}$, $W^{[2]}$ and $b^{[2]}$)
- (3). Upadate parameters :

$$\begin{cases} W^{[1]} &= W^{[1]} - learningrate * dW^{[1]} \\ b^{[1]} &= b^{[1]} - learningrate * db^{[1]} \\ W^{[2]} &= W^{[2]} - learningrate * dW^{[2]} \\ b^{[2]} &= b^{[2]} - learningrate * db^{[2]} \end{cases}$$

1.3.7 Random Initialization

- For Logistic regression, weights can be initialized to 0
- For NN, weights must not be initialized to 0 (bias term can be initialized with zero)
  - all hidden units will be completely identical (symmetric) - compute exactly the same function
  - on each gradient descent iteration all the hidden units will always update the same
- So that's why we use random initialization with small values.
  - $W^{[1]} = np.random.randn((n[1], n[0])) * 0.01$
  - $b^{[1]} = np.zeros((n[1], 1))$
  - $W^{[2]} = np.random.randn((n[2], n[1])) * 0.01$
  - $b^{[2]} = np.zeros((n[2], 1))$
- We need small values so that the gradient descent does not overshoot the global minimum.

The following figure illustrates the influence of the initialization (picture taken from the book Python Machine Learning, Sebastian Raschka).



Large learning rate: Overshooting.

Small learning rate: Many iterations until convergence and trapping in local minima.

1.4 Week 4 : Deep Neural Networks

Understand the key computations underlying deep learning, use them to build and train deep neural networks, and apply it to computer vision.

1.4.1 Deep L-layer neural network

A Deep NN is a NN with three or more layers, in contrary to Shallow NN which is a NN with one or two layers.
Let's start by defining some notations that will be used for the whole week :

- L : the number of layers (so $L-1$ is the number of hidden layers)
- $n[l]$: the number of neurons in a specific layer l ($n[0]$: number of input features, $n[L]$: number of neurons in output layer) $\Rightarrow n.shape = (1, L+1)$
- $g[l]$: activation function for the l^{th} layer $\Rightarrow g.shape = (1, L)$
- $W^{[l]}$: weights used for the l^{th} layer $\Rightarrow W^{[l]}.shape = (n[l], n[l-1])$
- $B^{[l]}$: bias for for the l^{th} layer $\Rightarrow B^{[l]}.shape = (n[l], 1)$

1.4.2 Forward Propagation in a Deep Network

Forward propagation for one input example :

$$\begin{cases} z^{[l]} &= W^{[l]}a^{[l-1]} + b^{[l]} \\ a^{[l]} &= g[l](z^{[l]}) \end{cases}$$

Forward propagation for the whole examples :

$$\begin{cases} Z^{[l]} &= W^{[l]}A^{[l-1]} + B^{[l]} \\ A^{[l]} &= g[l](Z^{[l]}) \end{cases}$$

Notice : It is ok to have a for loop for the forward propagation.

1.4.3 Getting your matrix dimensions right

Recall that :

- $W^{[l]}.shape = (n[l], n[l-1])$
- $B^{[l]}.shape = (n[l], 1)$
- $dW.shape = W.shape$, $dB.shape = B.shape$
- shape of $Z^{[l]}, A^{[l]}, dZ^{[l]}$ and $dA^{[l]}$ is $(n[l], m)$

1.4.4 Why deep representations?

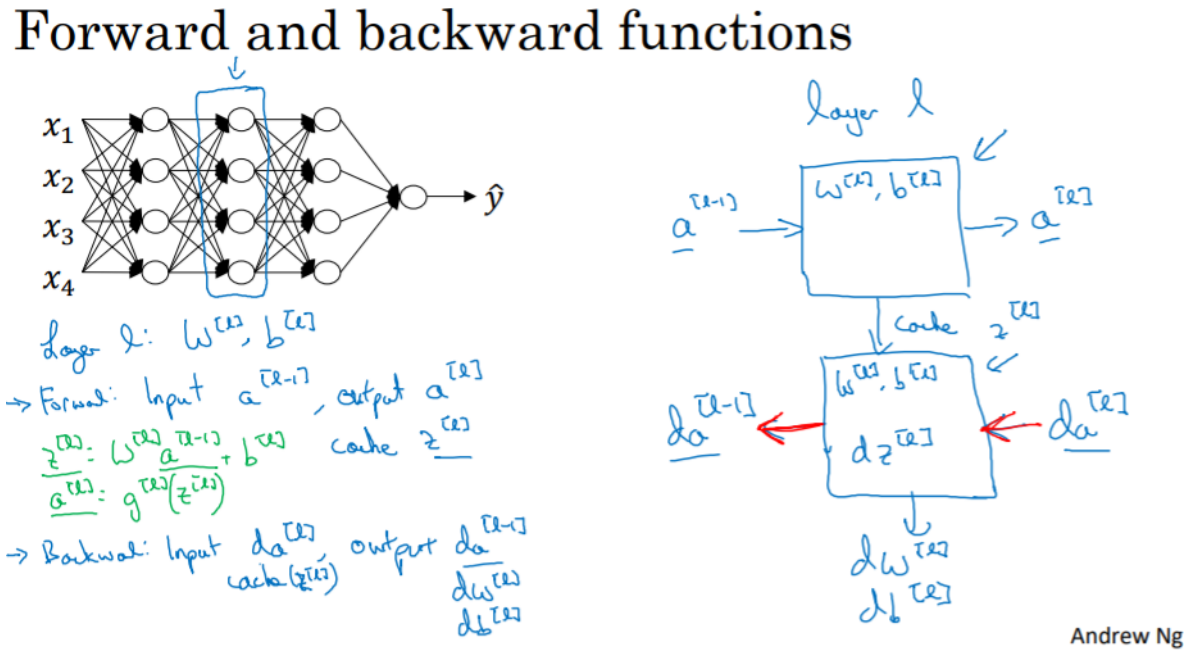
Deep Neural Network makes relations with data from simpler to complex.In each layer the NN tries to make a relation with the previous layer. To illustrate, let's see the following examples :

1. Face recognition application : image ⇒ Edges ⇒ Face part ⇒ desired face
2. Audio recognition application : Audio ⇒ Low level sound features ⇒ Phonemes ⇒ Words ⇒ Sentences

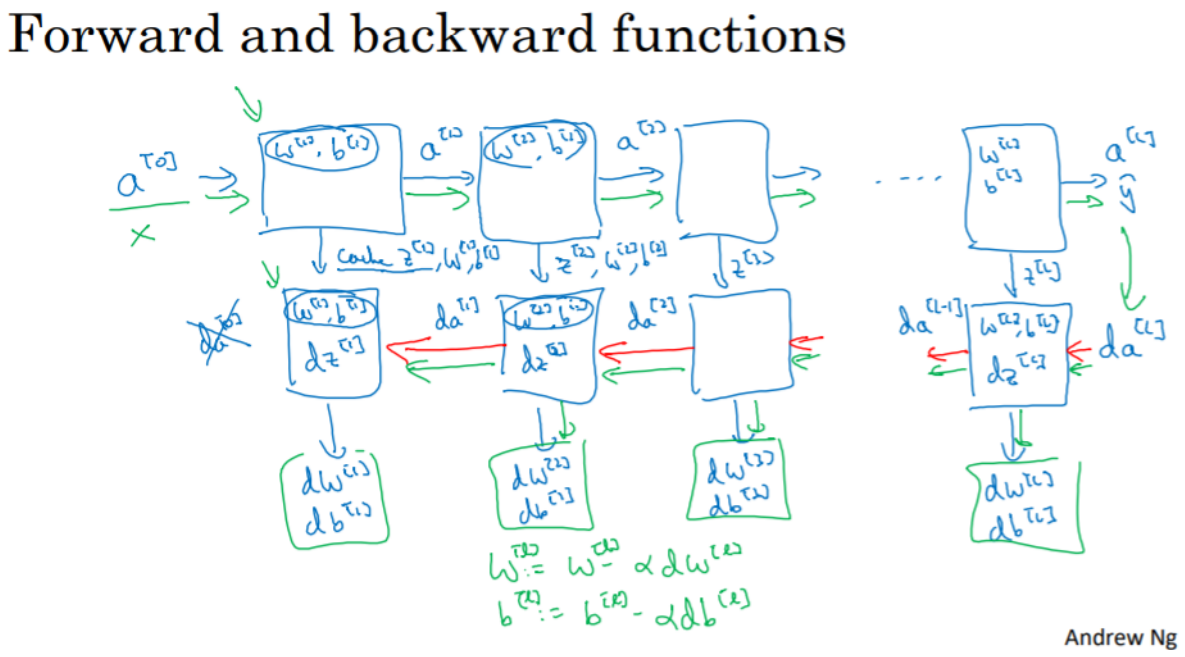
Notice : When starting an application, always start by the simplest solution : logistic regression, then Shallow NN, then Deep NN, etc.

1.4.5 Building blocks of deep neural networks

We are going to explain the forward and backward propagation using blocks. For the l^{th} layer :



For a Deep NN the blocks are :



1.4.6 Forward and Backward Propagation

To recap :

1. Forward propagation :

$$
\begin{aligned}
Z^{[1]} &= W^{[1]}X + b^{[1]} \\
A^{[1]} &= g^{[1]}(Z^{[1]}) \\
Z^{[2]} &= W^{[2]}A^{[1]} + b^{[2]} \\
A^{[2]} &= g^{[2]}(Z^{[2]}) \\
&\vdots \\
A^{[L]} &= g^{[L]}(Z^{[L]}) = \hat{Y}
\end{aligned}
$$

2. Back propagation :

$$
\begin{aligned}
dZ^{[L]} &= A^{[L]} - Y \\
dW^{[L]} &= \frac{1}{m} dZ^{[L]} A^{[L]^T} \\
db^{[L]} &= \frac{1}{m} np.\text{sum}(\text{d}Z^{[L]}, axis = 1, keepdims = True) \\
dZ^{[L-1]} &= dW^{[L]^T} dZ^{[L]} g'^{[L]}(Z^{[L-1]}) \\
&\vdots \\
dZ^{[1]} &= dW^{[L]^T} dZ^{[2]} g'^{[1]}(Z^{[1]}) \\
dW^{[1]} &= \frac{1}{m} dZ^{[1]} A^{[1]^T} \\
db^{[1]} &= \frac{1}{m} np.\text{sum}(\text{d}Z^{[1]}, axis = 1, keepdims = True)
\end{aligned}
$$

1.4.7 Parameters vs Hyperparameters

Parameters :
The main parameters for a Neural Network are : W and B.

Hyperparameters :

- Learning rate
- Number of layers L
- Number of units n
- Number of iteration
- Activation function

1.4.8 What does this have to do with the brain

- The analogy that "It is like the brain" has become really an oversimplified explanation.
- There is a very simplistic analogy between a single logistic unit and a single neuron in the brain.
- No human today understand how a human brain neuron works.
- No human today know exactly how many neurons on the brain.
- Deep learning in Andrew's opinion is very good at learning very flexible, complex functions to learn X to Y mappings, to learn input-output mappings (supervised learning).
- The field of computer vision has taken a bit more inspiration from the human brains then other disciplines that also apply deep learning.
- NN is a small representation of how brain work. The most near model of human brain is in the computer vision (CNN)

2 Course 2 Improving Deep Neural Networks Hyperparameter tuning, Regularization and Optimization

This course will teach you the "magic" of getting deep learning to work well. Rather than the deep learning process being a black box, you will understand what drives performance, and be able to more systematically get good results. You will also learn TensorFlow. After 3 weeks, you will :

- Understand industry best-practices for building deep learning applications.
- Be able to effectively use the common neural network "tricks", including initialization, L2 and dropout regularization, Batch normalization, gradient checking,
- Be able to implement and apply a variety of optimization algorithms, such as mini-batch gradient descent, Momentum, RMSprop and Adam, and check for their convergence.
- Understand new best-practices for the deep learning era of how to set up train/dev/test sets and analyze bias/variance
- Be able to implement a neural network in TensorFlow.

2.1 Week 1

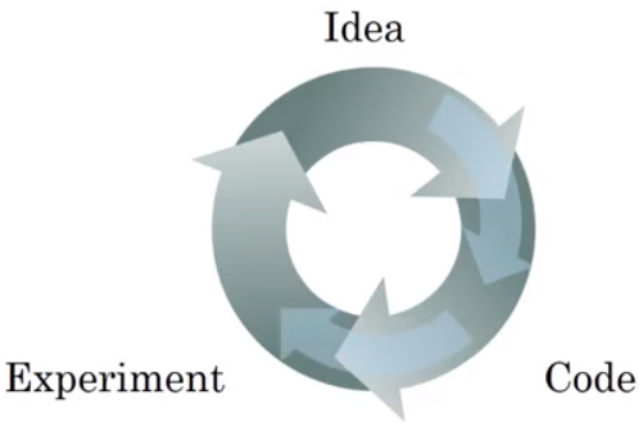
Recall that different types of initializations lead to different results
Recognize the importance of initialization in complex neural networks.
Recognize the difference between train/dev/test sets
Diagnose the bias and variance issues in your model
Learn when and how to use regularization methods such as dropout or L2 regularization.
Understand experimental issues in deep learning such as Vanishing or Exploding gradients and learn how to deal with them
Use gradient checking to verify the correctness of your backpropagation implementation

2.1.1 Setting up your Machine Learning Application

(a) Train/Dev/Test :
Deep Neural Networks need many hyperparameter to work, such as :

- number of layers
- number of hidden units
- learning rate
- activation function
- etc

Si it is difficult to get them right at the first time. To overcome this issue we use The Machine Learning process illustrated below :



We start by an idea which is a set of hyperparameters, then we code it and check the results of the experiment.
This process is repeated many times until we find the best hyperparameters ⇒ needs computation power (CPU, GPU, etc).

One of the ways to speed up this process is to setup data into train/dev/test :

The ratio of train/dev/test differ depending on the context :

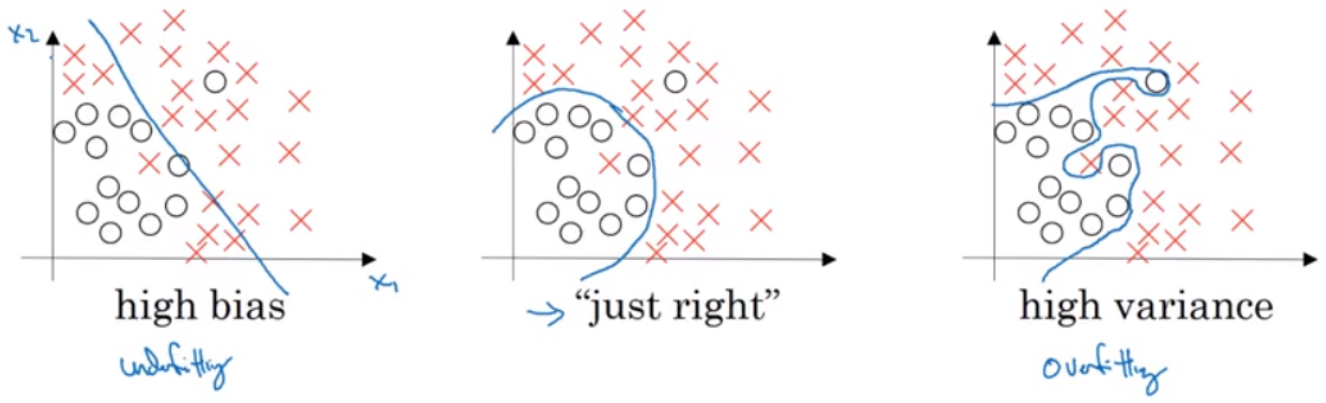
- Previous era : 70%/30%, 60%/20%/20%
- Big Data : 98%/1%/1%

Mismatched train/test Distribution :
⇒ Make sure that dev/test set come from the same distribution as the train set. Let's see the following example :

1. Training cat pictures from webpages : hight resolution, very framed pictures
2. Dev/Test cat pictures from users using the app : low resolution, etc

Notice : It is ok to not having a test set.

