Course summary

Here are the course summary as its given on the course link:

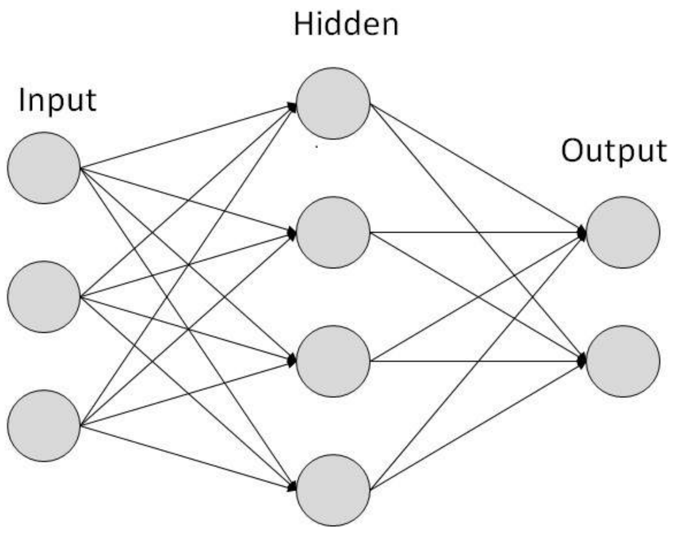
> If you want to break into cutting-edge AI, this course will help you do so. Deep learning engineers are highly sought after, and mastering deep learning will give you numerous new career opportunities. Deep learning is also a new "superpower" that will let you build AI systems that just weren't possible 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.

Introduction to deep learning

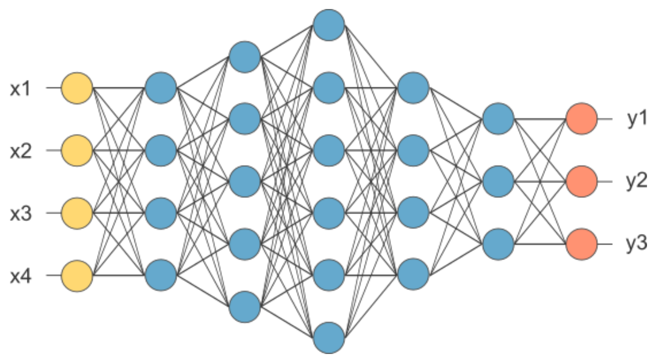
> Be able to explain the major trends driving the rise of deep learning, and understand where and how it is applied today.

What is a (Neural Network) NN?

- Single neuron == linear regression without applying activation(perceptron)
- Basically a single neuron will calculate weighted sum of input(W.T*X) and then we can set a threshold to predict output in a perceptron. If weighted sum of input cross the threshold, perceptron fires and if not then perceptron doesn't predict.
- Perceptron can take real values input or boolean values.
- Actually, when w·x+b=0 the perceptron outputs 0.
- Disadvantage of perceptron is that it only output binary values and if we try to give small change in weight and bais then perceptron can flip the output. We need some system which can modify the output slightly according to small change in weight and bias. Here comes sigmoid function in picture.
- If we change perceptron with a sigmoid function, then we can make slight change in output.
- e.g. output in perceptron = 0, you slightly changed weight and bias, output becomes = 1 but actual output is 0.7. In case of sigmoid, output1 = 0, slight change in weight and bias, output = 0.7.
- If we apply sigmoid activation function then Single neuron will act as Logistic Regression.
- we can understand difference between perceptron and sigmoid function by looking at sigmoid function graph.
- Simple NN graph:



  -
  - Image taken from tutorialspoint.com
- RELU stands for rectified linear unit is the most popular activation function right now that makes deep NNs train faster now.
- Hidden layers predicts connection between inputs automatically, thats what deep learning is good at.

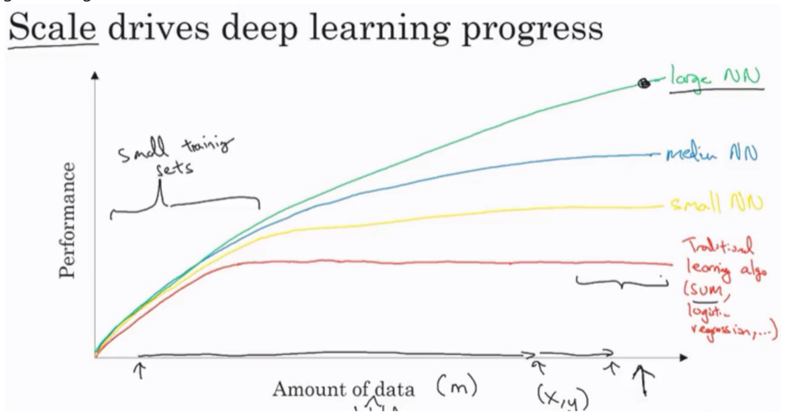
- Deep NN consists of more hidden layers (Deeper layers)
    - 

    - Image taken from opennn.net
- Each Input will be connected to the hidden layer and the NN will decide the connections.
- Supervised learning means we have the (X,Y) and we need to get the function that maps X to Y.

Supervised learning with neural networks

- Different types of neural networks for supervised learning which includes:
    - CNN or convolutional neural networks (Useful in computer vision)
    - RNN or Recurrent neural networks (Useful in Speech recognition or NLP)
    - Standard NN (Useful for Structured data)
    - Hybrid/custom NN or a Collection of NNs types
- Structured data is like the databases and tables.
- Unstructured data is like images, video, audio, and text.
- Structured data gives more money because companies relies on prediction on its big data.

Why is deep learning taking off?

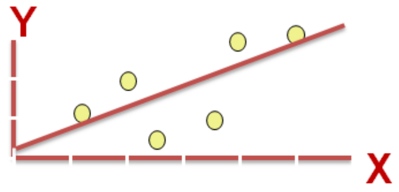
- Deep learning is taking off for 3 reasons:
    i. Data:
        - Using this image we can conclude:
        - 

        - For small data NN can perform as Linear regression or SVM (Support vector machine)
        - For big data a small NN is better that SVM
        - For big data a big NN is better that a medium NN is better that small NN.
        - Hopefully we have a lot of data because the world is using the computer a little bit more
            - Mobiles
            - IOT (Internet of things)
    ii. Computation:
        - GPUs.
        - Powerful CPUs.
        - Distributed computing.
        - ASICs
    iii. Algorithm:
        a. 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.

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.

Binary classification

- Mainly he is talking about how to do a logistic regression to make a binary classifier.
    - 

    - Image taken from 3.bp.blogspot.com
- He talked about an example of knowing if the current image contains a cat or not.
- Here are some notations:
    - `M is the number of training vectors`
    - `Nx is the size of the input vector`
    - `Ny is the size of the output vector`
    - `X(1) is the first input vector`
    - `Y(1) is the first output vector`
    - `X = [x(1) x(2).. x(M)]`
    - `Y = (y(1) y(2).. y(M))`
- We will use python in this course.
- In NumPy we can make matrices and make operations on them in a fast and reliable time.

Logistic regression

- Algorithm is used for classification algorithm of 2 classes.
- Equations:
    - Simple equation: `y = wx + b`
    - If x is a vector: `y = w(transpose)x + b`
    - If we need y to be in between 0 and 1 (probability): `y = sigmoid(w(transpose)x + b)`
    - In some notations this might be used: `y = sigmoid(w(transpose)x)`
        - While `b` is `w0` of `w` and we add `x0 = 1`. but we won't use this notation in the course (Andrew said that the first notation is better).
- In binary classification `Y` has to be between `0` and `1`.
- In the last equation `w` is a vector of `Nx` and `b` is a real number

Logistic regression cost function

- First loss function would be the square root error: `L(y',y) = 1/2 (y' - y)^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.
- This is the function that we will use: `L(y',y) = - (y*log(y') + (1-y)*log(1-y'))`
- To explain the last function lets see:
    - if `y = 1` ==> `L(y',1) = -log(y')` ==> we want `y'` to be the largest ==> `y`' biggest value is 1
    - if `y = 0` ==> `L(y',0) = -log(1-y')` ==> we want `1-y'` to be the largest ==> `y'` to be smaller as possible because it can only has 1 value.
- Then the Cost function will be: `J(w,b) = (1/m) * Sum(L(y'[i],y[i]))`
- The loss function computes the error for a single training example; the cost function is the average of the loss functions of the entire training set.

Gradient Descent

- We want to predict `w` and `b` that minimize the cost function.
- Our cost function is convex.
- 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 alpha is the learning rate and `dw` is the derivative of `w` (Change to `w`) The derivative is also the slope of `w`
- Looks like greedy algorithms. the derivative give us the direction to improve our parameters.

- The actual equations we will implement:
  - `w = w - alpha * d(J(w,b) / dw)` (how much the function slopes in the w direction)
  - `b = b - alpha * d(J(w,b) / db)` (how much the function slopes in the d direction)

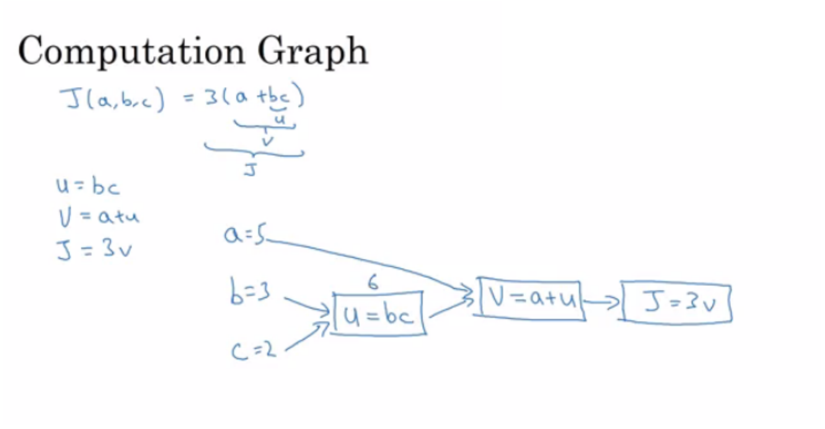
Derivatives

- We will talk about some of required calculus.
- You don't need to be a calculus geek to master deep learning but you'll need some skills from it.
- Derivative of a linear line is its slope.
  - ex. `f(a) = 3a` `d(f(a))/d(a) = 3`
  - if `a = 2` then `f(a) = 6`
  - if we move a a little bit `a = 2.001` then `f(a) = 6.003` means that we multiplied the derivative (Slope) to the moved area and added it to the last result.

