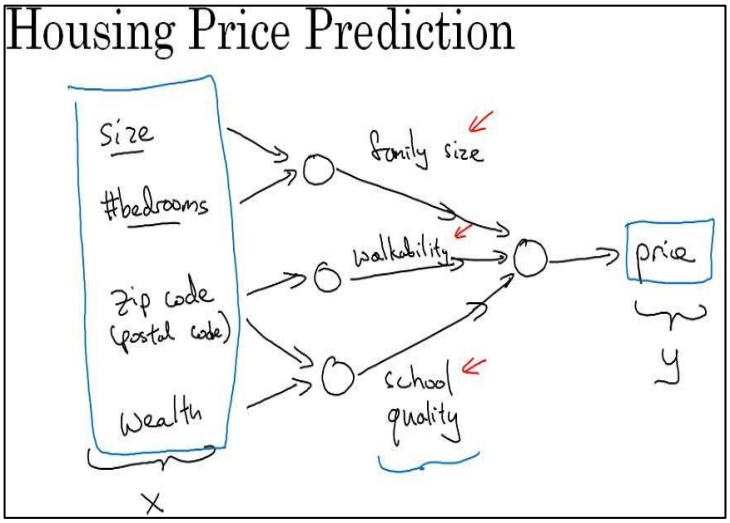
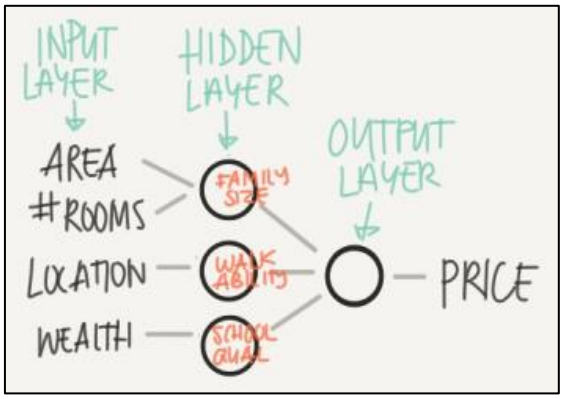
Introduction to Deep Learning

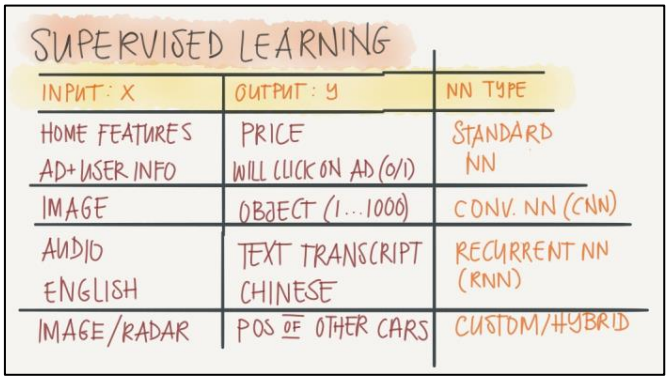
Note : **the illustrations used among the slides of andrew ng, or tess ferandez and badry notes**





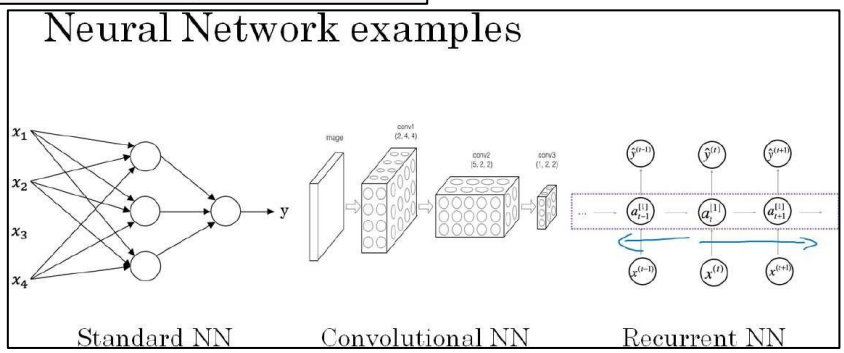
➔ The magic of a neural network is that when you implement it, you need to give it just the inputs " X " and the output " Y " for a number of examples in your training set and all these things in the middle (hidden layers), they will figure out *by itself*.