(b) Bias/Variance :

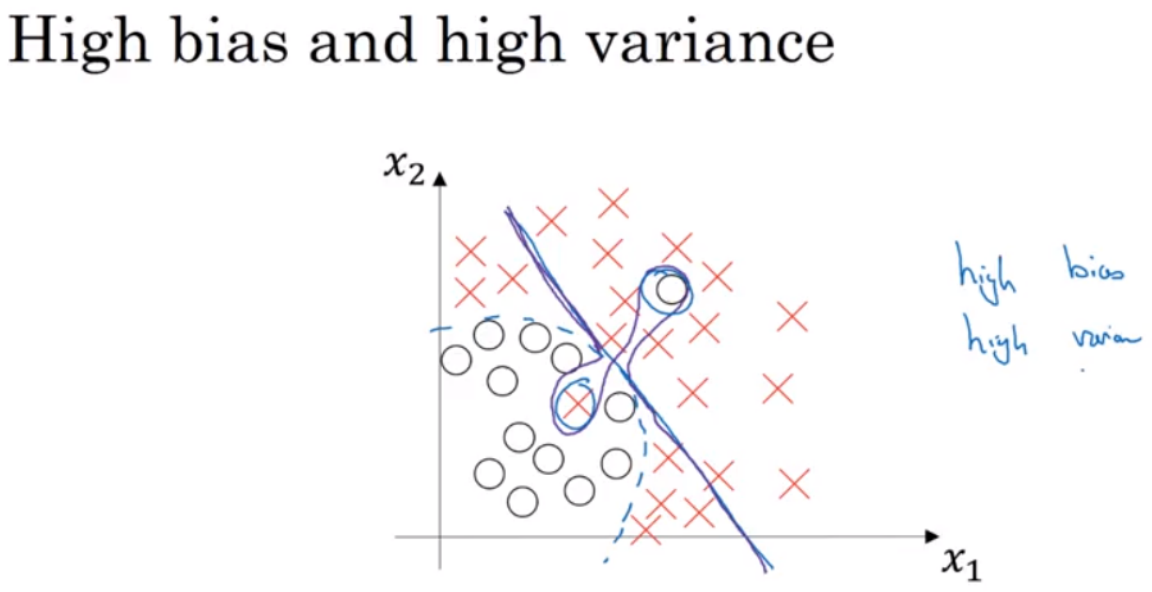


In a 2D problem we can plot the decision boundary and inspect the trade-off bias-variance. But in a multidim problem we could use some metrics to inspect that trade-off :

- Overfitting ⇒ High Variance :
$$\begin{cases} Train_{error} & = 1\% \\ Dev_{error} & = 16\% \end{cases}$$
- Underfitting ⇒ High Bias :
$$\begin{cases} Train_{error} & = 15\% \\ Dev_{error} & = 16\% \end{cases}$$
- High Bias & High Variance :
$$\begin{cases} Train_{error} & = 15\% \\ Dev_{error} & = 30\% \end{cases}$$
- Low Bias & Low Variance :
$$\begin{cases} Train_{error} & = 0.5\% \\ Dev_{error} & = 1\% \end{cases}$$

Notice : The base error used is a human error = 0%.

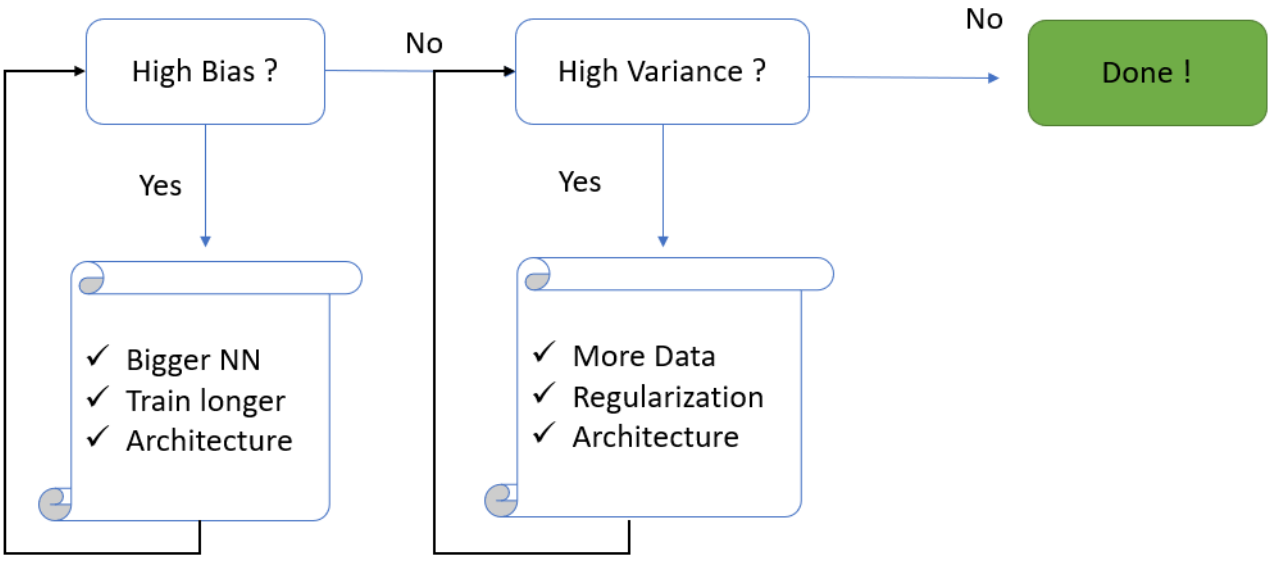
High Bias /High Variance :



- High Bias : because of the linear model that can not fit data perfectly.
- Hight Variance : because of the flexibility to fit those two mislabel.

We are going now to see how can wee deal with variance-bias.

(c) Basic "recipe for Machine Learning" :
After having trained an initial model, we ask the following questions :



Notice :

- The pre deep learning era : there has not been much tool to deal with bias-variance without hurting the other one.
- In deep learning era : by having bigger network, we can only reduce bias without hurting variance. And bu having more data, we can only reduce down variance without hurting bias.

2.1.2 Regularizing your neural Network

(a) Regularization :

Logistic Regression :
Given : $\underset{w,b}{\text{minimize}} J(w,b)$, $w \in \mathbb{R}^{n_x}$, $b \in \mathbb{R}$, λ regularization parameter.
The new cost function with the regularization term is :

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

Two types of regularization were shown :

1. $L2$ Regularization : $||w||_2^2 = \sum_{i=1}^{n_x} w_j^2 = w^T w$
2. $L1$ Regularization : $||w||_1 = \sum_{i=1}^{n_x} |w_j|$ used to compress models

Neural Network :

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

where $||W^{[l]}||_F^2 = \sum_{i=1}^{n[l-1]}\sum_{j=1}^{n[l]}(w_{ij^{[l]}})^2$: Frobenius norm
So how can we implemente Gradient Descent with this definition :

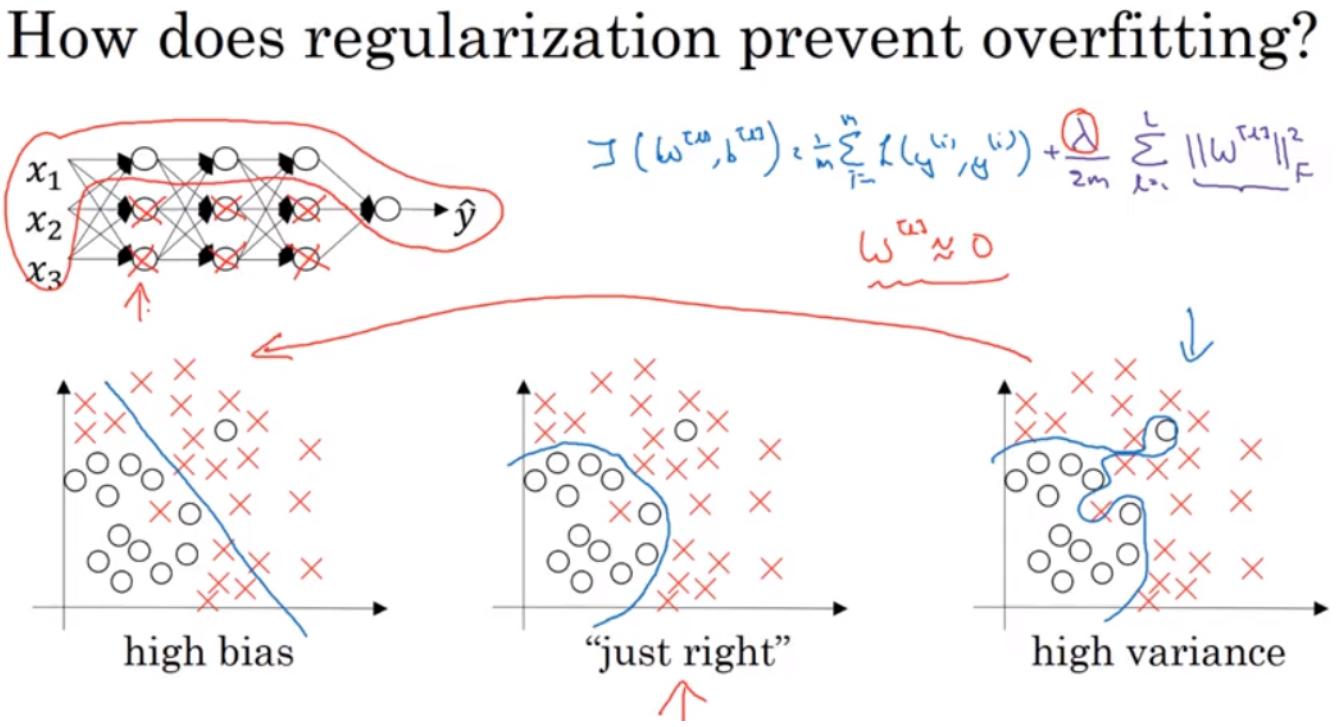
$$\begin{cases} dW^{[L]} &= (frombackprop) + \frac{\lambda}{m}W^{[L]} \\ W^{[L]} &= W^{[L]} - \alpha dW^{[L]} \end{cases}$$

$\Rightarrow$ Consequently :

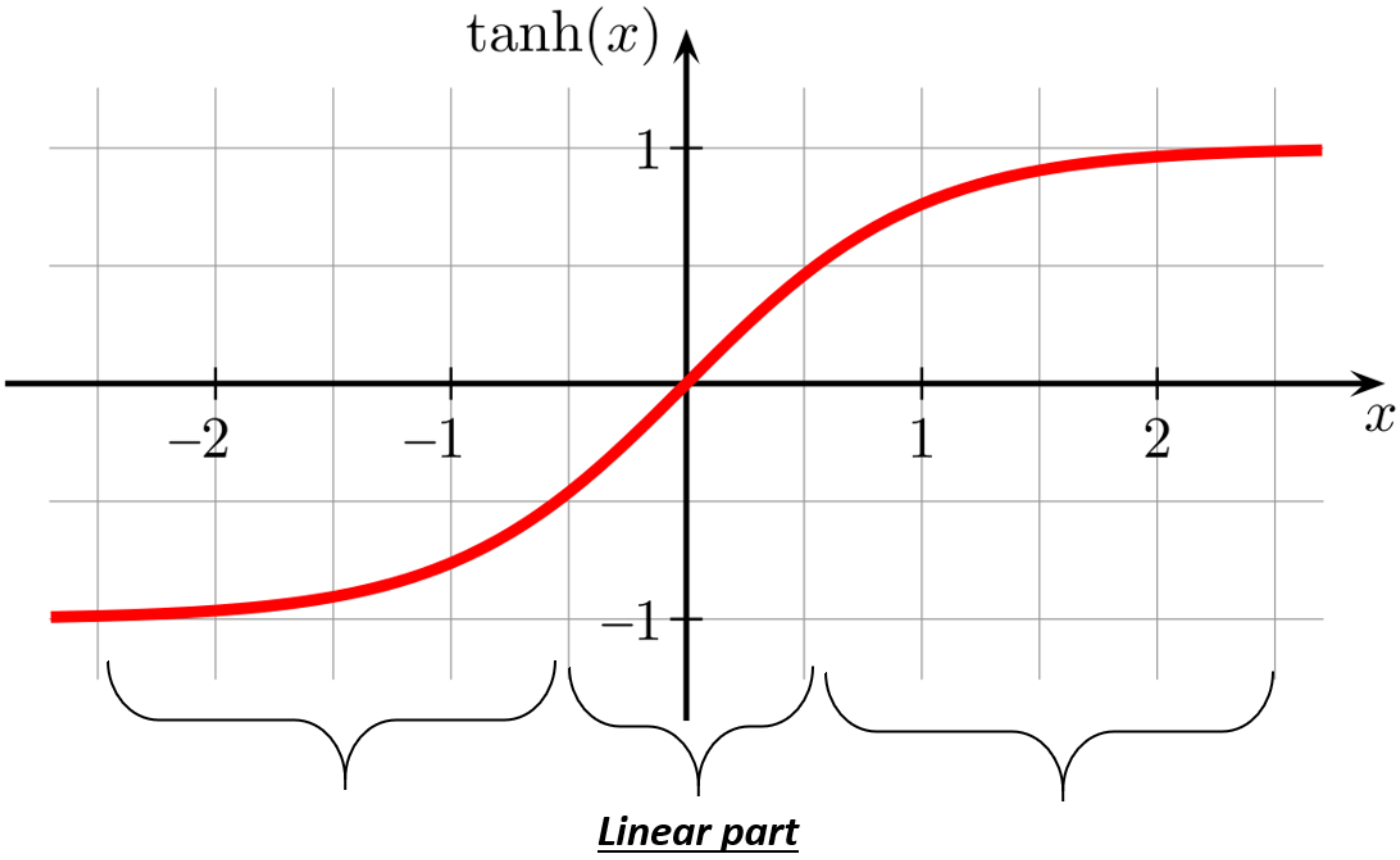
$$\begin{cases} dW^{[L]} &= (frombackprop) + \frac{\lambda}{m}W^{[L]} \\ W^{[L]} &= (1 - \frac{\alpha\lambda}{m})W^{[L]} - \alpha(frombackprop) \end{cases}$$