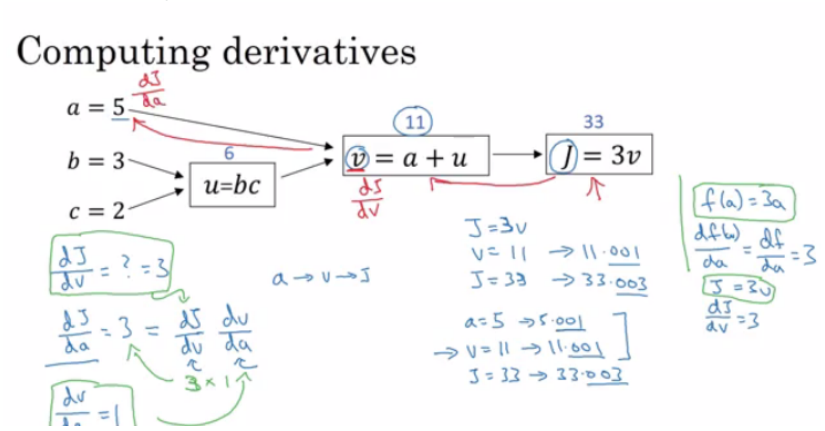
More Derivatives examples

- `f(a) = a^2` `==>` `d(f(a))/d(a) = 2a`
  - `a = 2` `==>` `f(a) = 4`
  - `a = 2.0001` `==>` `f(a) = 4.0004` approx.
- `f(a) = a^3` `==>` `d(f(a))/d(a) = 3a^2`
- `f(a) = log(a)` `==>` `d(f(a))/d(a) = 1/a`
- To conclude, Derivative is the slope and slope is different in different points in the function thats why the derivative is a function.

Computation graph

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

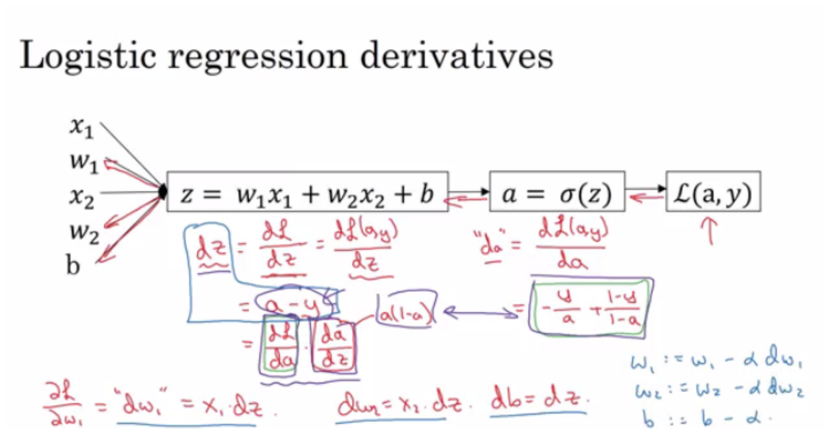
Derivatives with a Computation Graph

- Calculus chain rule says: If `x -> y -> z` (x effect y and y effects z) Then `d(z)/d(x) = d(z)/d(y) * d(y)/d(x)`
- The video illustrates a big example.
  - 

- 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 variable with respect to various intermediate quantities.

Logistic Regression Gradient Descent

- In the video he discussed the derivatives of gradient decent example for one sample with two features `x1` and `x2`.

- 
Logistic regression derivatives
x_1
w_1
x_2
w_2
b
$z = w_1x_1 + w_2x_2 + b$
$a = \sigma(z)$
$\mathcal{L}(a,y)$
$dz = \frac{dL}{dz} = \frac{dL(a,y)}{dz}$
$= a - y$
$= \frac{dL}{da} \cdot \frac{da}{dz}$
$a(1-a)$
"da" $= \frac{dL(a,y)}{da}$
$= -\frac{y}{a} + \frac{1-y}{1-a}$
$\frac{\partial L}{\partial w_1}$ = "dw_1" $= x_1 \cdot dz$
$dw_2 = x_2 \cdot dz$
$db = dz$
$w_1 := w_1 - \alpha dw_1$
$w_2 := w_2 - \alpha dw_2$
$b := b - \alpha$

Gradient Descent on m Examples

- Lets say we have these variables:

```
	X1                  				Feature
	X2                  Feature
	W1                  Weight of the first feature.
	W2                  Weight of the second feature.
	B                   Logistic Regression parameter.
	M                   Number of training examples
	Y(i)                			Expected output of i
```

- So we have:

```
X1  \
W1   \
X2   ===>  z(i) = X1W1 + X2W2 + B ===> a(i) = Sigmoid(z(i)) ===> l(a(i),Y(i)) = - (Y(i)*log(a(i)) + (1-Y(i))*log(1-a(i)))
Y2   /
B   /
```

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

```
	d(a)  = d(l)/d(a) = -(y/a) + ((1-y)/(1-a))
	d(z)  = d(l)/d(z) = a - y
	d(W1) = X1 * d(z)
	d(W2) = X2 * d(z)
	d(B) = d(z)
```

- From the above we can conclude the logistic regression pseudo code:

```
	J = 0; dw1 = 0; dw2 =0; db = 0;                 # Devs.
	w1 = 0; w2 = 0; b=0;                                                    # Weights
	for i = 1 to m
		# Forward pass
		z(i) = W1*x1(i) + W2*x2(i) + b
		a(i) = Sigmoid(z(i))
		J += (Y(i)*log(a(i)) + (1-Y(i))*log(1-a(i)))

		# Backward pass
		dz(i) = a(i) - Y(i)
		dw1 += dz(i) * x1(i)
		dw2 += dz(i) * x2(i)
		db  += dz(i)
	J /= m
	dw1/= m
	dw2/= m
	db/= m

	# Gradient descent
	w1 = w1 - alpa * dw1
	w2 = w2 - alpa * dw2
	b = b - alpa * db
```

- The above code should run for some iterations to minimize 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

- We will implement Logistic Regression using one for loop then without any for loop.
- As an input we have a matrix `X` and its `[Nx, m]` and a matrix `Y` and its `[Ny, m]`.
- We will then compute at instance `[z1,z2...zm] = W' * X + [b,b,...b]`. This can be written in python as:

```
        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)
```

- Vectorizing Logistic Regression's Gradient Output:

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

Notes on Python and NumPy

- In NumPy, `obj.sum(axis = 0)` sums the columns while `obj.sum(axis = 1)` sums the rows.
- In NumPy, `obj.reshape(1,4)` changes the shape of the matrix by broadcasting the values.
- Reshape is cheap in calculations so put it everywhere you're not sure about the calculations.
- Broadcasting works when you do a matrix operation with matrices that doesn't match for the operation, in this case NumPy automatically makes the shapes ready for the operation by broadcasting the values.
- In general principle of broadcasting. If you have an (m,n) matrix and you add(+) or subtract(-) or multiply(*) or divide(/) with a (1,n) matrix, then this will copy it m times into an (m,n) matrix. The same with if you use those operations with a (m , 1) matrix, then this will copy it n times into (m, n) matrix. And then apply the addition, subtraction, and multiplication of division element wise.
- Some tricks to eliminate all the strange bugs in the code:
  - If you didn't specify the shape of a vector, it will take a shape of `(m,)` and the transpose operation won't work. You have to reshape it to `(m, 1)`
  - Try to not use the rank one matrix in ANN
  - Don't hesitate to use `assert(a.shape == (5,1))` to check if your matrix shape is the required one.
  - If you've found a rank one matrix try to run reshape on it.
- Jupyter / IPython notebooks are so useful library in python that makes it easy to integrate code and document at the same time. It runs in the browser and doesn't need an IDE to run.
  - To open Jupyter Notebook, open the command line and call: `jupyter-notebook` It should be installed to work.
- To Compute the derivative of Sigmoid:

```
        s = sigmoid(x)
        ds = s * (1 - s)       # derivative  using calculus
```

- To make an image of `(width,height,depth)` be a vector, use this:

```
v = image.reshape(image.shape[0]*image.shape[1]*image.shape[2],1)  #reshapes the image.
```

- Gradient descent converges faster after normalization of the input matrices.

General Notes

- The main steps for building a Neural Network are:
  - Define the model structure (such as number of input features and outputs)
  - Initialize the model's parameters.
  - Loop.
    - Calculate current loss (forward propagation)
    - Calculate current gradient (backward propagation)
    - Update parameters (gradient descent)
- Preprocessing the dataset is important.
- Tuning the learning rate (which is an example of a "hyperparameter") can make a big difference to the algorithm.

- kaggle.com is a good place for datasets and competitions.
- Pieter Abbeel is one of the best in deep reinforcement learning.

Shallow neural networks

> Learn to build a neural network with one hidden layer, using forward propagation and backpropagation.

Neural Networks Overview

- In logistic regression we had:

```
X1  \
X2   ==>  z = XW + B ==> a = Sigmoid(z) ==> l(a,Y)
X3  /
```

- In neural networks with one layer we will have:

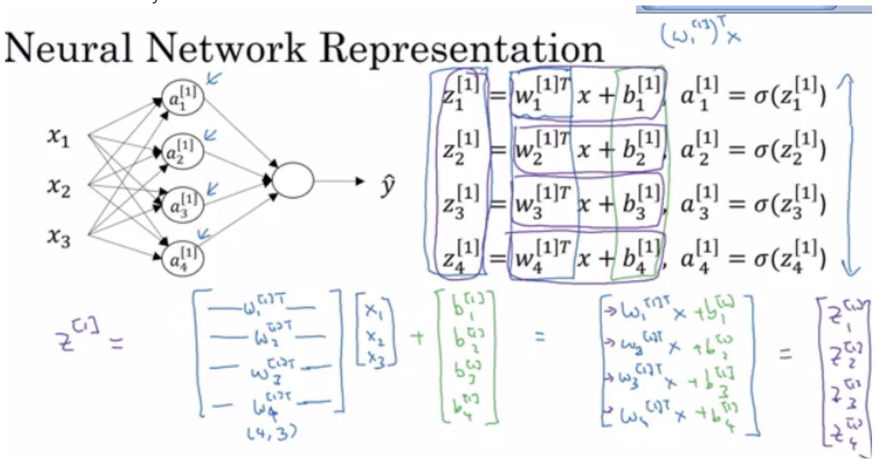
```
X1  \
X2   =>  z1 = XW1 + B1 => a1 = Sigmoid(z1) => z2 = a1W2 + B2 => a2 = Sigmoid(z2) => l(a2,Y)
X3  /
```

- `X` is the input vector `(X1, X2, X3)`, and `Y` is the output variable `(1x1)`
- NN is stack of logistic regression objects.

Neural Network Representation

- We will define the neural networks that has one hidden layer.
- NN contains of input layers, hidden layers, output layers.
- Hidden layer means we cant see that layers in the training set.
- `a0 = x` (the input layer)
- `a1` will represent the activation of the hidden neurons.
- `a2` will represent the output layer.
- We are talking about 2 layers NN. The input layer isn't counted.