❖ Supervied learning

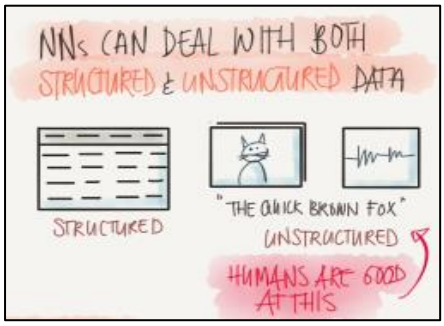


Supervised learning with neural networks

- **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**



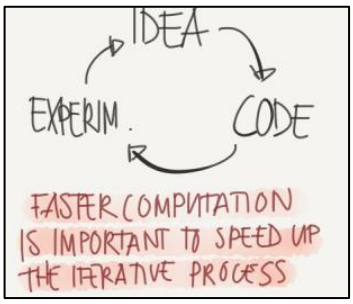
❖ Structured and Unstructured data



- **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.**
- **it's been much harder for computers to make sense of unstructured data compared to structured data. => *unstructured data***
- **each of the features such as size of a house on upper bedrooms or the age of a user has a very well-defined meaning. => *structured data***

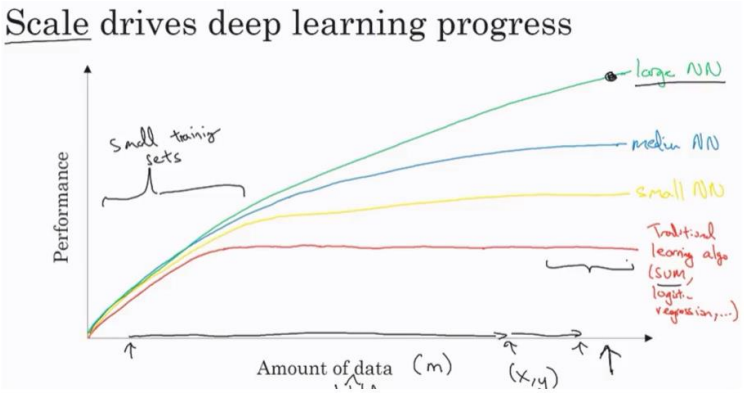
⇨ Computers are now able to **understand unstructured data** in a better way due to deep learning. Techniques in this course will apply both in structured and un-structured data.

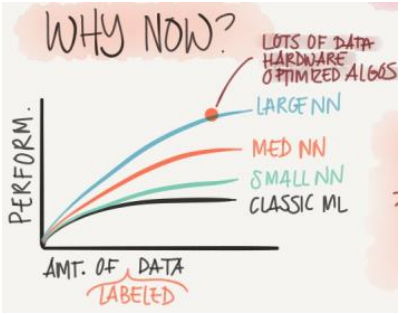
❖ The rise od deep learning



Deep learning is taking off because of availability of lots of digital data which we are continuously creating, lots of research and availability of better hardware, GPU's, quasi networks for computation and advancements in deep learning algorithms. this all helps in implementing idea quickly and in training bigger network.