(b) Why Regularization reduces overfitting :
We have : $J(W^{[L]}, b^{[L]}) = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\sum_{l=1}^{L} ||W^{[l]}||_F^2$

If λ is large $\Rightarrow W^{[L]} \simeq 0$ (because we minimize J)$\Rightarrow$ Neural Network would be simpler $\Rightarrow$ [High Variance $\rightarrow$ High Bias]

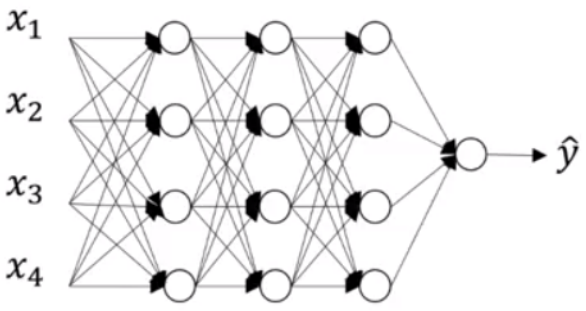


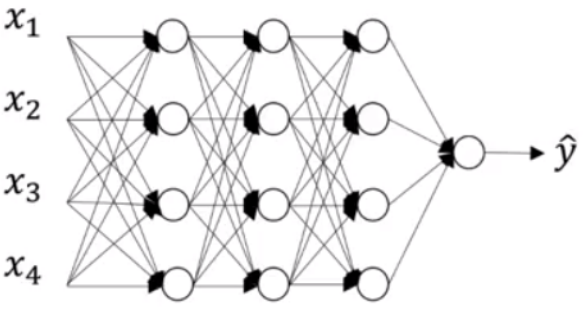
Another intuition using tanh :

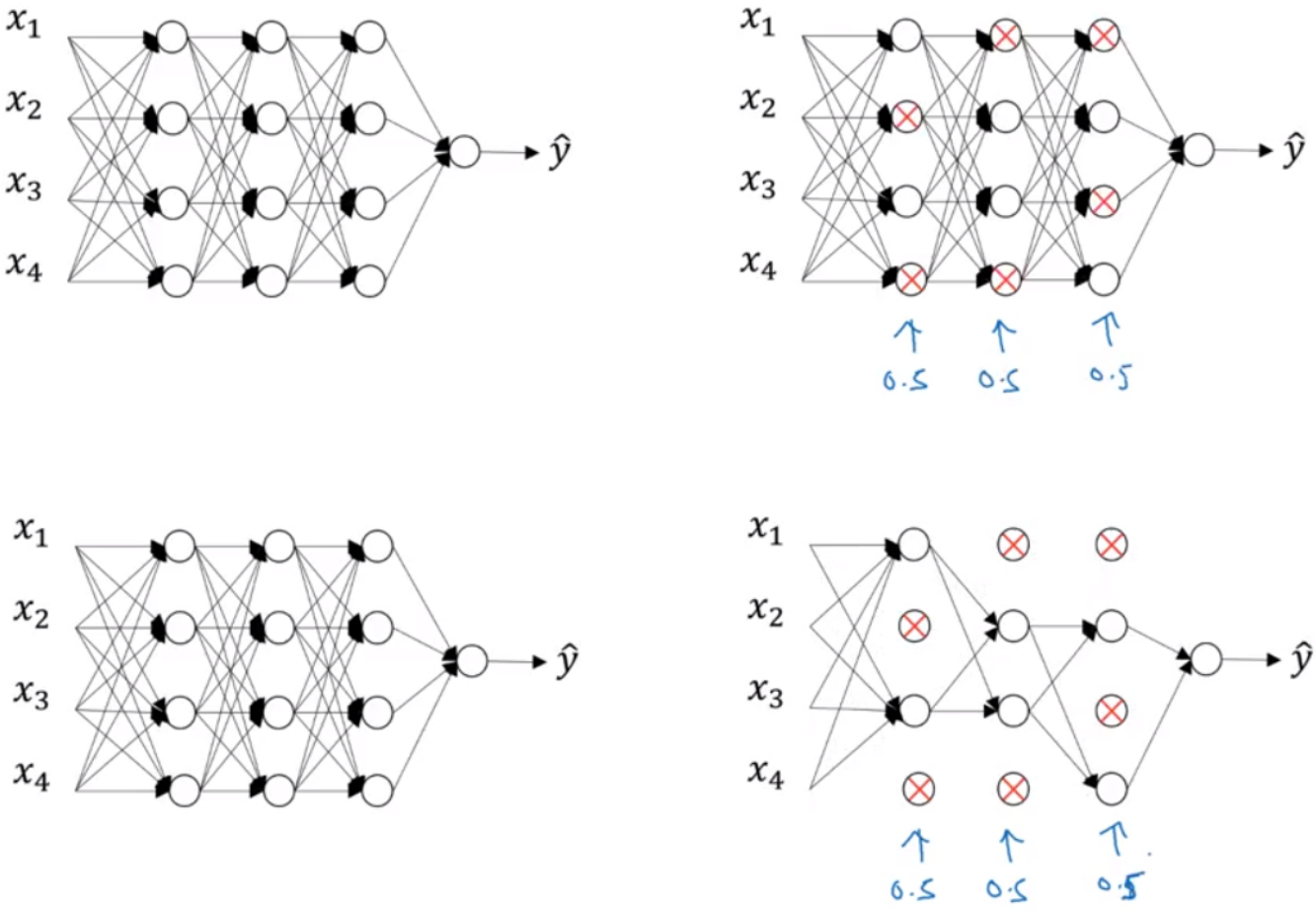


If λ is large $\Rightarrow W^{[L]} \simeq 0$ (because we minimize J)$\Rightarrow Z^{[L]} \simeq 0 \Rightarrow$ Neural Network would be simpler $\Rightarrow$ High Bias

(c) Drop out Regularization :
In addition to $L2$ Regularization, we have another type of Regularization called : Drop-out Regularization. For this new regularization we set for each layer a probability of keeping/removing a node. The following figures illustrates our saying :







Implementing dropout ("Inverted dropout") :
Let's illustrate with layer l= 3.

```
d3 = np.random.rand(a3.shape[0],a3.shape[1])<keep.prob
#keep.prob:  is the probability of keeping a node
a3 = np.multiply(a3,d3)  # a3*=d3
a3 = a3/keep.prob    # Inverted drop out technique
# get back the expected  value of a3.
```

(d) Understanding dropout :

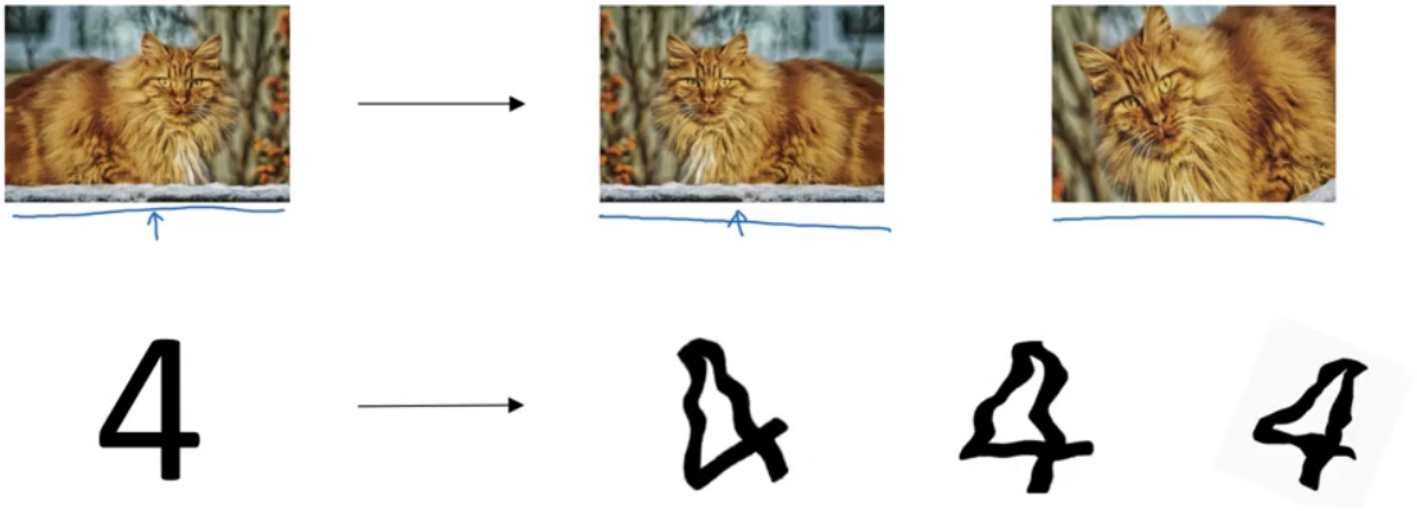
- The intuition was that dropout randomly knocks out units in the network network. So it's as if on every iteration you're working with a smaller NN, and so using a smaller NN seems like it should have a regularizing effect.
- Another intuition: can't rely on any one feature, so have to spread out weights.

- It's possible to show that dropout has a similar effect to L2 regularization.
- Dropout can have different *keepprob* per layer.
- The input layer dropout has to be near 1 (or 1 - no dropout) because you don't want to eliminate a lot of features.
- If you're more worried about some layers overfitting than others, you can set a lower *keepprob* for some layers than others. The downside is, this gives you even more hyperparameters to search for using cross-validation. One other alternative might be to have some layers where you apply dropout and some layers where you don't apply dropout and then just have one hyperparameter, which is a *keepprob* for the layers for which you do apply dropouts.
- A lot of researchers are using dropout with Computer Vision (CV) because they have a very big input size and almost never have enough data, so overfitting is the usual problem. And dropout is a regularization technique to prevent overfitting.
- A downside of dropout is that the cost function J is not well defined and it will be hard to debug (plot J by iteration) :To solve that you'll need to turn off dropout, set all the *keepprobs* to 1, and then run the code and check that it monotonically decreases J and then turn on the dropouts again.

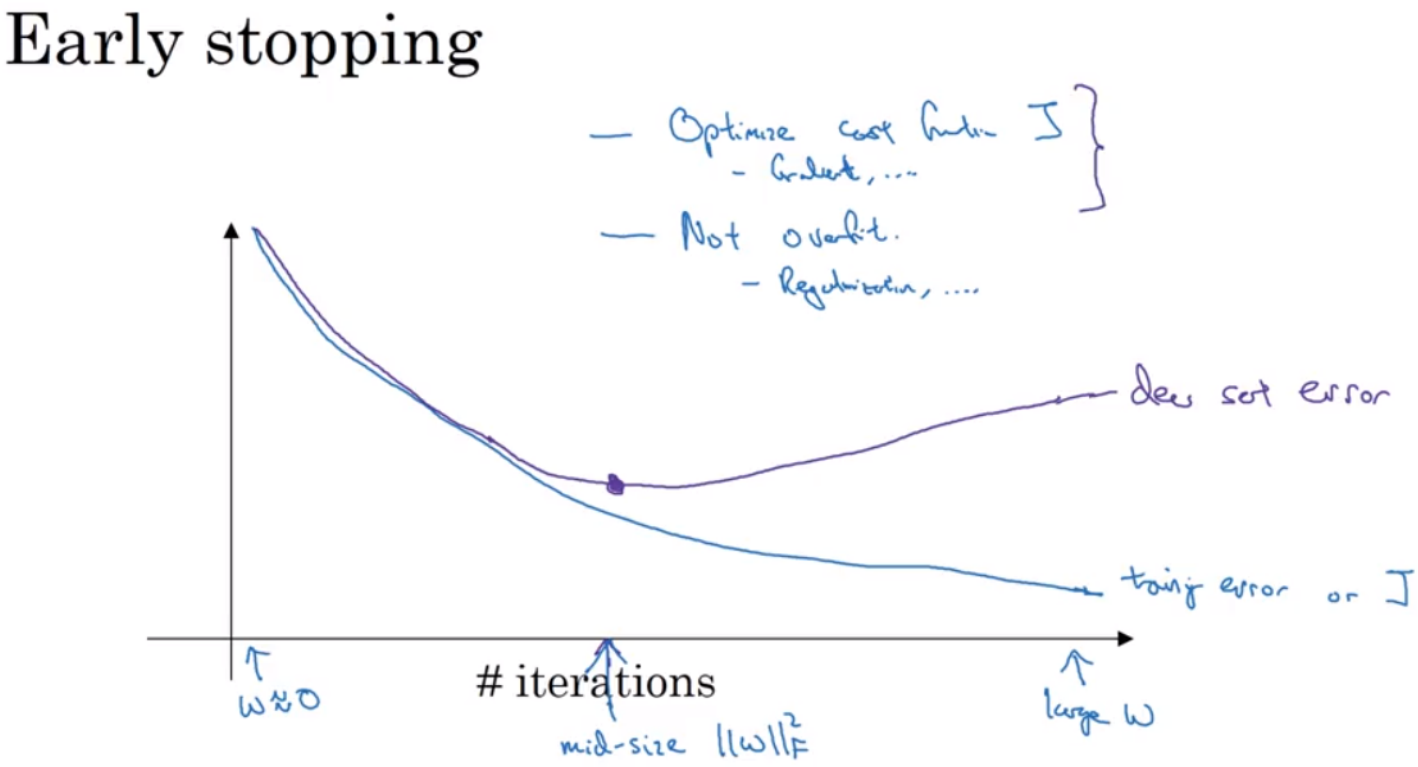
(e) Other regularization methods :
In addition to $L2$and dropout regularization

- Data augmentation : create extra fake examples by :
  1. flipping horizontally pictures
  2. randomly zoom into the pictures
  3. randomly distortion into the pictures.
  4. for OCR, you can create random rotations
  5. New data obtained using those techniques is not good as the real independent data, but still can be used as a regularization technique.