Computing a Neural Network's Output

- Equations of Hidden layers:
  - 
    Neural Network Representation
$(w_1^{[1]})^T x$
x_1
x_2
x_3
$a_1^{[1]}$
$a_2^{[1]}$
$a_3^{[1]}$
$a_4^{[1]}$
$\hat{y}$
$z_1^{[1]} = w_1^{[1]T} x + b_1^{[1]}, \quad a_1^{[1]} = \sigma(z_1^{[1]})$
$z_2^{[1]} = w_2^{[1]T} x + b_2^{[1]}, \quad a_2^{[1]} = \sigma(z_2^{[1]})$
$z_3^{[1]} = w_3^{[1]T} x + b_3^{[1]}, \quad a_3^{[1]} = \sigma(z_3^{[1]})$
$z_4^{[1]} = w_4^{[1]T} x + b_4^{[1]}, \quad a_4^{[1]} = \sigma(z_4^{[1]})$
$z^{[1]} = \underbrace{\begin{bmatrix} - w_1^{[1]T} - \\ - w_2^{[1]T} - \\ - w_3^{[1]T} - \\ - w_4^{[1]T} - \end{bmatrix}}_{(4,3)} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \\ b_4^{[1]} \end{bmatrix} = \begin{bmatrix} w_1^{[1]T} x + b_1^{[1]} \\ w_2^{[1]T} x + b_2^{[1]} \\ w_3^{[1]T} x + b_3^{[1]} \\ w_4^{[1]T} x + b_4^{[1]} \end{bmatrix} = \begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \\ z_4^{[1]} \end{bmatrix}$
- Here are some informations about the last image:
  - `noOfHiddenNeurons = 4`
  - `Nx = 3`
  - Shapes of the variables:
    - `W1` is the matrix of the first hidden layer, it has a shape of `(noOfHiddenNeurons,nx)`
    - `b1` is the matrix of the first hidden layer, it has a shape of `(noOfHiddenNeurons,1)`
    - `z1` is the result of the equation `z1 = W1*X + b`, it has a shape of `(noOfHiddenNeurons,1)`
    - `a1` is the result of the equation `a1 = sigmoid(z1)`, it has a shape of `(noOfHiddenNeurons,1)`
    - `W2` is the matrix of the second hidden layer, it has a shape of `(1,noOfHiddenNeurons)`
    - `b2` is the matrix of the second hidden layer, it has a shape of `(1,1)`
    - `z2` is the result of the equation `z2 = W2*a1 + b`, it has a shape of `(1,1)`
    - `a2` is the result of the equation `a2 = sigmoid(z2)`, it has a shape of `(1,1)`

Vectorizing across multiple examples

- Pseudo code for forward propagation for the 2 layers NN:

```
for i = 1 to m
  z[1, i] = W1*x[i] + b1      # shape of z[1, i] is (noOfHiddenNeurons,1)
  a[1, i] = sigmoid(z[1, i])  # shape of a[1, i] is (noOfHiddenNeurons,1)
  z[2, i] = W2*a[1, i] + b2   # shape of z[2, i] is (1,1)
  a[2, i] = sigmoid(z[2, i])  # shape of a[2, i] is (1,1)
```

- Lets say we have `X` on shape `(Nx,m)`. So the new pseudo code:

```
Z1 = W1X + b1     # shape of Z1 (noOfHiddenNeurons,m)
A1 = sigmoid(Z1)  # shape of A1 (noOfHiddenNeurons,m)
Z2 = W2A1 + b2    # shape of Z2 is (1,m)
A2 = sigmoid(Z2)  # shape of A2 is (1,m)
```

- If you notice always m is the number of columns.
- In the last example we can call `X` = `A0`. So the previous step can be rewritten as:

```
Z1 = W1A0 + b1    # shape of Z1 (noOfHiddenNeurons,m)
A1 = sigmoid(Z1)  # shape of A1 (noOfHiddenNeurons,m)
Z2 = W2A1 + b2    # shape of Z2 is (1,m)
A2 = sigmoid(Z2)  # shape of A2 is (1,m)
```

Activation functions

- So far we are using sigmoid, but in some cases other functions can be a lot better.
- Sigmoid can lead us to gradient decent problem where the updates are so low.
- Sigmoid activation function range is [0,1] `A = 1 / (1 + np.exp(-z)) # Where z is the input matrix`
- Tanh activation function range is [-1,1] (Shifted version of sigmoid function)
  - In NumPy we can implement Tanh using one of these methods: `A = (np.exp(z) - np.exp(-z)) / (np.exp(z) + np.exp(-z)) # Where z is the input matrix`

    Or `A = np.tanh(z) # Where z is the input matrix`
- 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.
- Sigmoid or Tanh function disadvantage is that if the input is too small or too high, the slope will be near zero which will cause us the gradient decent problem.
- 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 of RELU is that if the input is negative the slope will be so small. 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.`
- In NN you will decide a lot of choices like:
  - No of hidden layers.
  - No of neurons in each hidden layer.
  - Learning rate. (The most important parameter)
  - Activation functions.
  - And others..
- It turns out there are no guide lines for that. You should try all activation functions for example.

Why do you need non-linear activation functions?

- If we removed the activation function from our algorithm that can be called linear activation function.
- Linear activation function will output linear activations
  - Whatever hidden layers you add, the activation will be always linear like logistic regression (So its useless in a lot of complex problems)
- 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.

Derivatives of activation functions

- Derivation of Sigmoid activation function:

```
g(z) = 1 / (1 + np.exp(-z))
g'(z) = (1 / (1 + np.exp(-z))) * (1 - (1 / (1 + np.exp(-z))))
g'(z) = g(z) * (1 - g(z))
```

- Derivation of Tanh activation function:

```
g(z)  = (e^z - e^-z) / (e^z + e^-z)
g'(z) = 1 - np.tanh(z)^2 = 1 - g(z)^2
```

- Derivation of RELU activation function:

```
g(z)  = np.maximum(0,z)
g'(z) = { 0  if z < 0
          1  if z >= 0  }
```

- Derivation of leaky RELU activation function:

```
g(z)  = np.maximum(0.01 * z, z)
g'(z) = { 0.01  if z < 0
          1     if z >= 0   }
```

Gradient descent for Neural Networks

- In this section we will have the full back propagation of the neural network (Just the equations with no explanations).
- Gradient descent algorithm:
  - NN parameters:
    - `n[0] = Nx`
    - `n[1] = NoOfHiddenNeurons`
    - `n[2] = NoOfOutputNeurons = 1`
    - `W1` shape is `(n[1],n[0])`
    - `b1` shape is `(n[1],1)`
    - `W2` shape is `(n[2],n[1])`
    - `b2` shape is `(n[2],1)`
  - Cost function `I = I(W1, b1, W2, b2) = (1/m) * Sum(L(Y,A2))`
  - Then Gradient descent:

```
Repeat:
                Compute predictions (y'[i], i = 0,...m)
                Get derivatives: dW1, db1, dW2, db2
                Update: W1 = W1 - LearningRate * dW1
                                b1 = b1 - LearningRate * db1
                                W2 = W2 - LearningRate * dW2
                                b2 = b2 - LearningRate * db2
```

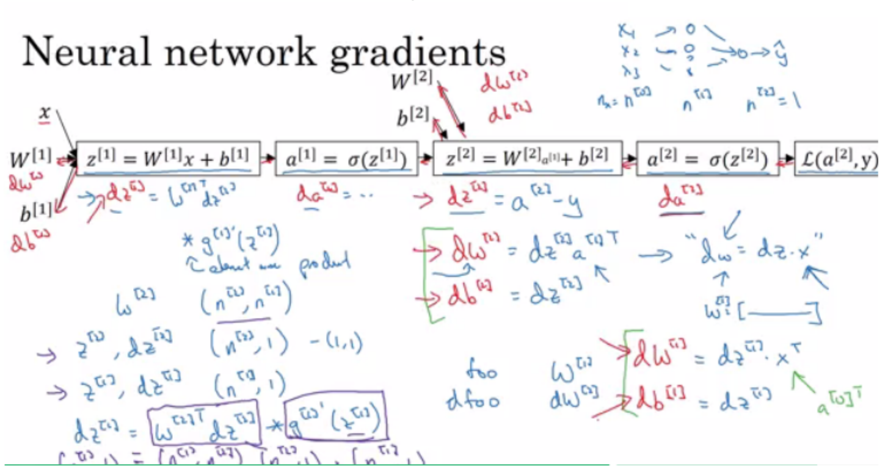
- Forward propagation:

```
Z1 = W1A0 + b1    # A0 is X
A1 = g1(Z1)
Z2 = W2A1 + b2
A2 = Sigmoid(Z2)      # Sigmoid because the output is between 0 and 1
```

- Backpropagation (derivations):

```
dZ2 = A2 - Y      # derivative of cost function we used * derivative of the sigmoid function
dW2 = (dZ2 * A1.T) / m
db2 = Sum(dZ2) / m
dZ1 = (W2.T * dZ2) * g'1(Z1)  # element wise product (*)
dW1 = (dZ1 * A0.T) / m   # A0 = X
db1 = Sum(dZ1) / m
# Hint there are transposes with multiplication because to keep dimensions correct
```

- How we derived the 6 equations of the backpropagation:


Neural network gradients
$z^{[1]} = W^{[1]}x + b^{[1]}$
$a^{[1]} = \sigma(z^{[1]})$
$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$
$a^{[2]} = \sigma(z^{[2]})$
$\mathcal{L}(a^{[2]},y)$
$dz^{[2]} = a^{[2]} - y$
$dW^{[2]} = dz^{[2]} a^{[1]T}$
$db^{[2]} = dz^{[2]}$
$dz^{[1]} = W^{[2]T} dz^{[2]} * g^{[1]'}(z^{[1]})$
$dW^{[1]} = dz^{[1]} x^T$
$db^{[1]} = dz^{[1]}$

Random Initialization

- In logistic regression it wasn't important to initialize the weights randomly, while in NN we have to initialize them randomly.
- If we initialize all the weights with zeros in NN it won't work (initializing bias with zero is OK):
  - 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
- To solve this we initialize the W's with a small random numbers:

```
W1 = np.random.randn((2,2)) * 0.01    # 0.01 to make it small enough
b1 = np.zeros((2,1))                  # its ok to have b as zero, it won't get us to the symmetry breaking
problem
```

- We need small values because in sigmoid (or tanh), for example, if the weight is too large you are more likely to end up even at the very start of training with very large values of Z. Which causes your tanh or your sigmoid activation function to be saturated, thus slowing down learning. If you don't have any sigmoid or tanh activation functions throughout your neural network, this is less of an issue.
- Constant 0.01 is alright for 1 hidden layer networks, but if the NN is deep this number can be changed but it will always be a small number.

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.

Deep L-layer neural network

- Shallow NN is a NN with one or two layers.
- Deep NN is a NN with three or more layers.
- We will use the notation `L` to denote the number of layers in a NN.
- `n[l]` is the number of neurons in a specific layer `l`.
- `n[0]` denotes the number of neurons input layer. `n[L]` denotes the number of neurons in output layer.
- `g[l]` is the activation function.
- `a[l] = g[l](z[l])`
- `w[l]` weights is used for `z[l]`
- `x = a[0]`, `a[l] = y'`
- These were the notation we will use for deep neural network.
- So we have:
  - A vector `n` of shape `(1, NoOfLayers+1)`
  - A vector `g` of shape `(1, NoOfLayers)`
  - A list of different shapes `w` based on the number of neurons on the previous and the current layer.
  - A list of different shapes `b` based on the number of neurons on the current layer.