❖ Why is deep learning taking off

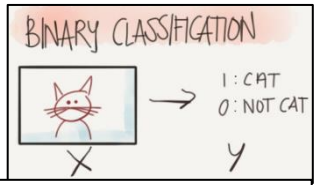


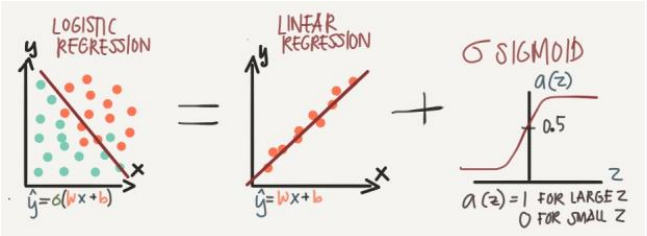


- ✓ Data:
  - For small data NN can perform as Linear regression or SVM (Support vector machine)
  - For big data a small NN is better that SVM
  - For big data a big NN is better that a medium NN is better that small NN.
  - Hopefully we have a lot of data because the world is using the computer a little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

❖ Logistic Regression as NN

⇨ Logistic regression is an algorithm for binary classification. Example of binary classf :



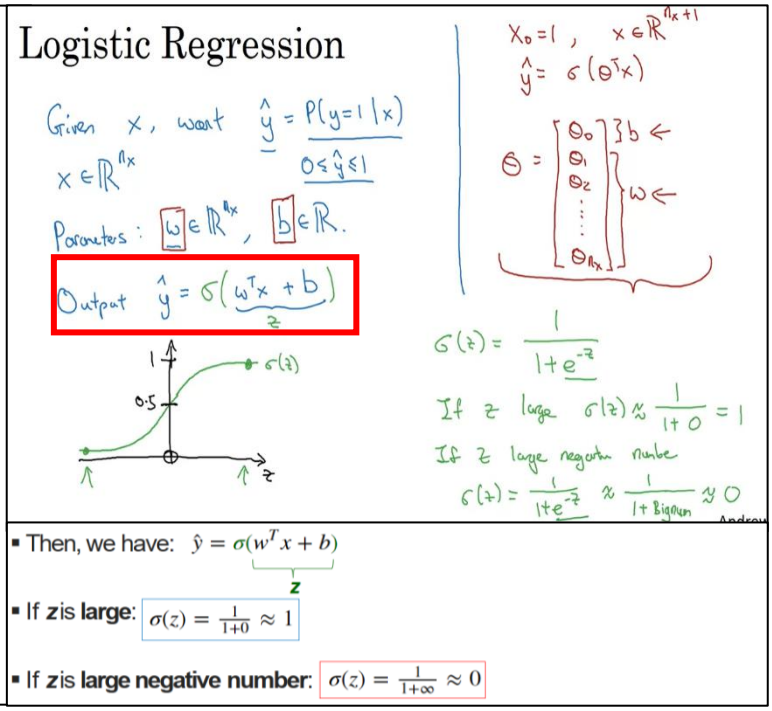


Some notations (we will see the details later)

- ✓ 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))

❖ 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
- 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

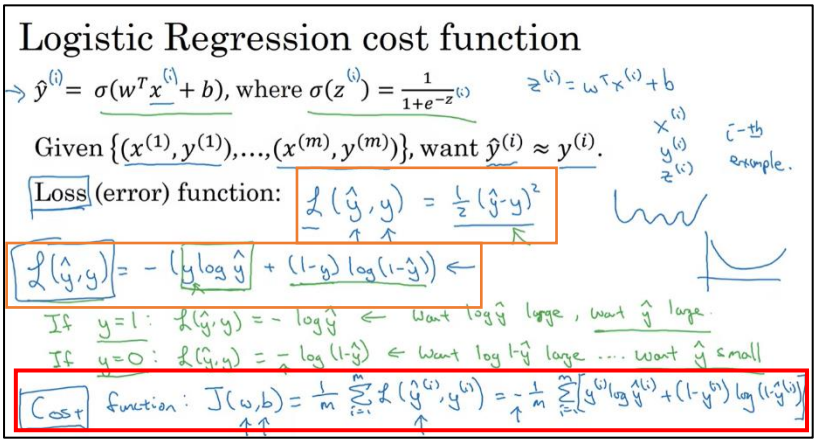


- Then, we have: $\hat{y} = \sigma(w^T x + b)$ (z)
- If **z is large**: $\sigma(z) = \frac{1}{1+0} \approx 1$
- If **z is large negative number**: $\sigma(z) = \frac{1}{1+\infty} \approx 0$