Data augmentation

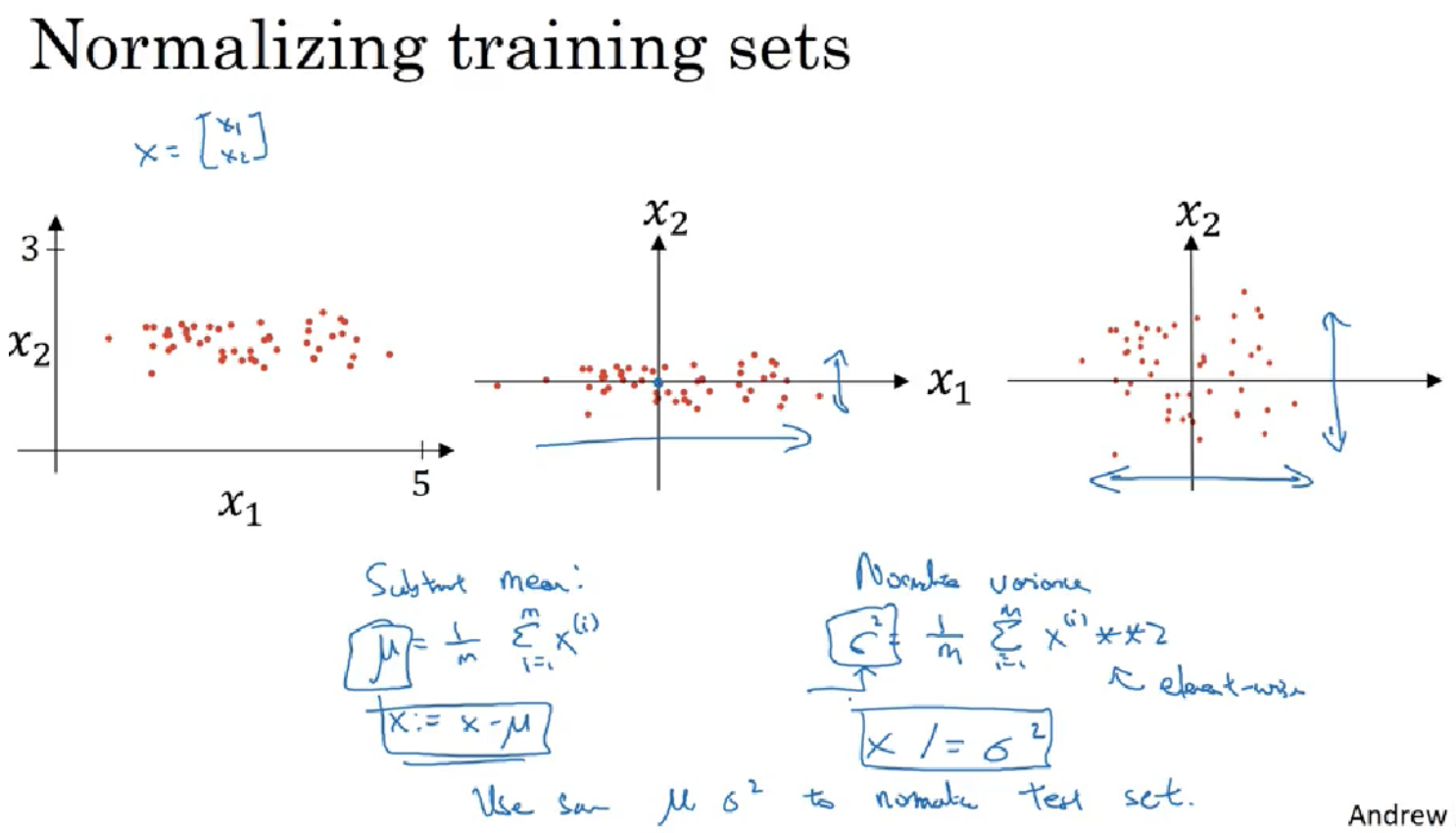


- Early stopping :
  1. In this technique we plot the training set and the dev set cost together for each iteration. At some iteration the dev set cost will stop decreasing and will start increasing.
  2. We will pick the point at which the training set error and dev set error are best (lowest training cost with lowest dev cost).
  3. We will take these parameters as the best parameters.
  4. Andrew prefers to use L2 regularization instead of early stopping because this technique simultaneously tries to minimize the cost function and not to overfit which contradicts the orthogonalization approach (will be discussed further).
  5. But its advantage is that you don't need to search a hyperparameter like in other regularization approaches (like lambda in L2 regularization).



2.1.3 Setting up your optimization problem

(a) Normalizing inputs :

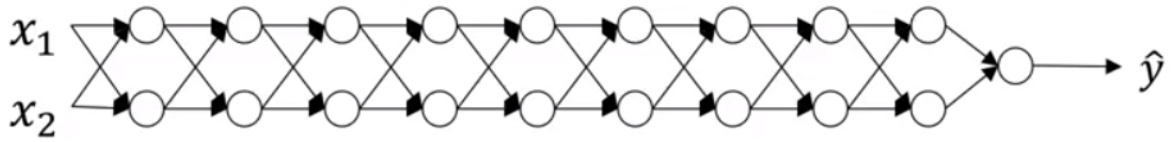


- Normalizing inputs speeds learning of the NN
- Use the same μ and σ^2 to normalize test set
- Why normalizing inputs : if the scale of features is not the same (saying $x1$ ranges from 1 to 10000 and $x2$ ranges from 0 to 1, etc) $\Rightarrow$ w_i will be updates differently $\Rightarrow$ The cost function will not converge fastly. In contrary, if all features have the same scale $\Rightarrow$ The cost function will converge fastly.

(b) Vanishing/exploding gradients :

- The problem of vanishing/exploding occurs when we are learning a very big Deep Neural Network. It is about when the derivatives (slopes) are either very small or very big.

Let's consider the following Deep Neural Network :



For simplicity, let's consider :

$$g(Z) = Z \text{ and } B^{[l]} = 0 \text{ for all layers.}$$

So the Deep Neural Network output is :

$$\hat{y} = W^{[L]}W^{[L-1]}...W^{[2]}W^{[1]}X$$

- if $\forall l \in [1, L-1]$

  $$W^{[l]} = \begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix}$$

  Then :

  $$\hat{y} = W^{[L]} \begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix}^{L-1} X$$

  $\Rightarrow$ Gradients will explode because $\hat{y}$ will have larger values
- if $\forall l \in [1, L-1]$

  $$W^{[l]} = \begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix}$$

  Then :

  $$\hat{y} = W^{[L]} \begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix}^{L-1} X$$

  $\Rightarrow$ Gradients will vanish because $\hat{y}$ will have very small values; the weights will decrease exponentially as a function of L.

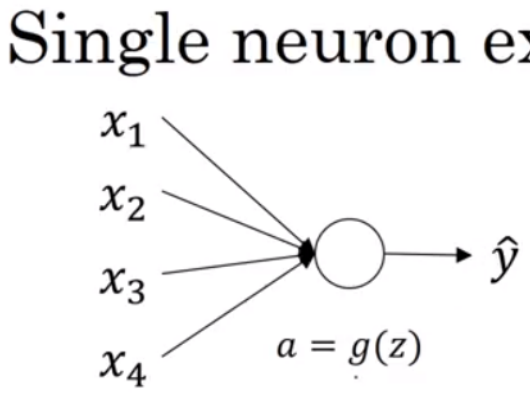
To recap :

- If $W^{[l]} > I \Rightarrow$ NN will explode
- If $W^{[l]} < I \Rightarrow$ NN will vanish.

To avoid the problem of exploding/vanishing gradients, we will use some techniques for initialization.

(c) Weight Initialization for Deep Networks :

Single neuron example :
Let's start by a single neuron example :

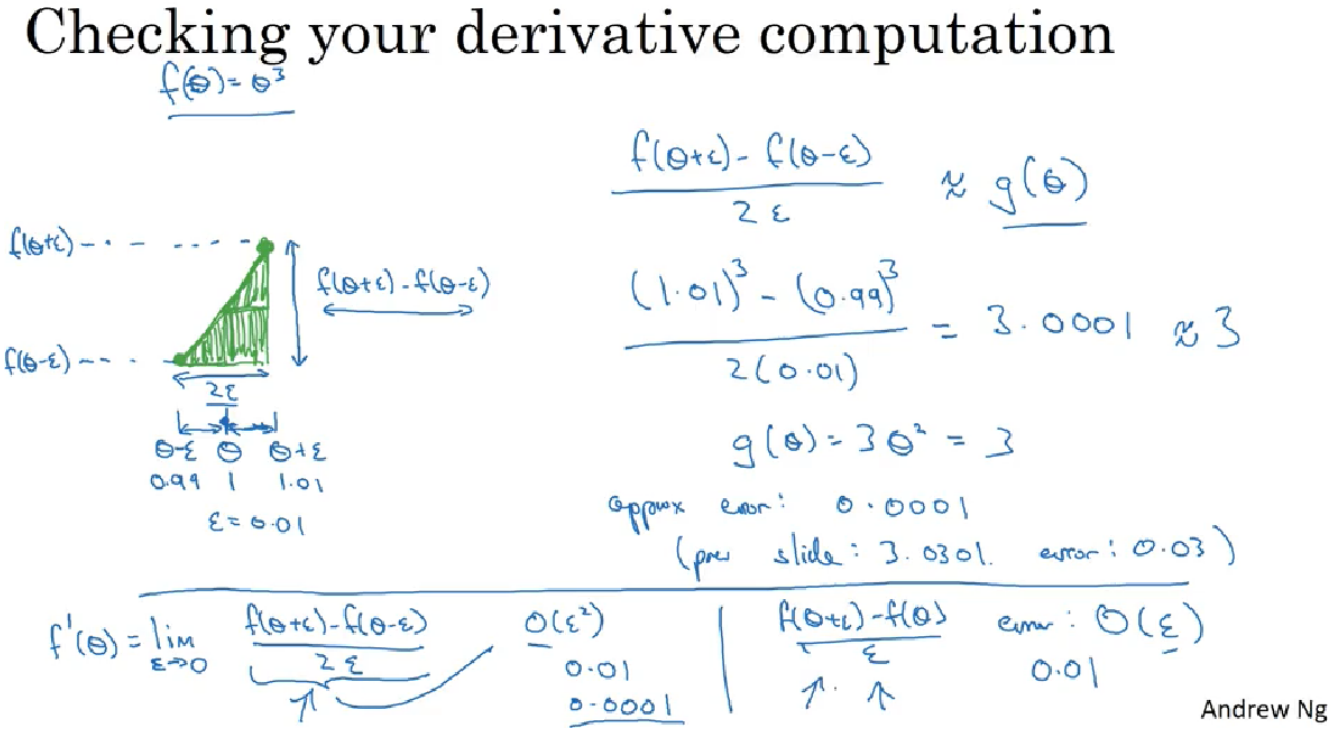


We know that : $z = \sum_{i=1}^{n} w_i x_i$ where n is the number of input features.
In order to make z not grow up/very small, notice that for large value of $n \Rightarrow$ smaller w_i should be.
One reasonable thing to do is : $Variance(w_i) = \frac{1}{n}$

Generalization for Deep Neural Network :
There many types of initialization weight, here some them that were explained by Andrew Ng :

- For tanh activation : $W^{[l]} = np.random.randn(n[l], n[l-1]) * np.sqrt(\frac{1}{n[l-1]})$
- For ReLU activation : $W^{[l]} = np.random.randn(n[l], n[l-1]) * np.sqrt(\frac{2}{n[l-1]})$
- Xavier initialization : $W^{[l]} = np.random.randn(n[l], n[l-1]) * np.sqrt(\frac{2}{n[l-1]})$
- Other one : $W^{[l]} = np.random.randn(n[l], n[l-1]) * np.sqrt(\frac{2}{n[l-1]+n[l]})$

(d) Numerical approximation for gradient :
- There is an technique called gradient checking which tells you if your implementation of backpropagation is correct.



- Gradient checking approximates the gradients and is very helpful for finding the errors in your backpropagation implementation but it's slower than gradient descent (so use only for debugging).

Implementation :

- First take $W^{[1]}$, $W^{[2]}$, ..., $W^{[L]}$ and $B^{[L]}$ and reshape them into one huge vector Θ.
- Then take $dW^{[1]}$, $dW^{[2]}$, ..., $dW^{[L]}$ and $dB^{[L]}$ and reshape them into one huge vector $d\Theta$.
- Algorithm :

```
eps = 10^-7    # small value for eps (eps-->0)

for i in range(len(theta)) :
        d_theta_approx[i] = (J(theta1,...,theta[i] + eps) -
              J(theta1,...,theta[i] - eps)) / 2*eps
```

- Finally we evaluate this formula :

$$difference = \frac{||d\Theta - d\Theta_{approx}||^2}{||d\Theta||^2 + ||d\Theta_{approx}||^2}$$

- If $difference < 10^{-7} \Rightarrow$ he backpropagation implementation is correct
- If $difference$ is around 10^{-5} : can be OK but need to inspect $d\Theta_{approx}$ values to see if there is any particularly value.
- If $difference$ >10^{-3} : must worry about the backpropagation implementation.

(e) Gradient Checking implementation notes :

- Don't use in training - only in debug
- If Algorithm fails grad check, look at components to try identify bug
- If you use regularization, do not forget to take into consideration the regularization term in cost derivative
- Does not work with drop out. You can turn it off by setting keepprob to 1, then after turn it on.
- Run gradient checking at random initialization and train the network for a while maybe there's a bug which can be seen when w's and b's become larger (further from 0) and can't be seen on the first iteration (when w's and b's are very small).

2.1.4 Notes from assignments

Initialization

- The weights $W^{[l]}$ should be initialized randomly to break symmetry.
- It is however okay to initialize the biases $b^{[l]}$ to zeros. Symmetry is still broken so long as $W^{[l]}$ is initialized randomly.
- Initializing weights to very large random values does not work well.
- Hopefully intializing with small random values does better. The important question is: how small should be these random values be? Lets find out in the next part!
- Different initializations lead to different results
- Random initialization is used to break symmetry and make sure different hidden units can learn different things
- Don't intialize to values that are too large
- He initialization works well for networks with ReLU activations.

Regularization :
Observations :
- The value of λ is a hyperparameter that you can tune using a dev set. - L2 regularization makes your decision boundary smoother. If λ is too large, it is also possible to "oversmooth", resulting in a model with high bias.

What is L2-regularization actually doing?

L2-regularization relies on the assumption that a model with small weights is simpler than a model with large weights. Thus, by penalizing the square values of the weights in the cost function you drive all the weights to smaller values. It becomes too costly for the cost to have large weights! This leads to a smoother model in which the output changes more slowly as the input changes.

What you should remember : The implications of L2-regularization on
- The cost computation: A regularization term is added to the cost

- The backpropagation function: There are extra terms in the gradients with respect to weight matrices
- Weights end up smaller ("weight decay"): Weights are pushed to smaller values.

Drop out :

- Dropout is a regularization technique.
- You only use dropout during training. Don't use dropout (randomly eliminate nodes) during test time.
- Apply dropout both during forward and backward propagation.
- During training time, divide each dropout layer by *keepprob* to keep the same expected value for the activations. For example, if *keepprob* is 0.5, then we will on average shut down half the nodes, so the output will be scaled by 0.5 since only the remaining half are contributing to the solution. Dividing by 0.5 is equivalent to multiplying by 2. Hence, the output now has the same expected value. You can check that this works even when *keepprob* is other values than 0.5.

Gradient Check

- Gradient checking verifies closeness between the gradients from backpropagation and the numerical approximation of the gradient (computed using forward propagation).
- Gradient checking is slow, so we don't run it in every iteration of training. You would usually run it only to make sure your code is correct, then turn it off and use backprop for the actual learning process.

2.2 Week 2

Deep learning works well in a huge dataset, But this makes learning slow. In case of having optimization algorithms can speed up efficiently learning. The objectif of the next sections is to study a variety of algorithms used for optimization.