Forward Propagation in a Deep Network

- Forward propagation general rule for one input:

```
z[l] = W[l]a[l-1] + b[l]
a[l] = g[l](a[l])
```

- Forward propagation general rule for `m` inputs:

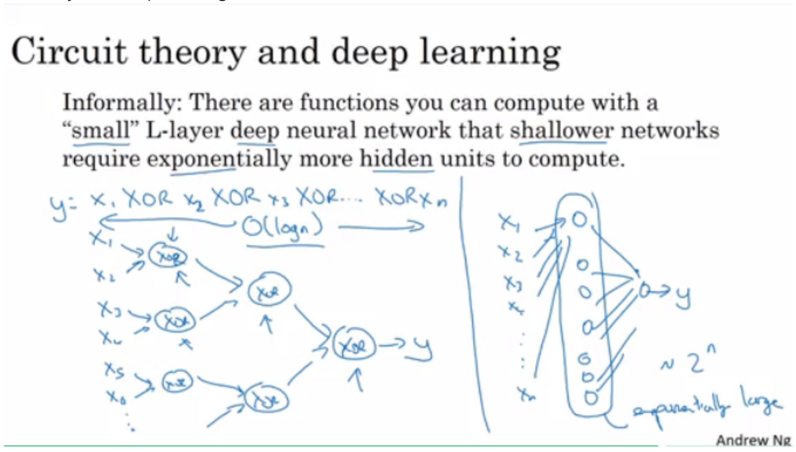
```
Z[l] = W[l]A[l-1] + B[l]
A[l] = g[l](A[l])
```

- We can't compute the whole layers forward propagation without a for loop so its OK to have a for loop here.
- The dimensions of the matrices are so important you need to figure it out.

Getting your matrix dimensions right

- The best way to debug your matrices dimensions is by a pencil and paper.
- Dimension of `W` is `(n[l],n[l-1])` . Can be thought by right to left.
- Dimension of `b` is `(n[l],1)`
- `dw` has the same shape as `W`, while `db` is the same shape as `b`
- Dimension of `Z[l],` `A[l]`, `dZ[l]`, and `dA[l]` is `(n[l],m)`

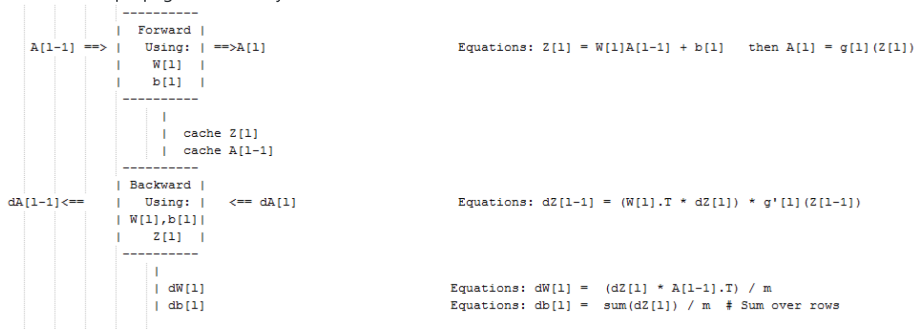
Why deep representations?

- Why deep NN works well, we will discuss this question in this section.
- Deep NN makes relations with data from simpler to complex. In each layer it tries to make a relation with the previous layer. E.g.:
  - a. Face recognition application:
    - Image ==> Edges ==> Face parts ==> Faces ==> desired face
  - b. Audio recognition application:
    - Audio ==> Low level sound features like (sss,bb) ==> Phonemes ==> Words ==> Sentences
- Neural Researchers think that deep neural networks "think" like brains (simple ==> complex)
- Circuit theory and deep learning:
  - 
    Circuit theory and deep learning

    Informally: There are functions you can compute with a "small" L-layer deep neural network that shallower networks require exponentially more hidden units to compute.

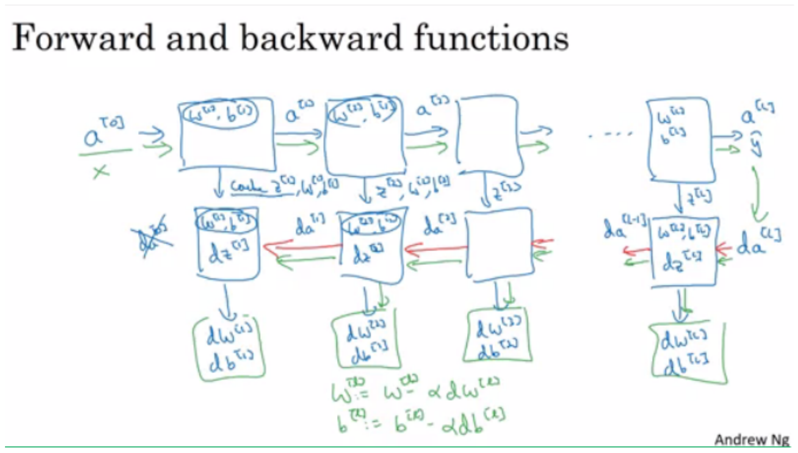
- When starting on an application don't start directly by dozens of hidden layers. Try the simplest solutions (e.g. Logistic Regression), then try the shallow neural network and so on.

Building blocks of deep neural networks

- Forward and back propagation for a layer l:
  - 

```
                ----------
               |  Forward |
  A[l-1] ==>   |   Using: | ==>A[l]            Equations: Z[l] = W[l]A[l-1] + b[l]   then A[l] = g[l](Z[l])
               |    W[l]  |
               |    b[l]  |
                ----------
                    |
                    |  cache Z[l]
                    |  cache A[l-1]
                ----------
               | Backward |
dA[l-1]<==     |   Using: |   <== dA[l]        Equations: dZ[l-1] = (W[l].T * dZ[l]) * g'[l](Z[l-1])
               | W[l],b[l]|
               |    Z[l]  |
                ----------
                   |
                   | dW[l]                    Equations: dW[l] =  (dZ[l] * A[l-1].T) / m
                   | db[l]                    Equations: db[l] =  sum(dZ[l]) / m  # Sum over rows
```

- Deep NN blocks:



Forward and Backward Propagation

- Pseudo code for forward propagation for layer l:

```
Input  A[l-1]
Z[l] = W[l]A[l-1] + b[l]
A[l] = g[l](Z[l])
Output A[l], cache(Z[l])
```

- Pseudo code for back propagation for layer l:

```
Input da[l], Caches
dZ[l] = dA[l] * g'[l](Z[l])
dW[l] = (dZ[l]A[l-1].T) / m
db[l] = sum(dZ[l])/m                # Dont forget axis=1, keepdims=True
dA[l-1] = w[l].T * dZ[l]            # The multiplication here are a dot product.
Output dA[l-1], dW[l], db[l]
```

- If we have used our loss function then:

```
dA[L] = (-(y/a) + ((1-y)/(1-a)))
```

Parameters vs Hyperparameters

- Main parameters of the NN is `W` and `b`
- Hyper parameters (parameters that control the algorithm) are like:
  - Learning rate.
  - Number of iteration.
  - Number of hidden layers `L`.
  - Number of hidden units `n`.
  - Choice of activation functions.
- You have to try values yourself of hyper parameters.
- In the earlier days of DL and ML learning rate was often called a parameter, but it really is (and now everybody call it) a hyperparameter.
- On the next course we will see how to optimize hyperparameters.

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)

Extra: Ian Goodfellow interview

- Ian is one of the world's most visible deep learning researchers.
- Ian is mainly working with generative models. He is the creator of GANs.

Course summary

Here are the course summary as its given on the course link:

> 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.
>
> This is the second course of the Deep Learning Specialization.

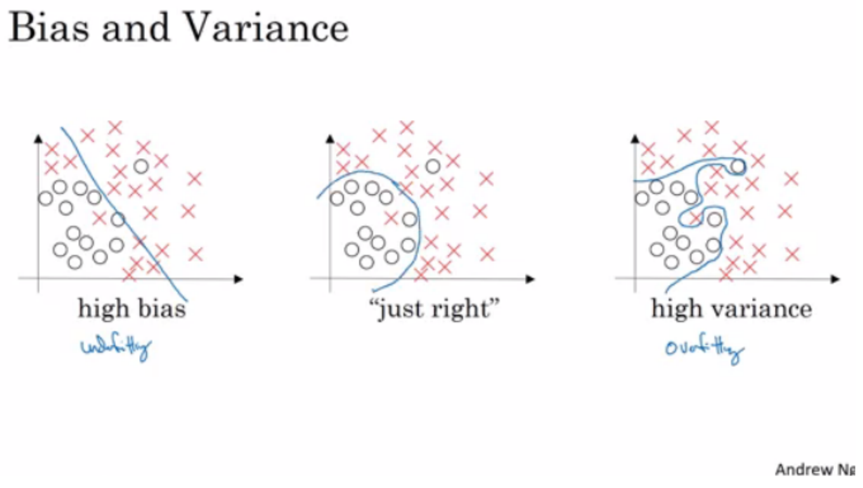
Practical aspects of Deep Learning

Train / Dev / Test sets

- Its impossible to get all your hyperparameters right on a new application from the first time.
- So the idea is you go through the loop: `Idea ==> Code ==> Experiment`.
- You have to go through the loop many times to figure out your hyperparameters.
- Your data will be split into three parts:
    - Training set. (Has to be the largest set)
    - Hold-out cross validation set / Development or "dev" set.
    - Testing set.
- You will try to build a model upon training set then try to optimize hyperparameters on dev set as much as possible. Then after your model is ready you try and evaluate the testing set.
- so the trend on the ratio of splitting the models:
    - If size of the dataset is 100 to 1000000 ==> 60/20/20
    - If size of the dataset is 1000000 to INF ==> 98/1/1 or 99.5/0.25/0.25
- The trend now gives the training data the biggest sets.
- Make sure the dev and test set are coming from the same distribution.
    - For example if cat training pictures is from the web and the dev/test pictures are from users cell phone they will mismatch. It is better to make sure that dev and test set are from the same distribution.
- The dev set rule is to try them on some of the good models you've created.
- Its OK to only have a dev set without a testing set. But a lot of people in this case call the dev set as the test set. A better terminology is to call it a dev set as its used in the development.

Bias / Variance

- Bias / Variance techniques are Easy to learn, but difficult to master.
- So here the explanation of Bias / Variance:
    - If your model is underfitting (logistic regression of non linear data) it has a "high bias"
    - If your model is overfitting then it has a "high variance"
    - Your model will be alright if you balance the Bias / Variance
    - For more:



- Another idea to get the bias / variance if you don't have a 2D plotting mechanism:
  - High variance (overfitting) for example:
    - Training error: 1%
    - Dev error: 11%
  - high Bias (underfitting) for example:
    - Training error: 15%
    - Dev error: 14%
  - high Bias (underfitting) && High variance (overfitting) for example:
    - Training error: 15%
    - Test error: 30%
  - Best:
    - Training error: 0.5%
    - Test error: 1%
  - These Assumptions came from that human has 0% error. If the problem isn't like that you'll need to use human error as baseline.

Basic Recipe for Machine Learning

- If your algorithm has a high bias:
  - Try to make your NN bigger (size of hidden units, number of layers)
  - Try a different model that is suitable for your data.
  - Try to run it longer.
  - Different (advanced) optimization algorithms.
- If your algorithm has a high variance:
  - More data.
  - Try regularization.
  - Try a different model that is suitable for your data.
- You should try the previous two points until you have a low bias and low variance.
- In the older days before deep learning, there was a "Bias/variance tradeoff". But because now you have more options/tools for solving the bias and variance problem its really helpful to use deep learning.
- Training a bigger neural network never hurts.