❖ Logistic Regression : Cost function

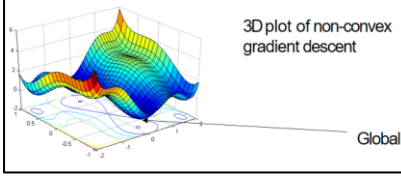
- First loss function would be the square root error: **`L(y',y) = 1/2 (y' - y)^2`**
  - But we won't use this notation because it leads us to optimization problem which is non convex, means it contains local optimum points.
- This is the function that we will use: **`L(y',y) = - (y*log(y') + (1-y)*log(1-y'))`**
- 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

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



▪ Squared Error: $L(\hat{y}, y) = \frac{1}{2}(\hat{y}, y)^2$

- We don't use this one because it makes gradient descent not work well so in logistic regression optimization problem becomes non-convex.



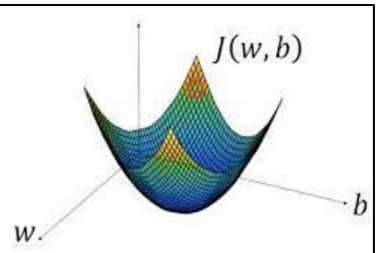
3D plot of non-convex gradient descent

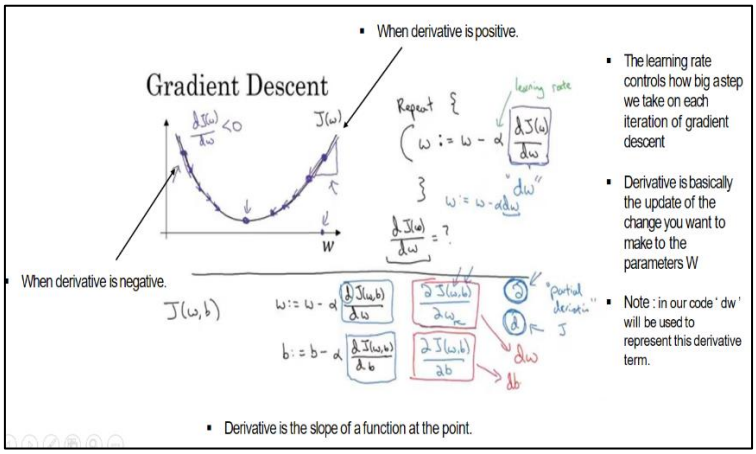
Global minimum

❖ Logistic Regression : Gradient Descent

- We want to predict `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.
- 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)

- Gradient Descent is a process of trying to minimize the error function.
- Gradient descent algorithm is been used to train and also learn the parameters W on your training set.
- In order to learn the set of parameters W and Bit seems natural that we want to find W and Bthat make the cost function J(W, b) as small.

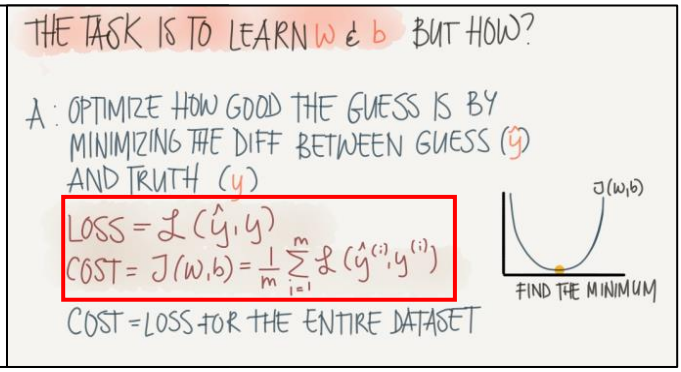




These equations are equivalent.