2.2.1 Mini-batch gradient descent

As discussed before, we know that vectorization allows us to efficiently compute on m examples :

$$\begin{cases} X &= [X^{(1)}, X^{(2)}, ..., X^{(m)}] \\ Y &= [Y^{(1)}, Y^{(2)}, ..., Y^{(m)}] \end{cases}$$

But when m is very large, learning can be slow.
To overcome this issue, we will use a technique called mini-batch gradient descent.

Mini-batch gradient descent VS Batch gradient descent

Batch gradient descent : is the same as the gradient descent. It is about processing the whole dataset.

Mini-batch gradient descent : in this technique we split our training dataset in mini-batches. Let's take the example where the number of minibatches is 5000 of 1000 examples each.

$$\begin{cases} X &= [X^{(1)}, X^{(2)}, ..., X^{(m)}] \\ Y &= [Y^{(1)}, Y^{(2)}, ..., Y^{(m)}] \end{cases}$$

$\Rightarrow$

$$\begin{cases} X &\to X^{\{1\}} = [X^{(1)}...X^{(1000)}], X^{\{2\}} = [X^{(1001)}...X^{(2000)}], ..., X^{\{5000\}} = [X^{(m-1000+1)}...X^{(m)}] \\ Y &\to Y^{\{1\}} = [Y^{(1)}...Y^{(1000)}], Y^{\{2\}} = [Y^{(1001)}...Y^{(2000)}], ..., Y^{\{5000\}} = [Y^{(m-1000+1)}...Y^{(m)}] \end{cases}$$

So a mini-batch t is defined as : $(X^{\{t\}}, Y^{\{t\}})$.
Having this definition of minibatches, mini gradient would be the following :

for t=1 .... 5000 :
$\to\to$ Forward prop on $X^{\{t\}}$: one step of gradient descent using $(X^{\{t\}}, Y^{\{t\}})$:

$$\begin{cases} Z^{[1]} &= W^{[1]}X^{\{t\}} + B^{[1]} \\ A^{[1]} &= g^{[1]}(Z^{[1]}) \\ ... &= ... \\ A^{[L]} &= g^{[L]}(Z^{[L]}) \end{cases}$$

$\to \to$ Compute cost :

$$J^{\{t\}} = \frac{1}{1000}\sum_{i=1}^{l} L(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2*1000}\sum_{l} ||W^{l}||_F^2$$

$\to \to$ Backprop to compute gradients with respect $J^{\{t\}}$ using $(X^{\{t\}}, Y^{\{t\}})$:

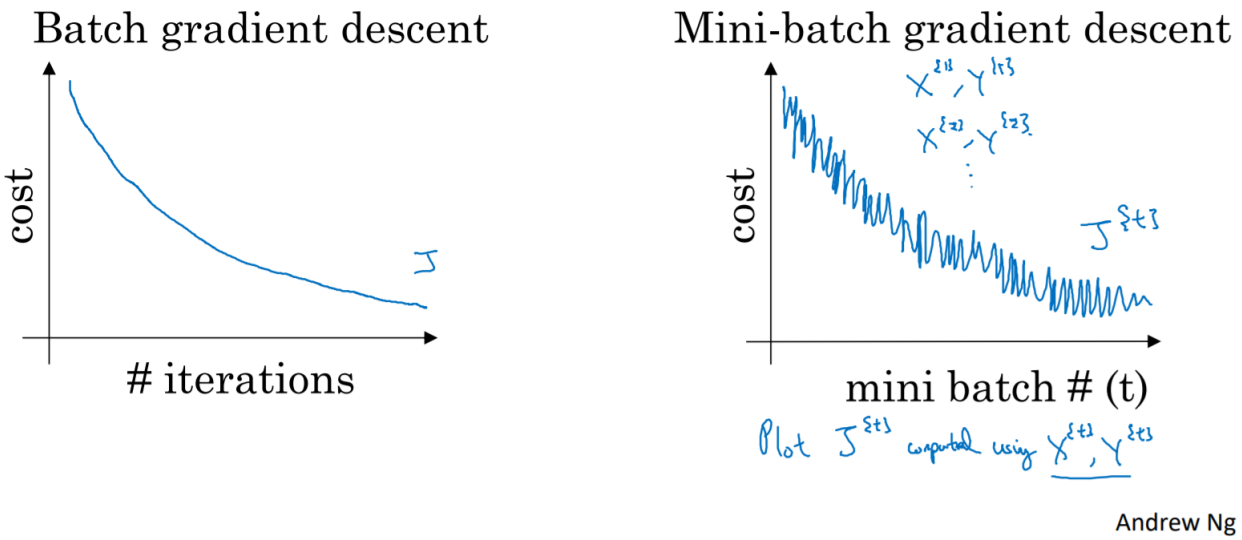
$$\begin{cases} W^{[l]} &= W^{[l]} - \alpha * dW^{[l]} \\ B^{[l]} &= B^{[l]} - \alpha * dB^{[l]} \end{cases}$$

To conclude :

- Batch gradient descent : one epoch[3] allows you to have only one gradient descent step.
- Minibatch gradient descent : one epoch allows you to have 5000 (number of minibatches) step in gradient descent.

[3]Epoch : a single pass through the training dataset

2.2.2 Understanding mini-batch gradient descent



Cost function

- In Batch gradient descent : the cost function decrease with each iteration
- In Minibatch gradient descent, the cost function is noisy (some ups/down). This is due to the fact that there harder/easy minibatches.

Choosing your minibatch size

- if minibatch size == m ⇒ Batch gradient descent
- if minibatch size == 1 ⇒ Stochastic gradient descent (every example is its own minibatch)
- if 1 < minibatch size < m ⇒ Minibatch gradient descent

1. Batch gradient descent :
   - converge directly
   - too long per iteration
2. Stochastic gradient descent :
   - a noisy trajectory will oscillate around the region of minimum
   - lose speed up of vectorization
3. Minibatch gradient descent :
   - fastest learning
   - vectorization possible
   - make progress without waiting to progress the entire dataset
   - it does bot always converge to the right minumum (oscillate around a very small region, but you can reduce learning rate to fix that)

Some guidelines to choose minibatch size (new hyperparameter)

1. If small training set (<=2000) : use batch gradient descent
2. otherwise : minibatch size has to be a power of 2 (because of the way computer memory is layed out and accessed) : 64, 128, 512, 1024, ...
3. make sure that minibatch fits in GPU/CPU memory.

2.2.3 Exponentially weighted average

Let's explain exponentially weighted average that will be used for other faster optimization algorithms.
Let's take an example by working on the following data that could represent the temperature of day through one year :

$$\begin{cases} \theta_1 &= 40 \\ \theta_2 &= 49 \\ \theta_3 &= 45 \\ ... &= ... \\ \theta_{180} &= 60 \\ ... &= .... \end{cases}$$

If we plot this data, it will be noisy. That's why we are going to compute exponentially weighted average. Exponentially weighted average works as follow :

$$\begin{cases} V_0 &= 0 \\ V_1 &= 0.9 * V_0 + 0.1 * \theta_1 \\ V_2 &= 0.9 * V_1 + 0.1 * \theta_2 \\ V_3 &= 0.9 * V_2 + 0.1 * \theta_3 \\ ... &= .... \end{cases}$$

The general equation is :

$$V_t = \beta * V_{t-1} + (1 - \beta) * \theta_t$$

$\Rightarrow$If we plot this it will represent averages over $\approx \frac{1}{1-\beta}$ entries :

1. $\beta = 0.9$ will average last 10 entries (red line in figure 1)
2. $\beta = 0.98$ will average last 50 entries (green line in figure 1)
3. $\beta = 0.5$ will average last 2 entries (yellow line in figure 2)

if $\beta \nearrow$ the plot become more smoothy.

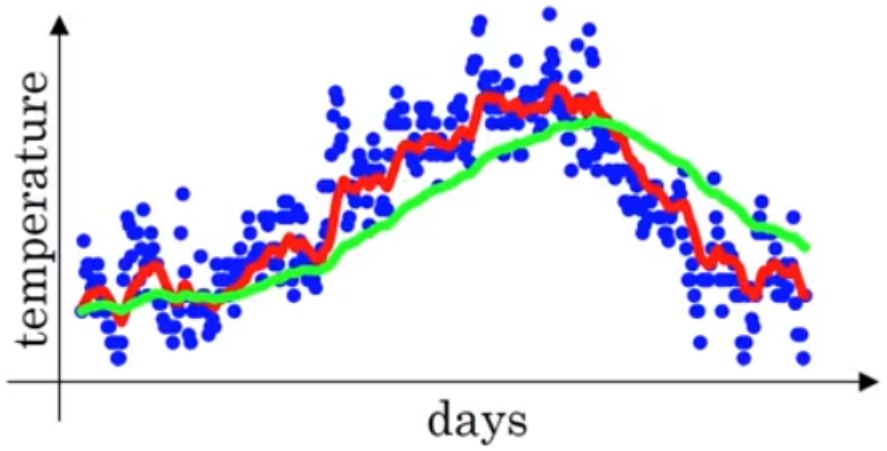


Figure 1: Exponentially averages

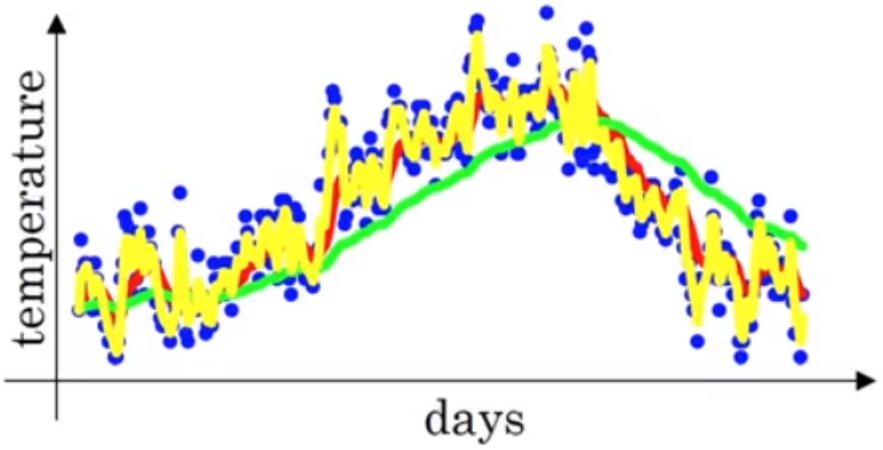


Figure 2: Exponentially averages

Understanding exponentially weighted averages :
Recall that the general equation of exponentially weighted averages is :

$$V_t = \beta * V_{t-1} + (1 - \beta) * \theta_t$$

Which gives for example (β =0.9) :

$$\begin{cases} V_{100} &= 0.9 * V_{99} + 0.1 * \theta_{100} \\ V_{99} &= 0.9 * V_{98} + 0.1 * \theta_{99} \\ ... &= .... \end{cases}$$

Let's see what is V_{100} :

$$\begin{cases} V_{100} &= 0.1 * \theta_{100} + 0.9 * V_{99} \\ &= 0.1 * \theta_{100} + 0.9 * (0.9 * V_{98} + 0.1 * \theta_{99}) \\ &= 0.1 * \theta_{100} + 0.9 * 0.1 * \theta_{99} + 0.9^2 * V_{98} \\ &= 0.1 * \theta_{100} + 0.9 * 0.1 * \theta_{99} + 0.1 * 0.9^2 * \theta_{98} + 0.1 * 0.9^3 * \theta_{97} + ... \end{cases}$$

$\Rightarrow$ This is really a weighted average.

Implementation :

```
v = 0
Repeat
{
    Get theta(t)
    v = beta*v+ (1-beta)*theta(t)
}
```

Bias correction in expenentially weighted averages

- Bias correction makes the computation of exponentially average more accurate.
- Because v(0) = 0, the bias of the weighted averages is shifted and the accuracy suffers at the start
- To solve the bias issue we have to use the equation below
$$V_t = \frac{\beta * V_{t-1} + (1-\beta) * \theta_t}{1-\beta^t}$$
- As t becomes larger the $(1 - \beta^t)$ becomes close to 1; for larger values of t we do not need bias correction. it is only needed for the start of the algorithm to estimate correctly the first values.

2.2.4 Gradient descent with momentum

The basic idea is to compute an exponentially weighted average of gradients, and then use that gradient to update weights instead.

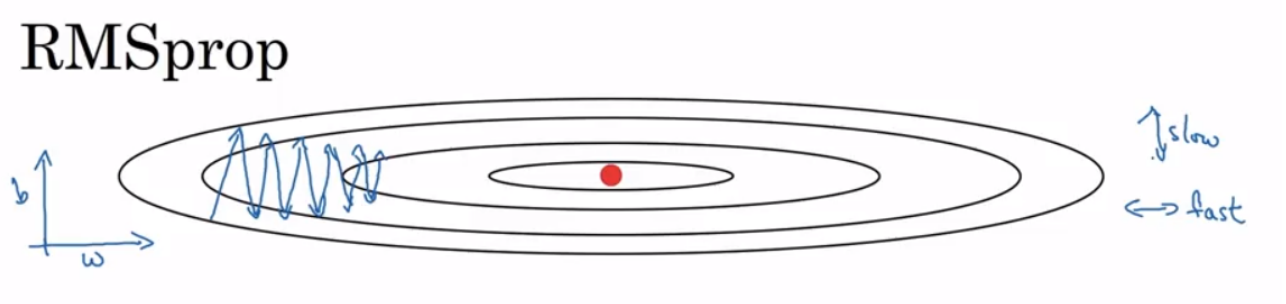
How it works ? :
On iteration t :

- Compute dw, db on current minibatch
$$\begin{cases} V_{dw} &= \beta * V_{dw} + (1 - \beta)dw \\ V_{db} &= \beta * V_{db} + (1 - \beta)db \end{cases}$$
- Update
$$\begin{cases} w &= w - \alpha * V_{dw} \\ b &= b - \alpha * V_{db} \end{cases}$$

- Those last two steps smooth out the steps of gradient descent. Allowing our learning to move faslty toward the optimum.
- Hyperparameters : α, β ($\beta = 0.9$ is the most common value. It means that we average the last ten iteration's gradients.)

2.2.5 RMS prop : Root Mean Square prop

Recall that when implementing gradient descent, we would have a gradient descent that can be illustrated by the following figure :



Two objectives are to consider :

1. Want to slow down learning in the b direction
2. speed up learning in the w direction

To accomplish that, RMS prop does :
On iteration t :

- Compute dw, db on current minibatch

$$\begin{cases} S_{dw} &= \beta_2 * S_{dw} + (1-\beta_2)dw^2 \\ S_{db} &= \beta_2 * S_{db} + (1-\beta_2)db^2 \end{cases}$$

- Update

$$\begin{cases} w &= w - \alpha * \frac{dw}{\sqrt{S_{dw}+\epsilon}} \\ b &= b - \alpha * \frac{db}{\sqrt{S_{db}+\epsilon}} \end{cases}$$

- ϵ : is used to ensure numerical stability in the case the denominator is zero (typical value would be $\epsilon = 10^{-8}$)
- To achieve our goals (slow down in b direction, and faster in w direction) we hope by applying RMS prop that :
  - $S_{dw} \searrow \searrow \Rightarrow \frac{dw}{\sqrt{S_{dw}+\epsilon}} \nearrow \nearrow$
  - $S_{dw} \nearrow \nearrow \Rightarrow \frac{dw}{\sqrt{S_{dw}+\epsilon}} \searrow \searrow$

Conclusion :
RMSprop has the same effect as the momentum because both of them damping out the oscillations in gradient descent, and in minibatch gradient descent.
Now we will put momentum & RMS prop together to get a much better optimization algorithm.