Regularization

- Adding regularization to NN will help it reduce variance (overfitting)
- L1 matrix norm:
  - `||W|| = Sum(|w[i,j]|) # sum of absolute values of all w`
- L2 matrix norm because of arcane technical math reasons is called Frobenius norm:
  - `||W||^2 = Sum(|w[i,j]|^2) # sum of all w squared`
  - Also can be calculated as `||W||^2 = W.T * W if W is a vector`
- Regularization for logistic regression:
  - The normal cost function that we want to minimize is: `J(w,b) = (1/m) * Sum(L(y(i),y'(i)))`
  - The L2 regularization version: `J(w,b) = (1/m) * Sum(L(y(i),y'(i))) + (lambda/2m) * Sum(|w[i]|^2)`
  - The L1 regularization version: `J(w,b) = (1/m) * Sum(L(y(i),y'(i))) + (lambda/2m) * Sum(|w[i]|)`
  - The L1 regularization version makes a lot of w values become zeros, which makes the model size smaller.
  - L2 regularization is being used much more often.
  - `lambda` here is the regularization parameter (hyperparameter)
- Regularization for NN:

- The normal cost function that we want to minimize is:
  `J(W1,b1...,WL,bL) = (1/m) * Sum(L(y(i),y'(i)))`
- The L2 regularization version:
  `J(w,b) = (1/m) * Sum(L(y(i),y'(i))) + (lambda/2m) * Sum((||W[l]||^2)`
- We stack the matrix as one vector `(mn,1)` and then we apply `sqrt(w1^2 + w2^2.....)`
- To do back propagation (old way):
  `dw[l] = (from back propagation)`
- The new way:
  `dw[l] = (from back propagation) + lambda/m * w[l]`
- So plugging it in weight update step:

  -
    ```
    w[l] = w[l] - learning_rate * dw[l]
         = w[l] - learning_rate * ((from back propagation) + lambda/m * w[l])
         = w[l] - (learning_rate*lambda/m) * w[l] - learning_rate * (from back propagation)
         = (1 - (learning_rate*lambda)/m) * w[l] - learning_rate * (from back propagation)
    ```
- In practice this penalizes large weights and effectively limits the freedom in your model.
- The new term `(1 - (learning_rate*lambda)/m) * w[l]` causes the **weight to decay** in proportion to its size.

Why regularization reduces overfitting?

Here are some intuitions:

- Intuition 1:
  - If `lambda` is too large - a lot of w's will be close to zeros which will make the NN simpler (you can think of it as it would behave closer to logistic regression).
  - If `lambda` is good enough it will just reduce some weights that makes the neural network overfit.
- Intuition 2 (with *tanh* activation function):
  - If `lambda` is too large, w's will be small (close to zero) - will use the linear part of the *tanh* activation function, so we will go from non linear activation to *roughly* linear which would make the NN a *roughly* linear classifier.
  - If `lambda` good enough it will just make some of *tanh* activations *roughly* linear which will prevent overfitting.

Implementation tip: if you implement gradient descent, one of the steps to debug gradient descent is to plot the cost function J as a function of the number of iterations of gradient descent and you want to see that the cost function J decreases **monotonically** after every elevation of gradient descent with regularization. If you plot the old definition of J (no regularization) then you might not see it decrease monotonically.

Dropout Regularization

- In most cases Andrew Ng tells that he uses the L2 regularization.
- The dropout regularization eliminates some neurons/weights on each iteration based on a probability.
- A most common technique to implement dropout is called "Inverted dropout".
- Code for Inverted dropout:

  ```
  keep_prob = 0.8   # 0 <= keep_prob <= 1
  l = 3  # this code is only for layer 3
  # the generated number that are less than 0.8 will be dropped. 80% stay, 20% dropped
  d3 = np.random.rand(a[l].shape[0], a[l].shape[1]) < keep_prob

  a3 = np.multiply(a3,d3)   # keep only the values in d3

  # increase a3 to not reduce the expected value of output
  # (ensures that the expected value of a3 remains the same) - to solve the scaling problem
  a3 = a3 / keep_prob
  ```

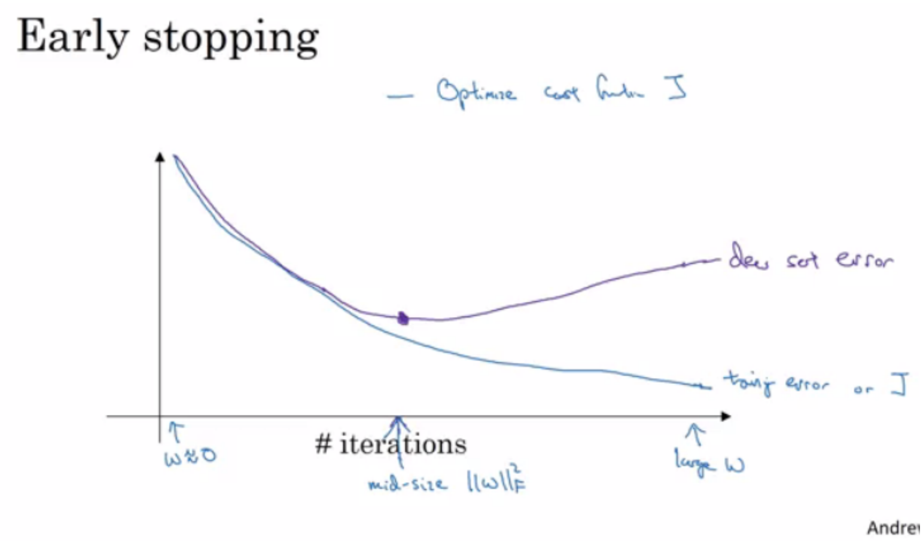
- Vector d[l] is used for forward and back propagation and is the same for them, but it is different for each iteration (pass) or training example.
- At test time we don't use dropout. If you implement dropout at test time - it would add noise to predictions.

Understanding Dropout

- In the previous video, the intuition was that dropout randomly knocks out units in your 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 `keep_prob` 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 `keep_prob` 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 `keep_prob` 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 `keep_prob`s to 1, and then run the code and check that it monotonically decreases J and then turn on the dropouts again.

Other regularization methods

- **Data augmentation**:
  - For example in a computer vision data:
    - You can flip all your pictures horizontally this will give you m more data instances.
    - You could also apply a random position and rotation to an image to get more data.
  - For example in OCR, you can impose random rotations and distortions to digits/letters.
  - New data obtained using this technique isn't as good as the real independent data, but still can be used as a regularization technique.
- **Early stopping**:
  - 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.
  - We will pick the point at which the training set error and dev set error are best (lowest training cost with lowest dev cost).
  - We will take these parameters as the best parameters.
    - 

  - 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).
  - But its advantage is that you don't need to search a hyperparameter like in other regularization approaches (like `lambda` in L2 regularization).
- **Model Ensembles**:
  - Algorithm:
    - Train multiple independent models.
    - At test time average their results.
  - It can get you extra 2% performance.
  - It reduces the generalization error.
  - You can use some snapshots of your NN at the training ensembles them and take the results.

Normalizing inputs

- If you normalize your inputs this will speed up the training process a lot.
- Normalization are going on these steps:
  i. Get the mean of the training set: `mean = (1/m) * sum(x(i))`
  ii. Subtract the mean from each input: `X = X - mean`
    - This makes your inputs centered around 0.

   iii. Get the variance of the training set: `variance = (1/m) * sum(x(i)^2)`
   iv. Normalize the variance. `X /= variance`
- These steps should be applied to training, dev, and testing sets (but using mean and variance of the train set).
- Why normalize?
  - If we don't normalize the inputs our cost function will be deep and its shape will be inconsistent (elongated) then optimizing it will take a long time.
  - But if we normalize it the opposite will occur. The shape of the cost function will be consistent (look more symmetric like circle in 2D example) and we can use a larger learning rate alpha - the optimization will be faster.

Vanishing / Exploding gradients

- The Vanishing / Exploding gradients occurs when your derivatives become very small or very big.
- To understand the problem, suppose that we have a deep neural network with number of layers L, and all the activation functions are **linear** and each `b = 0`
  - Then:

    ```
    Y' = W[L]W[L-1].....W[2]W[1]X
    ```

  - Then, if we have 2 hidden units per layer and x1 = x2 = 1, we result in:

    ```
    if W[l] = [1.5   0]
              [0   1.5] (l != L because of different dimensions in the output layer)
    Y' = W[L] [1.5  0]^(L-1) X = 1.5^L      # which will be very large
              [0  1.5]
    ```

    ```
    if W[l] = [0.5  0]
              [0  0.5]
    Y' = W[L] [0.5  0]^(L-1) X = 0.5^L      # which will be very small
              [0  0.5]
    ```

- The last example explains that the activations (and similarly derivatives) will be decreased/increased exponentially as a function of number of layers.
- So If W > I (Identity matrix) the activation and gradients will explode.
- And If W < I (Identity matrix) the activation and gradients will vanish.
- Recently Microsoft trained 152 layers (ResNet)! which is a really big number. With such a deep neural network, if your activations or gradients increase or decrease exponentially as a function of L, then these values could get really big or really small. And this makes training difficult, especially if your gradients are exponentially smaller than L, then gradient descent will take tiny little steps. It will take a long time for gradient descent to learn anything.
- There is a partial solution that doesn't completely solve this problem but it helps a lot - careful choice of how you initialize the weights (next video).

Weight Initialization for Deep Networks

- A partial solution to the Vanishing / Exploding gradients in NN is better or more careful choice of the random initialization of weights
- In a single neuron (Perceptron model): `Z = w1x1 + w2x2 + ... + wnxn`
  - So if `n_x` is large we want `W`'s to be smaller to not explode the cost.
- So it turns out that we need the variance which equals `1/n_x` to be the range of `W`'s
- So lets say when we initialize `W`'s like this (better to use with `tanh` activation):

  ```
  np.random.rand(shape) * np.sqrt(1/n[l-1])
  ```

  or variation of this (Bengio et al.):

  ```
  np.random.rand(shape) * np.sqrt(2/(n[l-1] + n[l]))
  ```

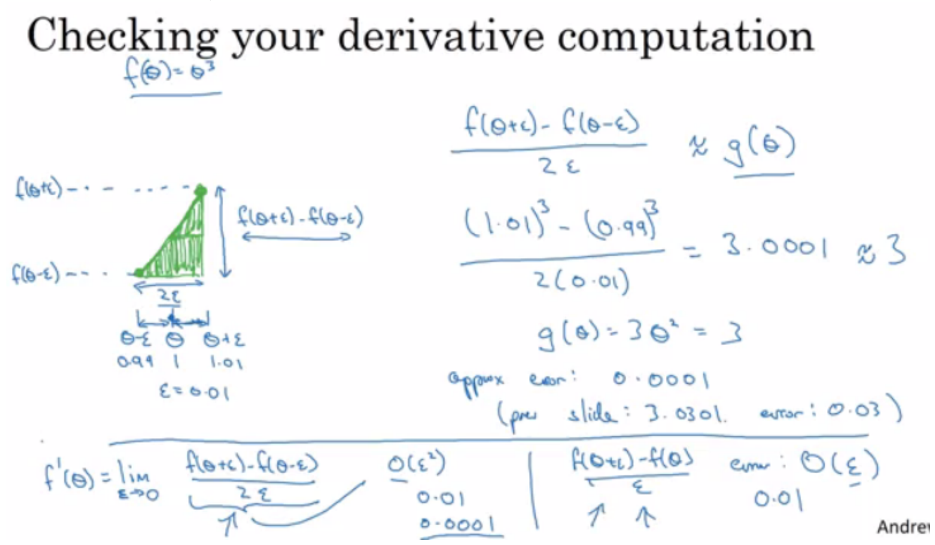
- Setting initialization part inside sqrt to `2/n[l-1]` for `ReLU` is better:

  ```
  np.random.rand(shape) * np.sqrt(2/n[l-1])
  ```

- Number 1 or 2 in the nominator can also be a hyperparameter to tune (but not the first to start with)
- This is one of the best way of partially solution to Vanishing / Exploding gradients (ReLU + Weight Initialization with variance) which will help gradients not to vanish/explode too quickly
- The initialization in this video is called "He Initialization / Xavier Initialization" and has been published in 2015 paper.

Numerical approximation of gradients

- There is an technique called gradient checking which tells you if your implementation of backpropagation is correct.
- There's a numerical way to calculate the derivative:



- 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 of this is very simple.
- Gradient checking:
  - First take `W[1],b[1],...,W[L],b[L]` and reshape into one big vector (`theta`)
  - The cost function will be `J(theta)`
  - Then take `dW[1],db[1],...,dW[L],db[L]` into one big vector (`d_theta`)
  - **Algorithm**:

    ```
    eps = 10^-7   # small number
    for i in len(theta):
      d_theta_approx[i] = (J(theta1,...,theta[i] + eps) -  J(theta1,...,theta[i] - eps)) / 2*eps
    ```

  - Finally we evaluate this formula `(||d_theta_approx - d_theta||) / (||d_theta_approx||+||d_theta||)` (`||` - Euclidean vector norm) and check (with eps = 10^-7):
    - if it is < 10^-7 - great, very likely the backpropagation implementation is correct
    - if around 10^-5 - can be OK, but need to inspect if there are no particularly big values in `d_theta_approx - d_theta` vector
    - if it is >= 10^-3 - bad, probably there is a bug in backpropagation implementation

Gradient checking implementation notes

- Don't use the gradient checking algorithm at training time because it's very slow.
- Use gradient checking only for debugging.
- If algorithm fails grad check, look at components to try to identify the bug.
- Don't forget to add `lamda/(2m) * sum(W[l])` to `J` if you are using L1 or L2 regularization.
- Gradient checking doesn't work with dropout because J is not consistent.
  - You can first turn off dropout (set `keep_prob = 1.0`), run gradient checking and then turn on dropout again.
- 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).

Initialization summary

- 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
- 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 summary

1. L2 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:
Implications of L2-regularization on:

- cost computation:
  - A regularization term is added to the cost
- backpropagation function:
  - There are extra terms in the gradients with respect to weight matrices
- weights:
  - weights end up smaller ("weight decay") - are pushed to smaller values.

2. Dropout

What you should remember about dropout:

- 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 keep_prob to keep the same expected value for the activations. For example, if `keep_prob` 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 keep_prob is other values than 0.5.

Optimization algorithms

Mini-batch gradient descent

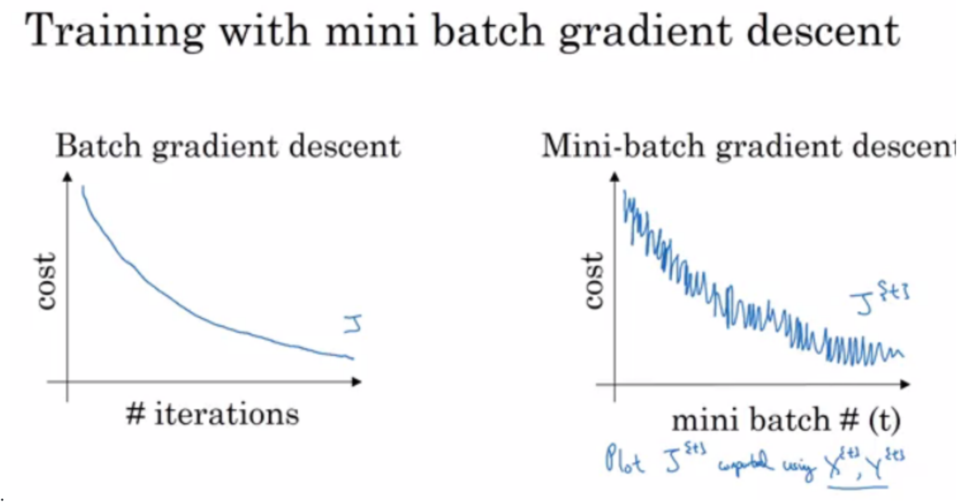
- Training NN with a large data is slow. So to find an optimization algorithm that runs faster is a good idea.
- Suppose we have `m = 50 million`. To train this data it will take a huge processing time for one step.
  - because 50 million won't fit in the memory at once we need other processing to make such a thing.
- It turns out you can make a faster algorithm to make gradient descent process some of your items even before you finish the 50 million items.
- Suppose we have split m to **mini batches** of size 1000.
  - `X{1} = 0 ... 1000`
  - `X{2} = 1001 ... 2000`
  - `...`
  - `X{bs} = ...`
- We similarly split `X` & `Y`.
- So the definition of mini batches ==> `t: X{t}, Y{t}`
- In **Batch gradient descent** we run the gradient descent on the whole dataset.
- While in **Mini-Batch gradient descent** we run the gradient descent on the mini datasets.
- Mini-Batch algorithm pseudo code:

```
for t = 1:No_of_batches                         # this is called an epoch
        AL, caches = forward_prop(X{t}, Y{t})
        cost = compute_cost(AL, Y{t})
        grads = backward_prop(AL, caches)
        update_parameters(grads)
```

- The code inside an epoch should be vectorized.
- Mini-batch gradient descent works much faster in the large datasets.

Understanding mini-batch gradient descent

- In mini-batch algorithm, the cost won't go down with each step as it does in batch algorithm. It could contain some ups and downs but generally it has to go down (unlike the batch gradient descent where cost function descreases on each



iteration).

- Mini-batch size:
  - (`mini batch size = m`) ==> Batch gradient descent
  - (`mini batch size = 1`) ==> Stochastic gradient descent (SGD)
  - (`mini batch size = between 1 and m`) ==> Mini-batch gradient descent
- Batch gradient descent:
  - too long per iteration (epoch)
- Stochastic gradient descent:
  - too noisy regarding cost minimization (can be reduced by using smaller learning rate)
  - won't ever converge (reach the minimum cost)
  - lose speedup from vectorization
- Mini-batch gradient descent:
  1. faster learning:
     - you have the vectorization advantage
     - make progress without waiting to process the entire training set
  2. doesn't always exactly converge (oscelates in a very small region, but you can reduce learning rate)
- Guidelines for choosing mini-batch size:
  1. If small training set (< 2000 examples) - use batch gradient descent.
  2. It has to be a power of 2 (because of the way computer memory is layed out and accessed, sometimes your code runs faster if your mini-batch size is a power of 2): `64, 128, 256, 512, 1024, ...`
  3. Make sure that mini-batch fits in CPU/GPU memory.
- Mini-batch size is a `hyperparameter`.

Exponentially weighted averages

- There are optimization algorithms that are better than **gradient descent**, but you should first learn about Exponentially weighted averages.
- If we have data like the temperature of day through the year it could be like this:

```
t(1) = 40
t(2) = 49
t(3) = 45
...
t(180) = 60
...
```

- This data is small in winter and big in summer. If we plot this data we will find it some noisy.
- Now lets compute the Exponentially weighted averages:

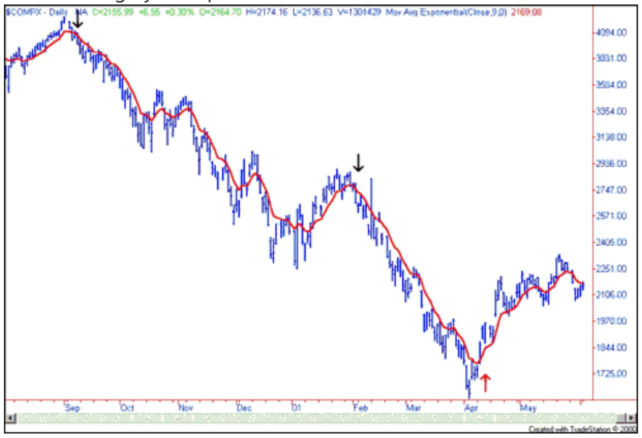
```
V0 = 0
V1 = 0.9 * V0 + 0.1 * t(1) = 4          # 0.9 and 0.1 are hyperparameters
V2 = 0.9 * V1 + 0.1 * t(2) = 8.5
V3 = 0.9 * V2 + 0.1 * t(3) = 12.15
...
```

- General equation

```
V(t) = beta * v(t-1) + (1-beta) * theta(t)
```

- If we plot this it will represent averages over `~ (1 / (1 - beta))` entries:
  - `beta = 0.9` will average last 10 entries
  - `beta = 0.98` will average last 50 entries
  - `beta = 0.5` will average last 2 entries

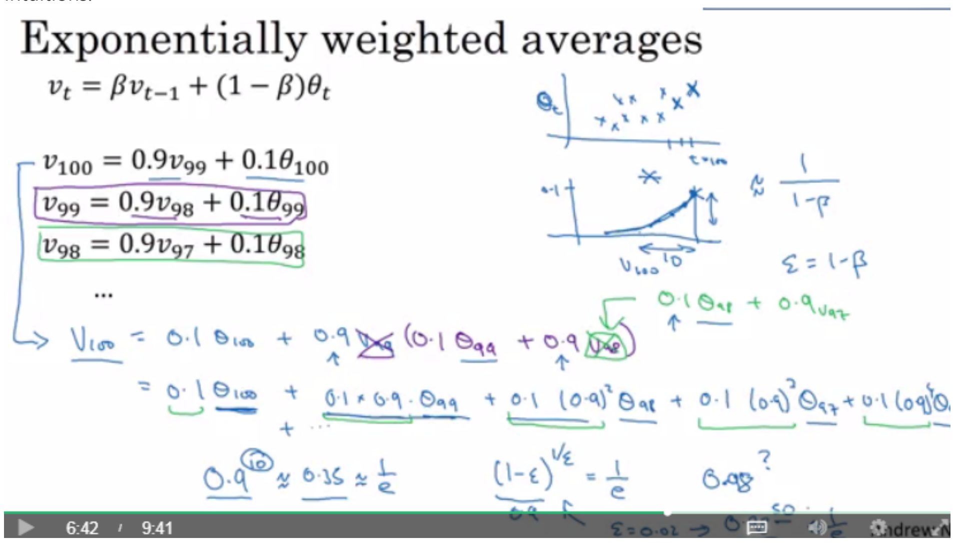
- Best beta average for our case is between 0.9 and 0.98
- Another imagery example:



(taken from investopedia.com)

Understanding exponentially weighted averages

- Intuitions:



Exponentially weighted averages

$v_t = \beta v_{t-1} + (1-\beta)\theta_t$

$v_{100} = 0.9 v_{99} + 0.1\theta_{100}$

$\boxed{v_{99} = 0.9 v_{98} + 0.1\theta_{99}}$

$\boxed{v_{98} = 0.9 v_{97} + 0.1\theta_{98}}$

...

$v_{100} = 0.1\,\theta_{100} + 0.9\,\cancel{v_{99}}\,(0.1\,\theta_{99} + 0.9\,\cancel{v_{98}})$ $\quad 0.1\,\theta_{98} + 0.9\,v_{97}$

$= 0.1\,\theta_{100} + 0.1 \times 0.9 \cdot \theta_{99} + 0.1\,(0.9)^2\theta_{98} + 0.1\,(0.9)^3\theta_{97} + 0.1\,(0.9)^4\theta_{96} + \dots$

$0.9^{10} \approx 0.35 \approx \frac{1}{e}$ $\quad (1-\varepsilon)^{1/\varepsilon} = \frac{1}{e}$ $\quad 0.98?$

$\approx \frac{1}{1-\beta}$ $\quad \varepsilon = 1-\beta$ $\quad \varepsilon = 0.02 \to$

- We can implement this algorithm with more accurate results using a moving window. But the code is more efficient and faster using the exponentially weighted averages algorithm.
- Algorithm is very simple:

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

Bias correction in exponentially weighted averages

- The bias correction helps make the exponentially weighted averages 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 this equation:

```
v(t) = (beta * v(t-1) + (1-beta) * theta(t)) / (1 - beta^t)
```

- As t becomes larger the `(1 - beta^t)` becomes close to `1`

Gradient descent with momentum

- The momentum algorithm almost always works faster than standard gradient descent.
- The simple idea is to calculate the exponentially weighted averages for your gradients and then update your weights with the new values.
- Pseudo code:

```
vdW = 0, vdb = 0
on iteration t:
        # can be mini-batch or batch gradient descent
        compute dw, db on current mini-batch
        vdW = beta * vdW + (1 - beta) * dW
        vdb = beta * vdb + (1 - beta) * db
        W = W - learning_rate * vdW
        b = b - learning_rate * vdb
```

- Momentum helps the cost function to go to the minimum point in a more fast and consistent way.
- `beta` is another `hyperparameter`. `beta = 0.9` is very common and works very well in most cases.
- In practice people don't bother implementing **bias correction**.

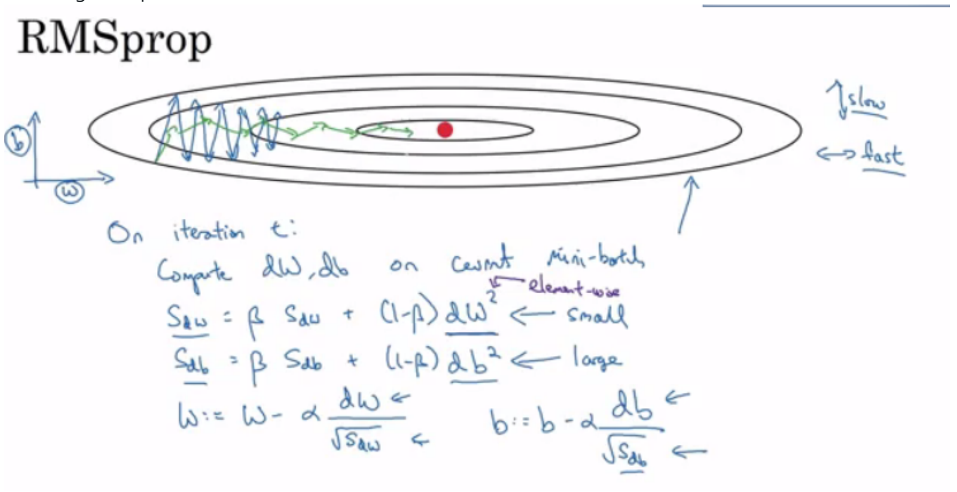
RMSprop

- Stands for **Root mean square prop**.
- This algorithm speeds up the gradient descent.
- Pseudo code:

```
sdW = 0, sdb = 0
on iteration t:
        # can be mini-batch or batch gradient descent
        compute dw, db on current mini-batch

        sdW = (beta * sdW) + (1 - beta) * dW^2  # squaring is element-wise
        sdb = (beta * sdb) + (1 - beta) * db^2  # squaring is element-wise
        W = W - learning_rate * dW / sqrt(sdW)
        b = B - learning_rate * db / sqrt(sdb)
```

- RMSprop will make the cost function move slower on the vertical direction and faster on the horizontal direction in the following example:



RMSprop
b
w
slow
fast
On iteration t:
Compute dW, db on current mini-batch
element-wise
$S_{dW} = \beta S_{dW} + (1-\beta) dW^2 \leftarrow$ small
$S_{db} = \beta S_{db} + (1-\beta) db^2 \leftarrow$ large
$W := W - \alpha \frac{dW}{\sqrt{S_{dW}}}$
$b := b - \alpha \frac{db}{\sqrt{S_{db}}}$

- Ensure that `sdW` is not zero by adding a small value `epsilon` (e.g. `epsilon = 10^-8`) to it:
  `W = W - learning_rate * dW / (sqrt(sdW) + epsilon)`
- With RMSprop you can increase your learning rate.
- Developed by Geoffrey Hinton and firstly introduced on Coursera.org course.

Adam optimization algorithm

- Stands for **Adaptive Moment Estimation**.
- Adam optimization and RMSprop are among the optimization algorithms that worked very well with a lot of NN architectures.
- Adam optimization simply puts RMSprop and momentum together!
- Pseudo code:

```
vdW = 0, vdW = 0
sdW = 0, sdb = 0
on iteration t:
        # can be mini-batch or batch gradient descent
        compute dw, db on current mini-batch

        vdW = (beta1 * vdW) + (1 - beta1) * dW     # momentum
        vdb = (beta1 * vdb) + (1 - beta1) * db     # momentum

        sdW = (beta2 * sdW) + (1 - beta2) * dW^2   # RMSprop
        sdb = (beta2 * sdb) + (1 - beta2) * db^2   # RMSprop

        vdW = vdW / (1 - beta1^t)      # fixing bias
        vdb = vdb / (1 - beta1^t)      # fixing bias

        sdW = sdW / (1 - beta2^t)      # fixing bias
        sdb = sdb / (1 - beta2^t)      # fixing bias
        W = W - learning_rate * vdW / (sqrt(sdW) + epsilon)
        b = B - learning_rate * vdb / (sqrt(sdb) + epsilon)
```

- Hyperparameters for Adam:
  - Learning rate: needed to be tuned.
  - `beta1`: parameter of the momentum - `0.9` is recommended by default.
  - `beta2`: parameter of the RMSprop - `0.999` is recommended by default.
  - `epsilon`: `10^-8` is recommended by default.

Learning rate decay

- Slowly reduce learning rate.
- As mentioned before mini-batch gradient descent won't reach the optimum point (converge). But by making the learning rate decay with iterations it will be much closer to it because the steps (and possible oscillations) near the optimum are smaller.
- One technique equations is `learning_rate = (1 / (1 + decay_rate * epoch_num)) * learning_rate_0`
  - `epoch_num` is over all data (not a single mini-batch).
- Other learning rate decay methods (continuous):
  - `learning_rate = (0.95 ^ epoch_num) * learning_rate_0`
  - `learning_rate = (k / sqrt(epoch_num)) * learning_rate_0`
- Some people perform learning rate decay discretely - repeatedly decrease after some number of epochs.
- Some people are making changes to the learning rate manually.
- `decay_rate` is another `hyperparameter`.
- For Andrew Ng, learning rate decay has less priority.

The problem of local optima

- The normal local optima is not likely to appear in a deep neural network because data is usually high dimensional. For point to be a local optima it has to be a local optima for each of the dimensions which is highly unlikely.
- It's unlikely to get stuck in a bad local optima in high dimensions, it is much more likely to get to the saddle point rather to the local optima, which is not a problem.
- Plateaus can make learning slow:
  - Plateau is a region where the derivative is close to zero for a long time.
  - This is where algorithms like momentum, RMSprop or Adam can help.

Hyperparameter tuning, Batch Normalization and Programming Frameworks

Tuning process

- We need to tune our hyperparameters to get the best out of them.
- Hyperparameters importance are (as for Andrew Ng):
  i. Learning rate.
  ii. Momentum beta.
  iii. Mini-batch size.
  iv. No. of hidden units.
  v. No. of layers.
  vi. Learning rate decay.
  vii. Regularization lambda.
  viii. Activation functions.
  ix. Adam `beta1` & `beta2`.
- Its hard to decide which hyperparameter is the most important in a problem. It depends a lot on your problem.
- One of the ways to tune is to sample a grid with `N` hyperparameter settings and then try all settings combinations on your problem.
- Try random values: don't use a grid.
- You can use `Coarse to fine sampling scheme`:
  - When you find some hyperparameters values that give you a better performance - zoom into a smaller region around these values and sample more densely within this space.
- These methods can be automated.

Using an appropriate scale to pick hyperparameters

- Let's say you have a specific range for a hyperparameter from "a" to "b". It's better to search for the right ones using the logarithmic scale rather then in linear scale:
  - Calculate: `a_log = log(a) # e.g. a = 0.0001 then a_log = -4`

- Calculate: `b_log = log(b) # e.g. b = 1 then b_log = 0`
- Then:

  ```
  r = (a_log - b_log) * np.random.rand() + b_log
  # In the example the range would be from [-4, 0] because rand range [0,1)
  result = 10^r
  ```

  It uniformly samples values in log scale from [a,b].
- If we want to use the last method on exploring on the "momentum beta":
  - Beta best range is from 0.9 to 0.999.
  - You should search for `1 - beta in range 0.001 to 0.1 (1 - 0.9 and 1 - 0.999)` and the use `a = 0.001` and `b = 0.1`. Then:

    ```
    a_log = -3
    b_log = -1
    r = (a_log - b_log) * np.random.rand() + b_log
    beta = 1 - 10^r   # because 1 - beta = 10^r
    ```

Hyperparameters tuning in practice: Pandas vs. Caviar

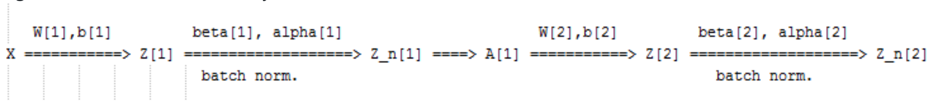
- Intuitions about hyperparameter settings from one application area may or may not transfer to a different one.
- If you don't have much computational resources you can use the "babysitting model":
  - Day 0 you might initialize your parameter as random and then start training.
  - Then you watch your learning curve gradually decrease over the day.
  - And each day you nudge your parameters a little during training.
  - Called panda approach.
- If you have enough computational resources, you can run some models in parallel and at the end of the day(s) you check the results.
  - Called Caviar approach.

Normalizing activations in a network

- In the rise of deep learning, one of the most important ideas has been an algorithm called **batch normalization**, created by two researchers, Sergey Ioffe and Christian Szegedy.
- Batch Normalization speeds up learning.
- Before we normalized input by subtracting the mean and dividing by variance. This helped a lot for the shape of the cost function and for reaching the minimum point faster.
- The question is: *for any hidden layer can we normalize `A[l]` to train `W[l]`, `b[l]` faster?* This is what batch normalization is about.
- There are some debates in the deep learning literature about whether you should normalize values before the activation function `Z[l]` or after applying the activation function `A[l]`. In practice, normalizing `Z[l]` is done much more often and that is what Andrew Ng presents.
- Algorithm:
  - Given `Z[l] = [z(1), ..., z(m)]`, i = 1 to m (for each input)
  - Compute `mean = 1/m * sum(z[i])`
  - Compute `variance = 1/m * sum((z[i] - mean)^2)`
  - Then `Z_norm[i] = (z[i] - mean) / np.sqrt(variance + epsilon)` (add `epsilon` for numerical stability if variance = 0)
    - Forcing the inputs to a distribution with zero mean and variance of 1.
  - Then `Z_tilde[i] = gamma * Z_norm[i] + beta`
    - To make inputs belong to other distribution (with other mean and variance).
    - gamma and beta are learnable parameters of the model.
    - Making the NN learn the distribution of the outputs.
    - *Note:* if `gamma = sqrt(variance + epsilon)` and `beta = mean` then `Z_tilde[i] = z[i]`

Fitting Batch Normalization into a neural network

- Using batch norm in 3 hidden layers NN:



```
  W[1],b[1]          beta[1], alpha[1]                      W[2],b[2]          beta[2], alpha[2]
X ===========> Z[1] ===================> Z_n[1] ====> A[1] ===========> Z[2] ===================> Z_n[2]
                      batch norm.                                                 batch norm.
```

- Our NN parameters will be:
  - `W[1]`, `b[1]`, ..., `W[L]`, `b[L]`, `beta[1]`, `gamma[1]`, ..., `beta[L]`, `gamma[L]`
  - `beta[1]`, `gamma[1]`, ..., `beta[L]`, `gamma[L]` are updated using any optimization algorithms (like GD, RMSprop, Adam)
- If you are using a deep learning framework, you won't have to implement batch norm yourself:
  - Ex. in Tensorflow you can add this line: `tf.nn.batch-normalization()`

- Batch normalization is usually applied with mini-batches.
- If we are using batch normalization parameters `b[1]`, ..., `b[L]` doesn't count because they will be eliminated after mean subtraction step, so:

  ```
  Z[l] = W[l]A[l-1] + b[l] => Z[l] = W[l]A[l-1]
  Z_norm[l] = ...
  Z_tilde[l] = gamma[l] * Z_norm[l] + beta[l]
  ```

  - Taking the mean of a constant `b[l]` will eliminate the `b[l]`
- So if you are using batch normalization, you can remove b[l] or make it always zero.
- So the parameters will be `W[l]`, `beta[l]`, and `alpha[l]`.
- Shapes:
  - `Z[l] - (n[l], m)`
  - `beta[l] - (n[l], m)`
  - `gamma[l] - (n[l], m)`

Why does Batch normalization work?

- The first reason is the same reason as why we normalize X.
- The second reason is that batch normalization reduces the problem of input values changing (shifting).
- Batch normalization does some regularization:
  - Each mini batch is scaled by the mean/variance computed of that mini-batch.
  - 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.
  - This has a slight regularization effect.
  - Using bigger size of the mini-batch you are reducing noise and therefore regularization effect.
  - 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).

Batch normalization at test time

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

Softmax Regression

- In every example we have used so far we were talking about binary classification.
- There are a generalization of logistic regression called Softmax regression that is used for multiclass classification/regression.
- For example if we are classifying by classes `dog`, `cat`, `baby chick` and `none of that`
  - Dog `class = 1`
  - Cat `class = 2`
  - Baby chick `class = 3`
  - None `class = 0`
  - To represent a dog vector `y = [0 1 0 0]`
  - To represent a cat vector `y = [0 0 1 0]`
  - To represent a baby chick vector `y = [0 0 0 1]`
  - To represent a none vector `y = [1 0 0 0]`
- Notations:
  - `C = no. of classes`
  - Range of classes is `(0, ..., C-1)`
  - In output layer `Ny = C`
- Each of C values in the output layer will contain a probability of the example to belong to each of the classes.
- In the last layer we will have to activate the Softmax activation function instead of the sigmoid activation.
- Softmax activation equations:

  ```
  t = e^(Z[L])                      # shape(C, m)
  A[L] = e^(Z[L]) / sum(t)          # shape(C, m), sum(t) - sum of t's for each example (shape (1, m))
  ```

Training a Softmax classifier

- There's an activation which is called hard max, which gets 1 for the maximum value and zeros for the others.
    - If you are using NumPy, its `np.max` over the vertical axis.
- The Softmax name came from softening the values and not harding them like hard max.
- Softmax is a generalization of logistic activation function to `C` classes. If `C = 2` softmax reduces to logistic regression.
- The loss function used with softmax:

```
L(y, y_hat) = - sum(y[j] * log(y_hat[j])) # j = 0 to C-1
```

- The cost function used with softmax:

```
J(w[1], b[1], ...) = - 1 / m * (sum(L(y[i], y_hat[i]))) # i = 0 to m
```

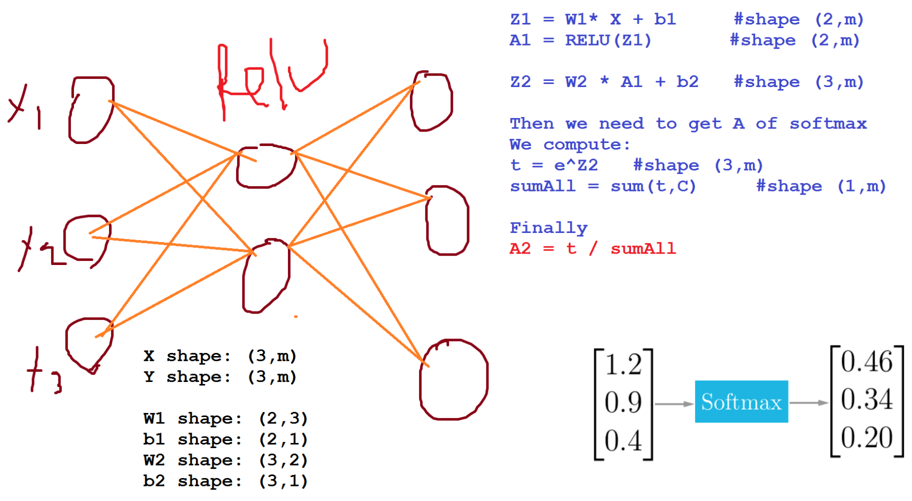
- Back propagation with softmax:

```
dZ[L] = Y_hat - Y
```

- The derivative of softmax is:

```
Y_hat * (1 - Y_hat)
```

- Example:



```
Z1 = W1* X + b1      #shape (2,m)
A1 = RELU(Z1)        #shape (2,m)

Z2 = W2 * A1 + b2    #shape (3,m)

Then we need to get A of softmax
We compute:
t = e^Z2    #shape (3,m)
sumAll = sum(t,C)      #shape (1,m)

Finally
A2 = t / sumAll
```

```
X shape: (3,m)
Y shape: (3,m)

W1 shape: (2,3)
b1 shape: (2,1)
W2 shape: (3,2)
b2 shape: (3,1)
```

Deep learning frameworks

- It's not practical to implement everything from scratch. Our numpy implementations were to know how NN works.
- There are many good deep learning frameworks.
- Deep learning is now in the phase of doing something with the frameworks and not from scratch to keep on going.
- Here are some of the leading deep learning frameworks:
    - Caffe/ Caffe2
    - CNTK
    - DL4j
    - Keras
    - Lasagne
    - mxnet
    - PaddlePaddle
    - TensorFlow
    - Theano
    - Torch/Pytorch
- These frameworks are getting better month by month. Comparison between them can be found here.
- How to choose deep learning framework:
    - Ease of programming (development and deployment)
    - Running speed
    - Truly open (open source with good governance)
- Programming frameworks can not only shorten your coding time but sometimes also perform optimizations that speed up your code.

TensorFlow

- In this section we will learn the basic structure of TensorFlow programs.
- Lets see how to implement a minimization function:
  - Example function: `J(w) = w^2 - 10w + 25`
  - The result should be `w = 5` as the function is `(w-5)^2 = 0`
  - Code v.1:

    ```
    import numpy as np
    import tensorflow as tf


    w = tf.Variable(0, dtype=tf.float32)                 # creating a variable w
    cost = tf.add(tf.add(w**2, tf.multiply(-10.0, w)), 25.0)        # can be written as this - cost = w**2 - 10
    train = tf.train.GradientDescentOptimizer(0.01).minimize(cost)

    init = tf.global_variables_initializer()
    session = tf.Session()
    session.run(init)
    session.run(w)    # Runs the definition of w, if you print this it will print zero
    session.run(train)

    print("W after one iteration:", session.run(w))

    for i in range(1000):
    	session.run(train)

    print("W after 1000 iterations:", session.run(w))
    ```

  - Code v.2 (we feed the inputs to the algorithm through coefficients):

    ```
    import numpy as np
    import tensorflow as tf


    coefficients = np.array([[1.], [-10.], [25.]])

    x = tf.placeholder(tf.float32, [3, 1])
    w = tf.Variable(0, dtype=tf.float32)                 # Creating a variable w
    cost = x[0][0]*w**2 + x[1][0]*w + x[2][0]

    train = tf.train.GradientDescentOptimizer(0.01).minimize(cost)

    init = tf.global_variables_initializer()
    session = tf.Session()
    session.run(init)
    session.run(w)    # Runs the definition of w, if you print this it will print zero
    session.run(train, feed_dict={x: coefficients})

    print("W after one iteration:", session.run(w))

    for i in range(1000):
    	session.run(train, feed_dict={x: coefficients})

    print("W after 1000 iterations:", session.run(w))
    ```

- In TensorFlow you implement only the forward propagation and TensorFlow will do the backpropagation by itself.
- In TensorFlow a placeholder is a variable you can assign a value to later.
- If you are using a mini-batch training you should change the `feed_dict={x: coefficients}` to the current mini-batch data.
- Almost all TensorFlow programs use this:

  ```
  with tf.Session() as session:       # better for cleaning up in case of error/exception
  	session.run(init)
  	session.run(w)
  ```

- In deep learning frameworks there are a lot of things that you can do with one line of code like changing the optimizer. ***Side notes:***
- Writing and running programs in TensorFlow has the following steps:
  i. Create Tensors (variables) that are not yet executed/evaluated.
  ii. Write operations between those Tensors.
  iii. Initialize your Tensors.
  iv. Create a Session.
  v. Run the Session. This will run the operations you'd written above.

- Instead of needing to write code to compute the cost function we know, we can use this line in TensorFlow : `tf.nn.sigmoid_cross_entropy_with_logits(logits = ..., labels = ...)`
- To initialize weights in NN using TensorFlow use:

  ```
  W1 = tf.get_variable("W1", [25,12288], initializer = tf.contrib.layers.xavier_initializer(seed = 1))

  b1 = tf.get_variable("b1", [25,1], initializer = tf.zeros_initializer())
  ```

- For 3-layer NN, it is important to note that the forward propagation stops at `Z3`. The reason is that in TensorFlow the last linear layer output is given as input to the function computing the loss. Therefore, you don't need `A3`!
- To reset the graph use `tf.reset_default_graph()`

Extra Notes

- If you want a good papers in deep learning look at the ICLR proceedings (Or NIPS proceedings) and that will give you a really good view of the field.
- Who is Yuanqing Lin?
  - Head of Baidu research.
  - First one to win ImageNet
  - Works in PaddlePaddle deep learning platform.

These Notes were made by Mahmoud Badry @2017