2.2.6 Adam (Adaptive moment estimation) optimization algorithm

The algorithm is the following :

1. $V_{dw} = 0$, $S_{dw} = 0$, $V_{db} = 0$, $S_{db} = 0$

2. on iteration t : compute dw, db using minibatch gradient descent

$$\text{(1) Momentum } \beta_1 : \begin{cases} V_{dw} &= \beta * V_{dw} + (1-\beta)dw \\ V_{db} &= \beta * V_{db} + (1-\beta)db \end{cases}$$

$$\text{(2) RMS prop } \beta_2 : \begin{cases} S_{dw} &= \beta_2 * S_{dw} + (1-\beta_2)dw^2 \\ S_{db} &= \beta_2 * S_{db} + (1-\beta_2)db^2 \end{cases}$$

$$\text{(3) Momentum corrected : } \begin{cases} V_{dw}^{corrected} &= \frac{V_{dw}}{1-\beta_1^t} \\ V_{db}^{corrected} &= \frac{V_{db}}{1-\beta_1^t} \end{cases}$$

$$\text{(4) RMS prop corrected : } \begin{cases} S_{dw}^{corrected} &= \frac{S_{dw}}{1-\beta_2^t} \\ S_{db}^{corrected} &= \frac{S_{db}}{1-\beta_2^t} \end{cases}$$

3. Update :

$$\begin{cases} w &= w - \alpha * \frac{V_{dw}^{corrected}}{\sqrt{S_{dw}^{corrected}} + \epsilon} \\ b &= b - \alpha * \frac{V_{db}^{corrected}}{\sqrt{S_{db}^{corrected}} + \epsilon} \end{cases}$$

4. Hyperparameters :
   - α : needs to be tunned (try a range of values)
   - β_1 : parameter of the momentum. 0.9 is a recommended value
   - β_2 : parameter of the RMS prop. 0.999 recommended by adam paper
   - ϵ : 10^{-8} is a remmended value

2.2.7 Learning rate decay

One of the things that might help speedup your learning algorithm is to slowly reduce your learning rate over time. It is called **Learning rate decay**.

→ In case you are using a fixed learning rate. Then you iterate , your steps will be a little bit noisy. And it will tend towards minumum, but never converge to it (see figure 3 line blue).

→ If you were to slowly reduce the learning rate α, then :

- in initial phases : α stills large ⇒ fast learning
- in final phases : α gets smaller ⇒ end up oscillating in a tighter region around the minimum (see figure 3 line green).

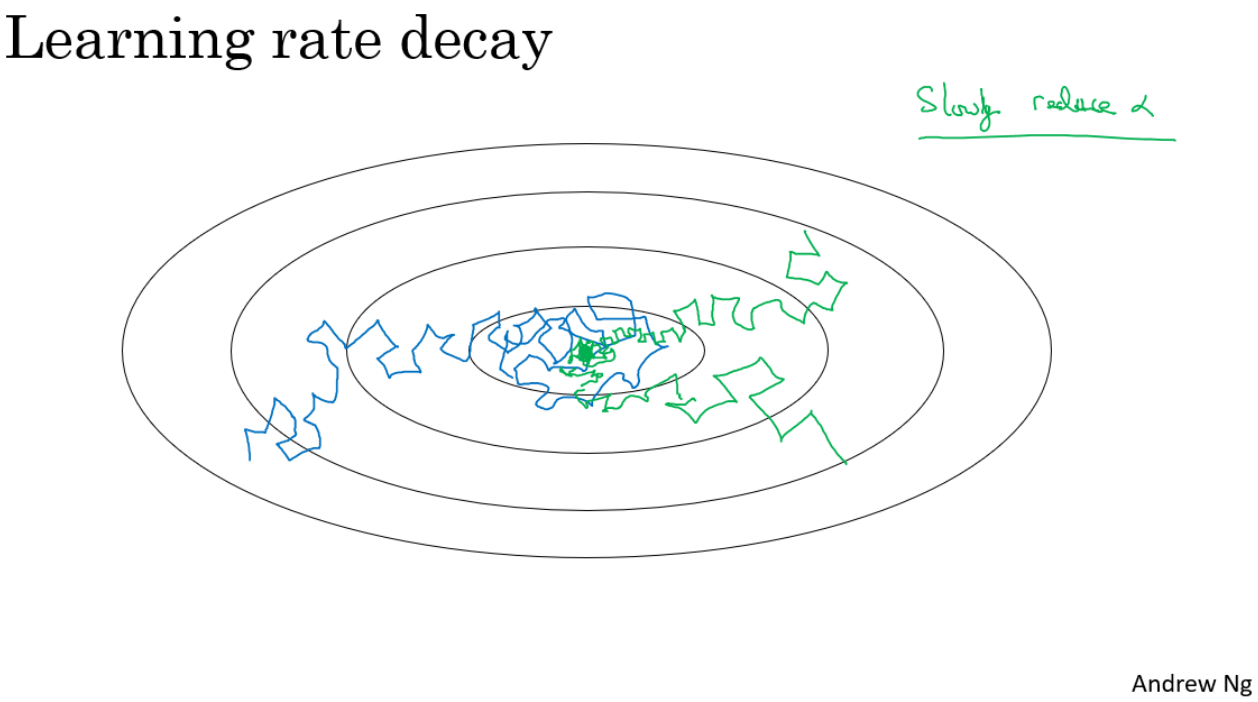


Figure 3: learning rate decay vs fixed learning rate

How to implement learning rate decay :
Recall that 1 epoch = 1 pass over the data (means also over all minibatches).
The differents ways to implement learning rate decay are listed below :

1. The first solution could be :

$$\alpha = \frac{1}{1+decay_{rate}*epoch_{num}} * \alpha_0$$

An illustration of the formula above is given in the following figure (figure 4)

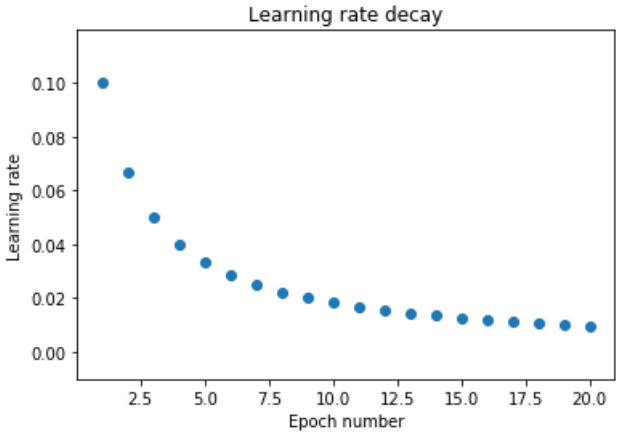


Figure 4: Evolution of learning rate with the epoch number. Done used my jupyter notebook

2. Exponentially decay :

$$\alpha = 0.95^{epoch_{number}}\alpha_0$$

3. Other formula like :

$$\alpha = \frac{k}{\sqrt{epoch_{number}}}\alpha_0$$

4. Some peaple perform learning rate decay with discret staircase
5. other peaple are making changes to the learning rate manually

To recap, decay rate is a hyperparamater that should be chosen to perform learning rate decay to achieve a good minimum for the gradient descent function. In the next week we will learn how to best choose meaningfull values for all hyperparameters.

2.2.8 The problem of local optima

- The normal local optima (left of figure 5) is not likely to appear in a deep neural network because data is usually high dimensional.
- it is unlikely to get stuck in a bad local optima in a high dimension problem, instead it is more much likely to get to the saddle point (right of figure 5), which is not a problem.
- Plateaus can make learning slow :
  1. Pleteaus is a region where the derivative is close to sero for a long time
  2. this is where algorithms like momentum, RMSprop and Adam can help.

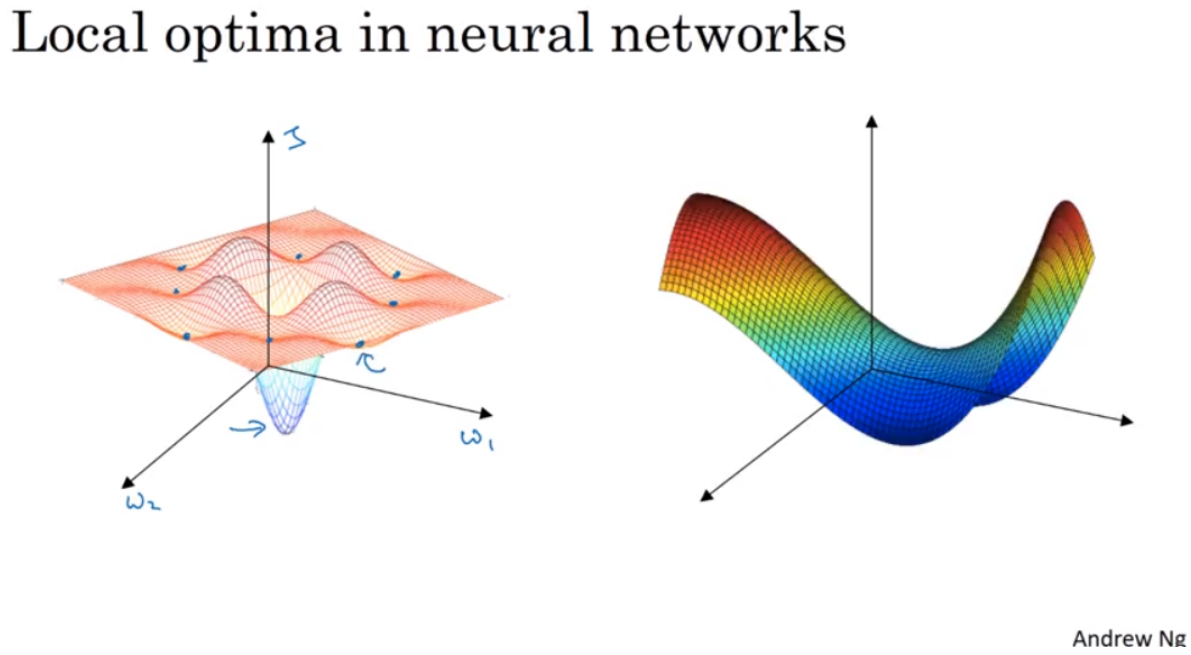


Figure 5: problem of local optima

2.3 Week 3

The objectif of this week is to master the process of hyperparameter tunning.

2.3.1 Hyperparameter tuning

(A) Tunning process : This part will give us some tips and guidlines for how to systimatically organize your hyperparameter tunning process.
What make tunning complex is the number of hyperparameters, such as :

- α
- β
- β_1
- β_2
- ϵ
- number of layers
- number of hidden units
- learning rate decay
- minibatch size

Those hyperparameters can be ordered towards their importance as follows:

1. The most important hyperparameter is : α
2. Then, comes the following list of parameters as the second in priority :

   (a) β
   (b) minibatch size
   (c) number of hidden units
3. finally :
   (a) number of layers
   (b) learning rate decay
   (c) when using adam algorithm : $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$

Now, if you are trying to tune some set of hyperparameters, how do you select a set of values to explore ?

1. In earlier generation of Machine learning : it was common to sample points in a grid like. Then explore those values and pick the best one. This still ok if the number of hyperparameters is relatively small.
2. In deep learning : try random values from the grid. This is important when you have more important parameters than others.

   Coarse to fine :
   When you sample hyperparameters, another common practice is to use a coarse to fine sampling schema.
   (a) Step (1) :

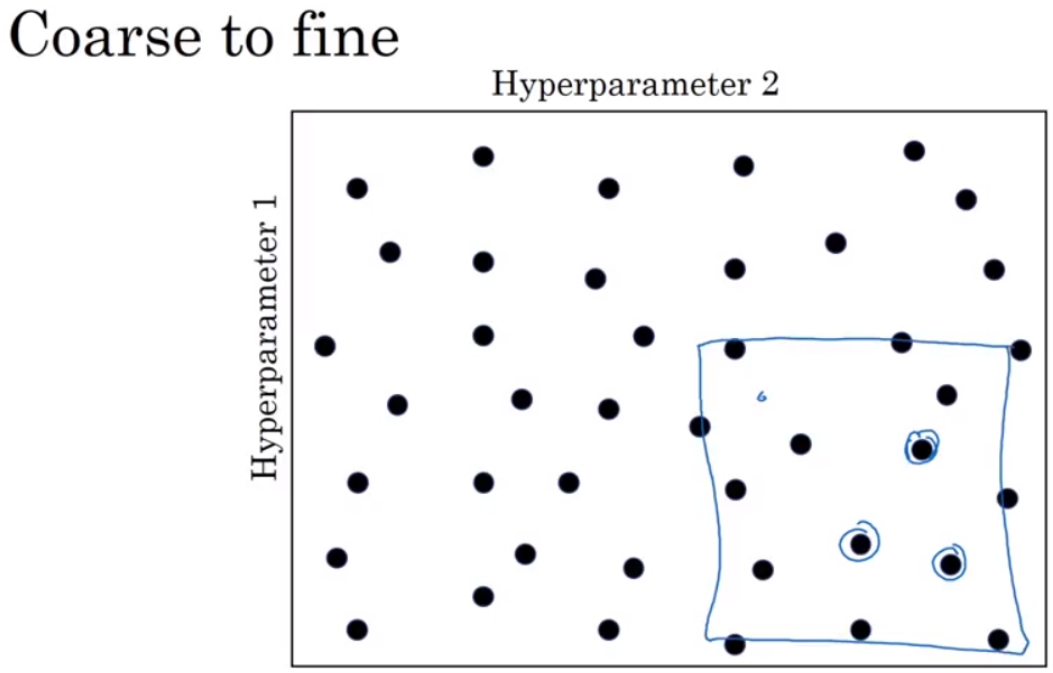


Figure 6: Coarse to fine : step (1)

after the first tunning, we may find that the values in blue box (see figure 6) are the best. So we create a mini square surrounding the values found.

(b) step (2) :

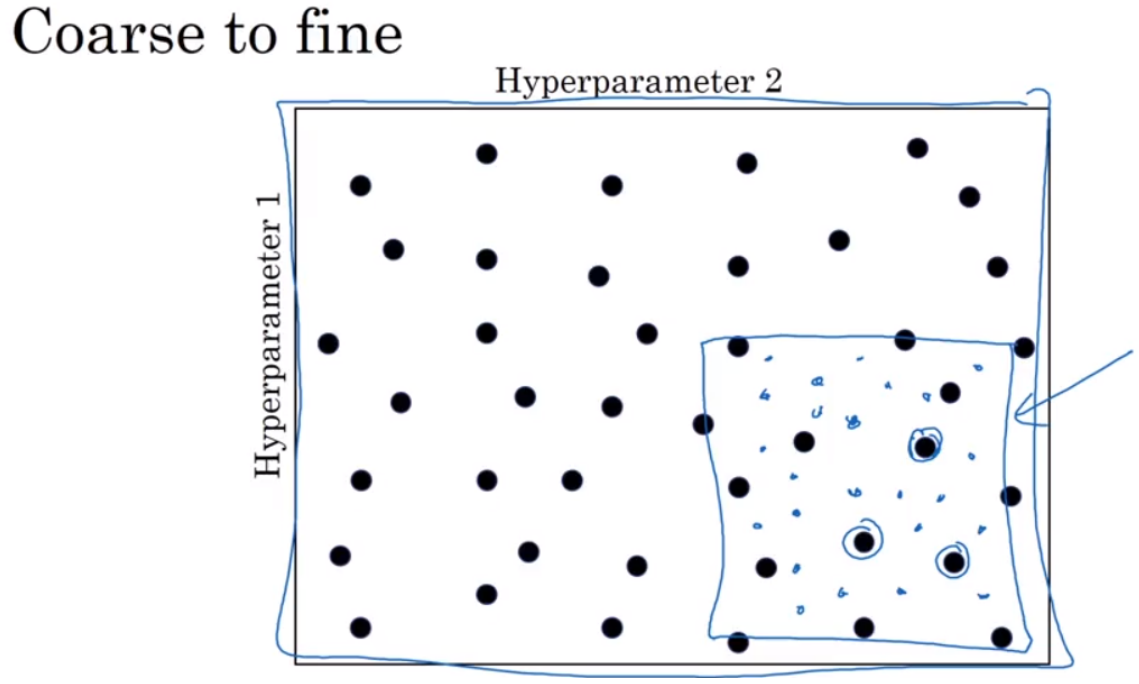


Figure 7: Coarse to fine : step (2)

Then sample more densely into smaller square and so on.

To conclude : the two key takeaways are :

1. Use random sampling and adequate search
2. Optionally consider implementing a coarse to fine search process.

(B) Using an appropriate scale to picke hyperparameters : In this paragraph we are goning to learn how to pick values for hyperparameters for tunning process.

1. Picking hyperparameters at random : for some hyperparameters, it is ok to pick up values at uniformly at random. For instance :
   $\rightarrow n^{[l]} = 50 \rightarrow 100$: taking values uniformly at random from this range will use a lot of ressources, it means it will explore a lot of values from the whole interval.
   $\rightarrow$ number of layers $L = 2 \rightarrow 4$: the same notice as previously.
2. Appropriate scale for hyperparameters : The previous method does not work for all hyperparameters. Let's work on the learning rate as an example for illustration :
   Let's say that we want to sample α from 0.0001 until 1, if we use the last method that consist on choosing value uniformly at random the most picked values will be between 0.1 and 1 ($\approx$ 90%). So we are not exploring the whole ressources (see figure below) :

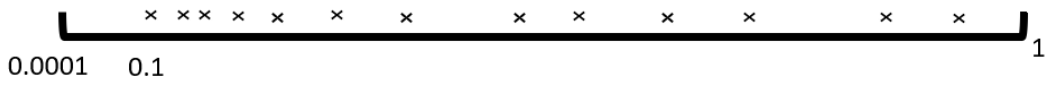


Figure 8: The drawback of using uniformly at random values for the learning rate

That's why it is more raisonable to search in a log scale :

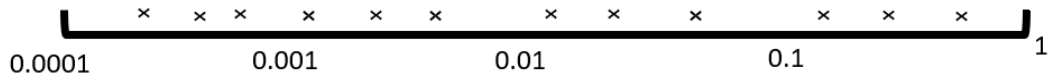


Figure 9: The advantage of using log scale for the learning rate

In python :

$r = -4 * np.random.rand() \rightarrow r \in [-4, 0]$
$\alpha = 10^r \rightarrow \alpha \in [10^{-4}, 10^0]$

In a more general way, to pick values from an interval whose values go from min until max :
→ we compute : $a = \log_{10}(min)$ and $b = \log_{10}(max)$
→ then we get the interval [a,b] uniformly at random
→ finally $\alpha = 10^r$

3. hyperparameters for exponentially weighted averages : the figure below (taken from the course) shows how to pick values for exponentially weighted averages.

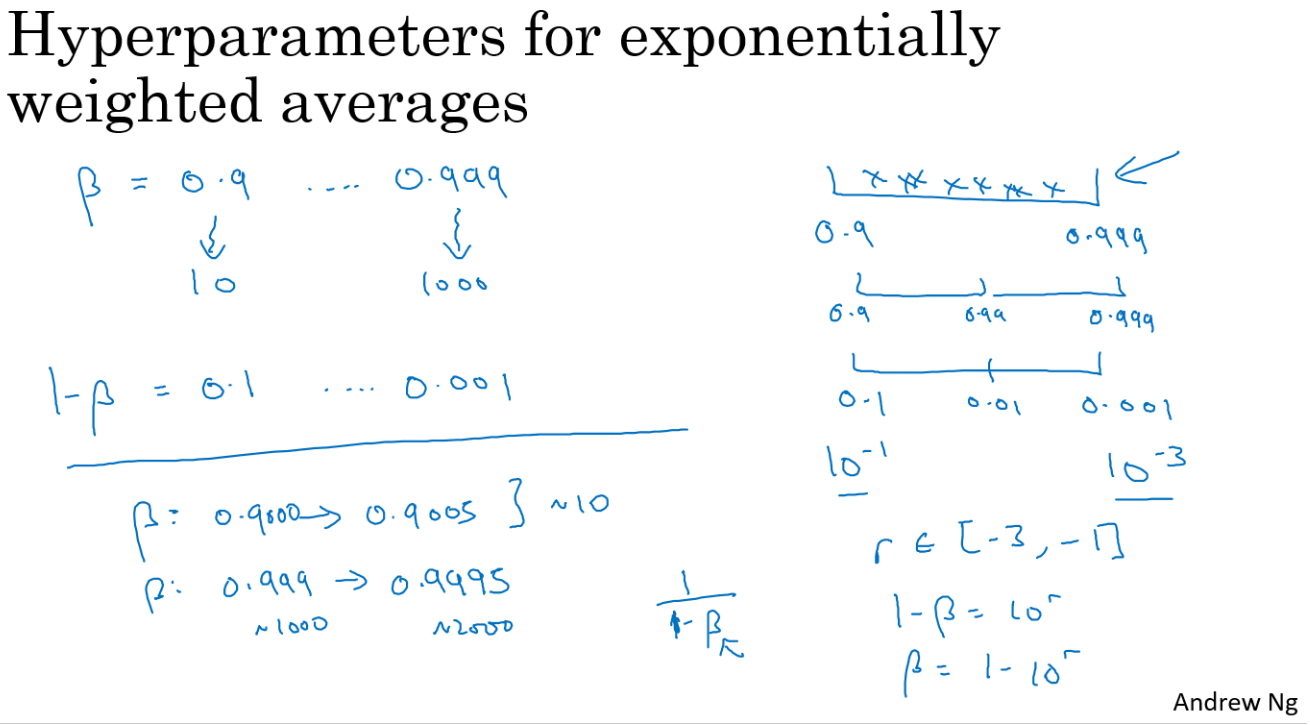


Figure 10: hyperparameters for exponentially weighted averages

By this way, we are exploring much ressources from 0.9 → 0.999.

(C) Hyperparameters tunning in practice : Pandas vs Caviar :
→ Deep learning today is applied to many different application areas.
→ Intuitions about hyperparameters settings from one application area may or may not transfer to a different one
→ Some intuitions can be applied in different application. Eg : Confenets or Resnet are used in computer vision, but still successfully applied in speech domain.
→ Intuitions do get stale. Re-evaluate occasionally. This may be due to data changes, upgraded servers in data centers, etc. So to make sur that the old hyperparameters still work well, we should re-evaluate them in new changes.
→ There are two methods for searching hyperparameters :

1. Babysitting one model : this method is used wehn we have a huge dataset but not enough computational resources, not a lot of CPUs and GPUs ⇒ so you can only afford to train one model (see left of figure 11).
   in this method, day by day you try nudging up and down your hyperparameters.
   ⇒ you are kind of babysitting the model.
2. Training many models in parallel : This method is used when you have enough computational resources. Then choose the best model (see right of figure 11)

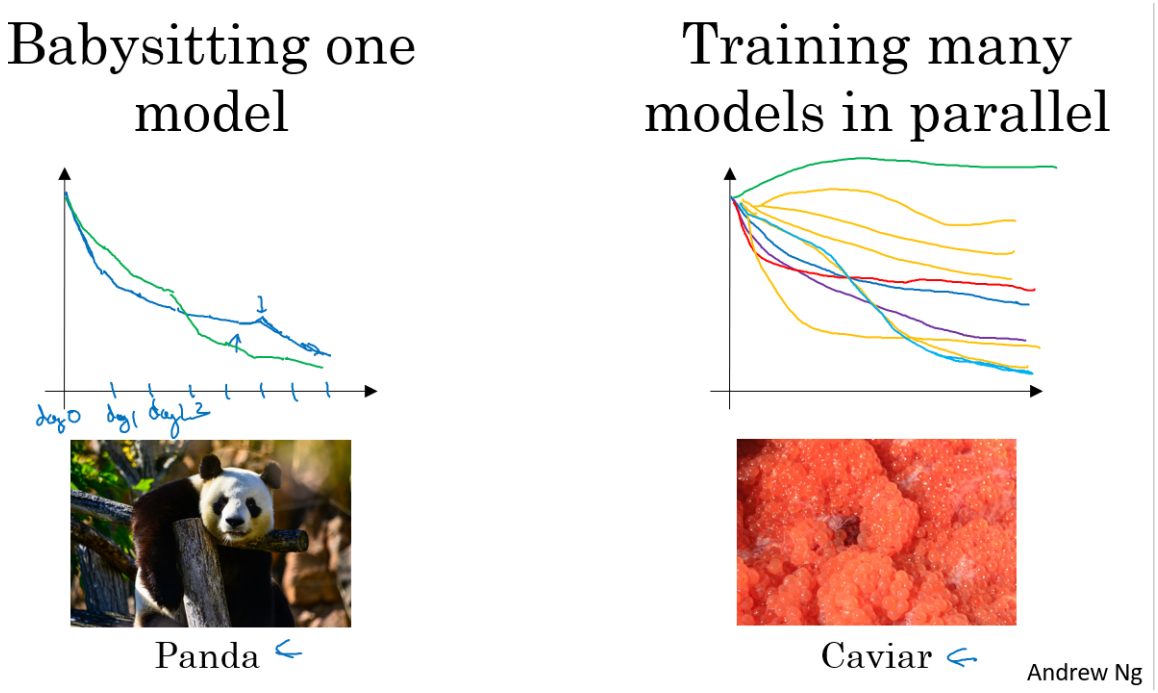


Figure 11: Methods to choose hyperparameters

2.3.2 Batch Normalization

(A) Normalizing activations in a network : In logistic regression we saw that normalizing features speed up learning.

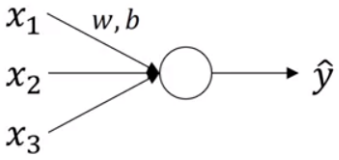


Figure 12: Logistic regression

$$\begin{cases} \mu &= \frac{1}{m}\sum_{i=1} X^{(i)} \\ X &= X - \mu \\ \sigma^2 &= \frac{1}{m}\sum_{i=1}(X^{(i)})^2 \\ X &= \frac{X}{\sigma^2} \end{cases}$$

In deep neural networks, in addition to input features we have activations terms.

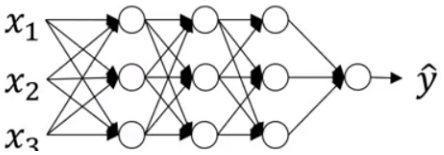


Figure 13: Deep Neural Netwrok

The questions is : Can we noramlize $a^{[l]}$ so as to train $W^{[l+1]}$ and $B^{[l+1]}$ faster ?

Notice : in fact and in practice we normalize $Z^{[l]}$. But there is a debate in deep learning litterature about which values should normalize either $Z^{[l]}$ or $a^{[l]}$.

Implementing Batch norm :
Given some intermediate values in NN for a certain layer : $Z^{(1)}$, ... $Z^{(l)}$:

$$\begin{cases} \mu &= \frac{1}{m}\sum_{i=1} Z^{(i)} \\ \sigma^2 &= \frac{1}{m}\sum_{i=1}(Z^{(i)} - \mu)^2 \\ Z^{(i)}_{norm} &= \frac{Z^{(i)} - \mu}{\sqrt{\sigma^2 + \epsilon}} \\ \tilde{Z}^{(i)} &= \gamma Z^{(i)}_{norm} + \beta \end{cases}$$

Where γ and β are learnable parameters of model.
So for the later computation in the NN, instead of using $Z^{(i)}$, we will use $\tilde{Z}^{(i)}$.

(B) Fitting Batch Norm into a neural network :
Normalizing activations in a network : knowing form previous paragraph how to implement Batch normalization (BN), let's see how to add BN to a network. The following figure illustrate computation of BN among a Deep NN :

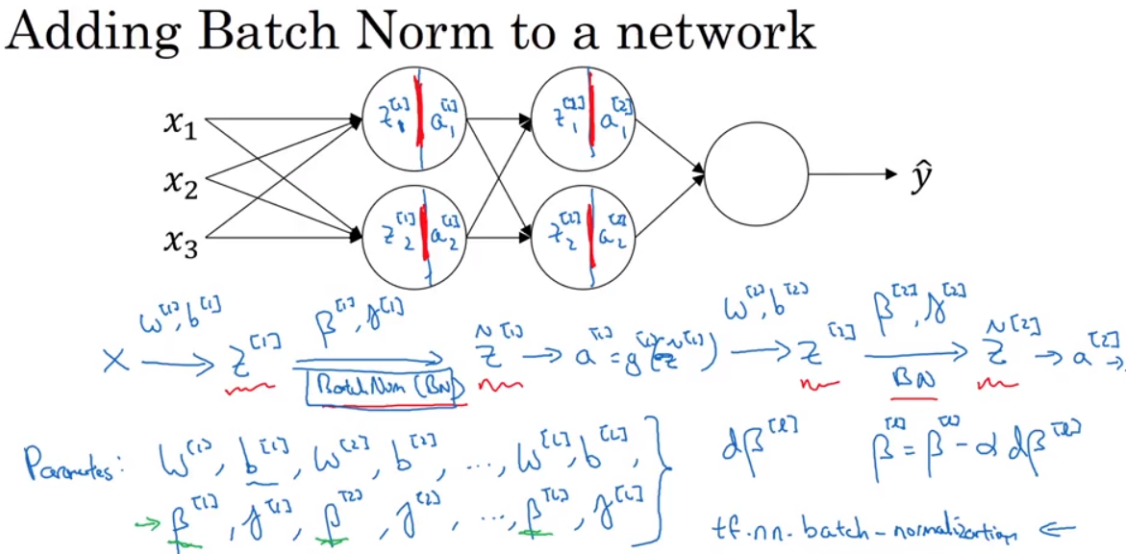


Figure 14: Deep Neural Netwrok Computation

Working with mini-batches : Batch normalization usually used with mini-batches

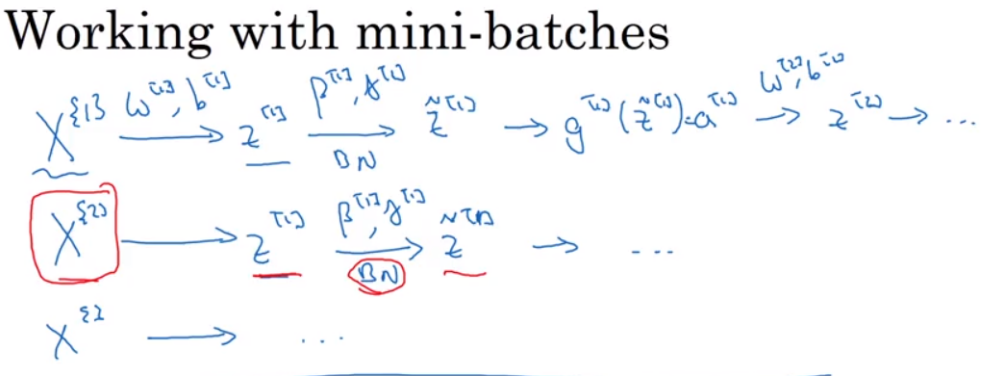


Figure 15: BN with minibatches

Implementing gradient descent :

for $t = 1 \rightarrow num_{minibatches}$:

- compute forward propagation on X^t :
  In each hidden layer, use BN to replace $Z^{[l]}$ with $\tilde{Z}^{[l]}$
- Use backprop to compute $dw^{[l]}$, $d\beta^{[l]}$ and $d\gamma^{[l]}$
- update parameters :

$$\begin{cases} W^{[l]} &= W^{[l]} - \alpha * dW^{[l]} \\ \beta^{[l]} &= \beta^{[l]} - \alpha * d\beta^{[l]} \\ \gamma^{[l]} &= \gamma^{[l]} - \alpha * d\gamma^{[l]} \end{cases}$$

(C) Why does Batch Norm work?

1. The first reason is like what we have seen for normalizing input features X. It gives similar ranges to all inputs, therefore it speeds up the learning. So batch norm does the same thing not only for inputs but for also the hidden layers.
2. The second reason is that batch normalization reduces the problem of input values changing (shifting).
3. Batch normalization does some regularization :
   (a) Each mini batch is scaled by the mean/variance computed of that mini-batch.
   (b) This adds some noise to the values $Z^{[l]}$ within that mini batch. So similar to dropout it adds some noise to each hidden layer's activations.
   (c) This has a slight regularization effect.
   (d) Using bigger size of the mini-batch you are reducing noise and therefore regularization effect.
   (e) Don't rely on batch normalization as a regularization. It's intended for normalization of hidden units, activations and therefore speeding up learning. For regularization use other regularization techniques (L2 or dropout).

(D) Batch Norm at test time

1. When we train a NN with Batch normalization, we compute the mean and the variance of the mini-batch.
2. In testing we might need to process examples one at a time. The mean and the variance of one example won't make sense.
3. We have to compute an estimated value of mean and variance to use it in testing time.
4. We can use the weighted average across the mini-batches.
5. We will use the estimated values of the mean and variance to test.
6. This method is also sometimes called "Running average".
7. In practice most often you will use a deep learning framework and it will contain some default implementation of doing such a thing.

2.3.3 Multi-class classification

So far, the classification examples we have talked about have used binary classification where you had two possible labels, 0 or 1.
What if we have multiple possible classes ?
There is a generalization of logistic regression called softmax regression that lets you make predictions when you try to predict or recognize one of the multiple classes.

(A) Softmax Regression Notation : C will denote the number of classes presented in our classifiaction problem.
Example : the following problem has a $C = 4$. So classes will be denoted 0,1,2,3.

Figure 16: Example for multi-classes problem

So for our Neural network, the last layer will output a $4 * 1$ vector as illustrated in the following figure :

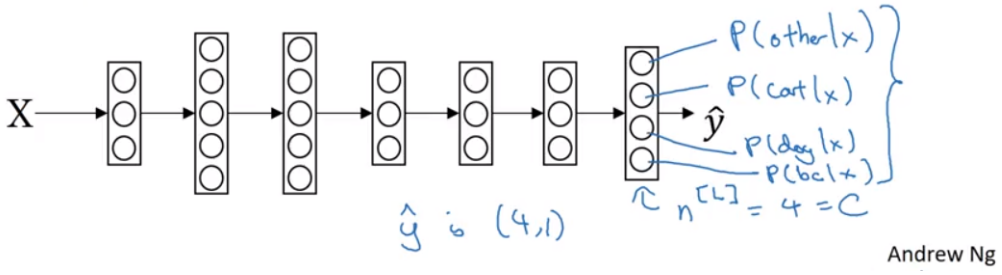


Figure 17: Output for a NN in multi-classes problem

The standard model that allows us to do this is called softmax layer.
Softmax layer :

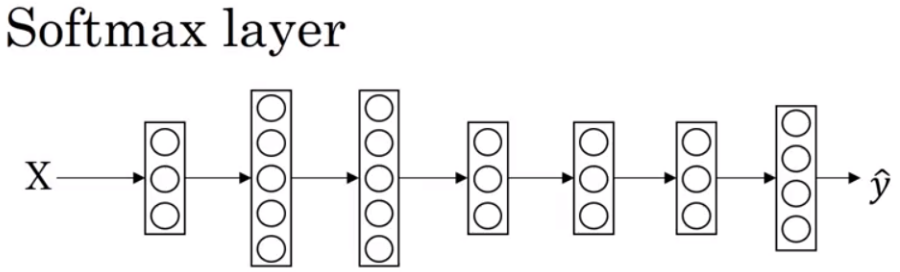


Figure 18: softmax layer

For the last layer we would have the following equations :

1. $Z^{[L]} = W^{[L]} * a^{[L]} + b^{[L]} \leftarrow$ (4,1)
2. Activation function :
   (a) $t = \exp(Z^{[L]}) \leftarrow$ (4,1)
   (b) $a^{[L]} = \frac{\exp(Z^{[L]})}{\sum_{j=1}^{4} t_i} \leftarrow$ (4,1)

Example :

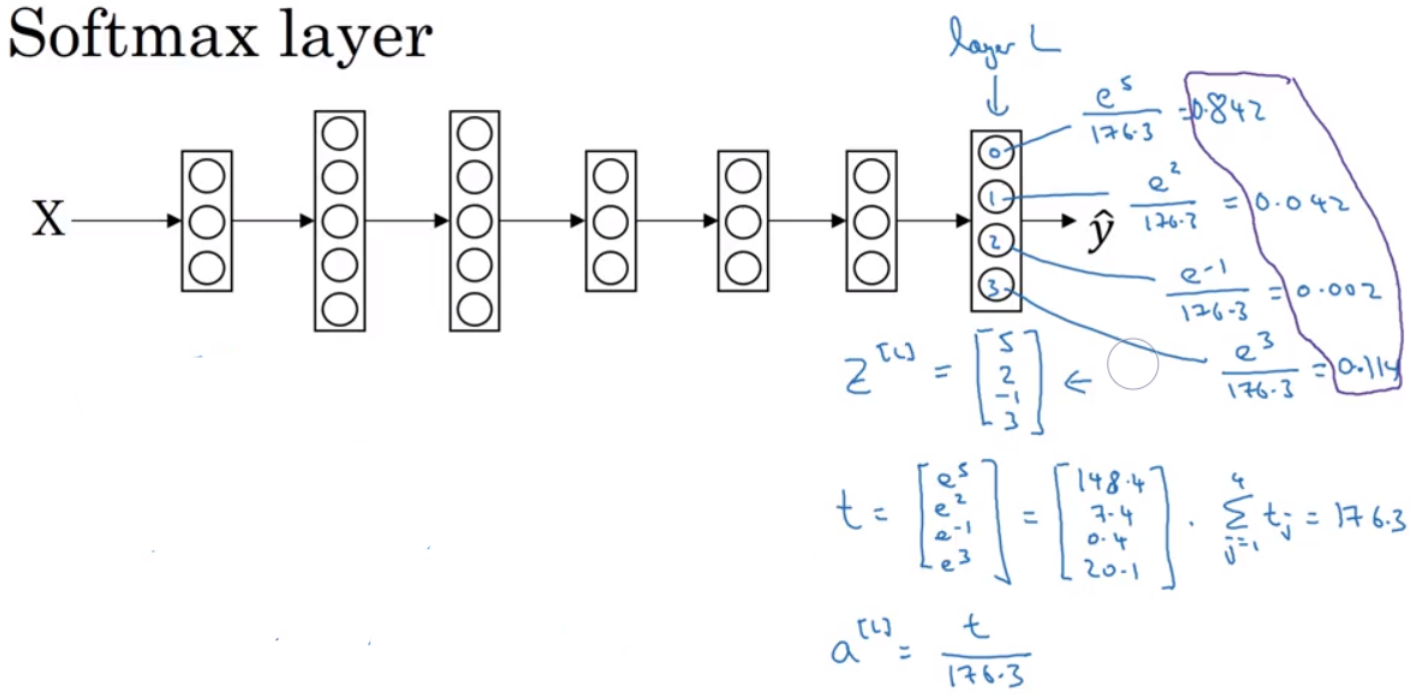


Figure 19: softmax layer example

(B) Training a softmax classifier :

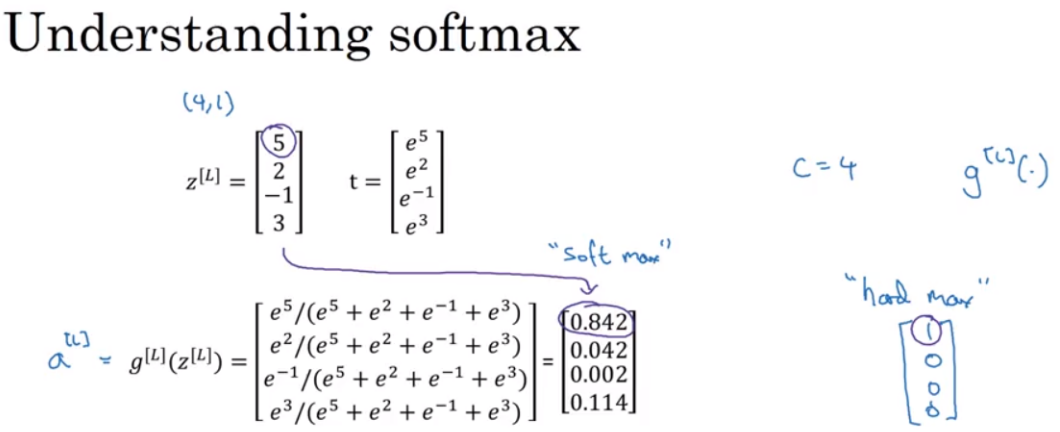


Figure 20: Understanding softmax

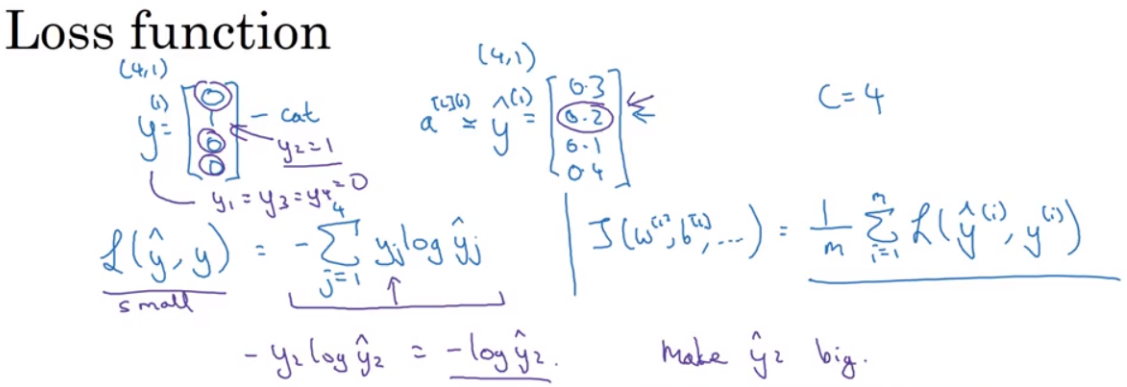


Figure 21: loss function

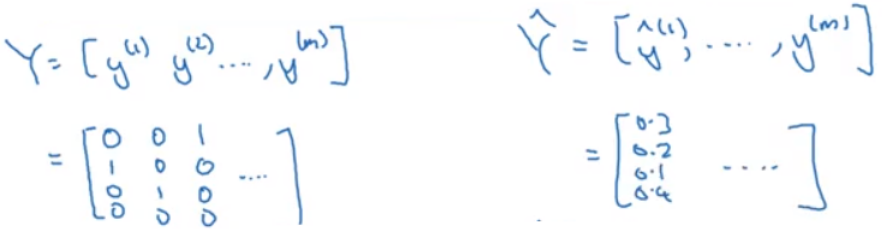


Figure 22: Y and Yhat representation

2.3.4 Introduction to programming frameworks

(A) Deep learning frameworks :

1. We have implemented some neural networks from scratch. It was useful to understand how exactly NN are doing.
2. But for more complex models like CNN, etc it is will be complicated to implement them from scratch.
3. Fortunately, there are now many good deep learning frameworks to do that.
4. Today there are many deep learning frameworks :
   (a) Caffe/Caffe2
   (b) CNTK
   (c) DL4J
   (d) Keras
   (e) Lasagne
   (f) mxnet
   (g) PaddlePaddle
   (h) TensorFlow
   (i) Theano
   (j) Torch
5. These frameworks are getting better month by month. Comparison between them can be found here.
6. Each of these framework is a credible choice for some subset of applications
7. Criteria that Andrew Ng recommend to take into consideration when choosing a DL framework are :
   - Ease of programming (development and deployment)
   - Running speed
   - Truly open (open source with good governance)
8. Programming frameworks can not only shorten your coding time but sometimes also perform optimizations that speed up your code.

(B) TensorFlow :
The programming assignment is very well done as an introduction for the Tenserflow framework. More than that it is applied in a real problem about the SIGNS dataset, where you will build a deep neural network model to recognize numbers from 0 to 5 in sign language with a pretty impressive accuracy.

3 Bibliography

Coursera plateforme

Mahmoud Badry notes

Beautifully drawn notes by Tess Ferrandez