$$w = w - \alpha * \frac{dJ(w,b)}{dw} \qquad w = w - \alpha * \frac{\partial J(w,b)}{\partial w}$$

$$b = b - \alpha * \frac{dJ(w,b)}{db} \qquad b = b - \alpha * \frac{\partial J(w,b)}{\partial b}$$

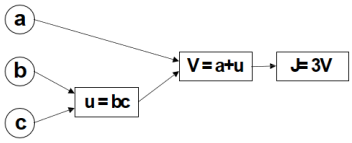


❖ Computing graph

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

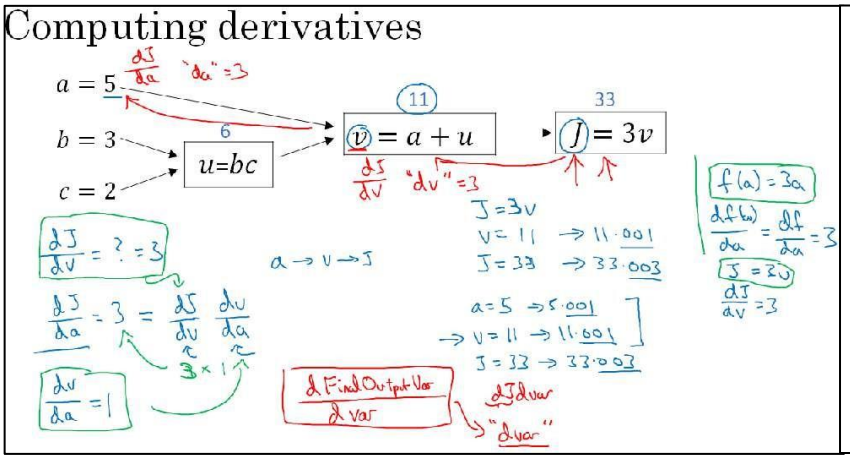
- Let's say that we're trying to compute a **function J**, which is a function of **three variables a, b,** and **c** and let's say that **function J is 3(a+bc).**
- Computing this function actually has three distinct steps:
  1. Compute **bc** and store it in the variable **u**, so **u=bc**
  2. Compute **V=a u,**
  3. Output **J** is **3V**.

J(a,b,c)=3(a+bc)
u=bc
V=a+u
J=3V



- The computations of a neural network are organized in terms of
  - a **forward pass** or a **forward propagation step**, in which we compute the output of the neural network,
  - followed by a **backward pass** or **back propagation step**, which we use to compute gradients or compute derivatives.
- The **computation graph** explains why it is organized this way.
- In order to illustrate the computation graph, let's use a simpler example than logistic regression or a full blown neural network.

❖ Derivatives with aComputing graph

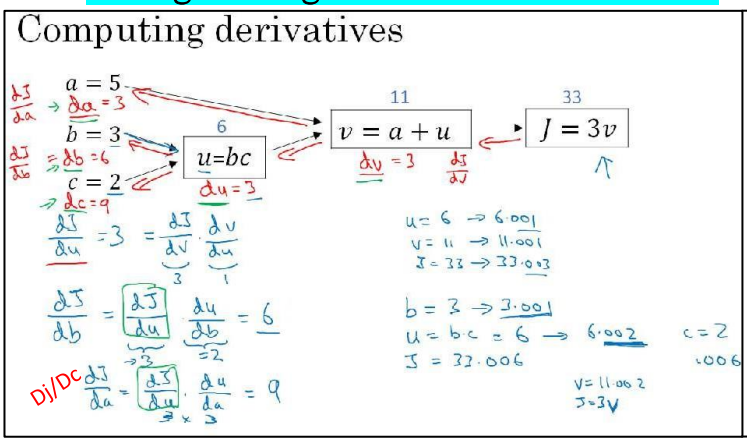


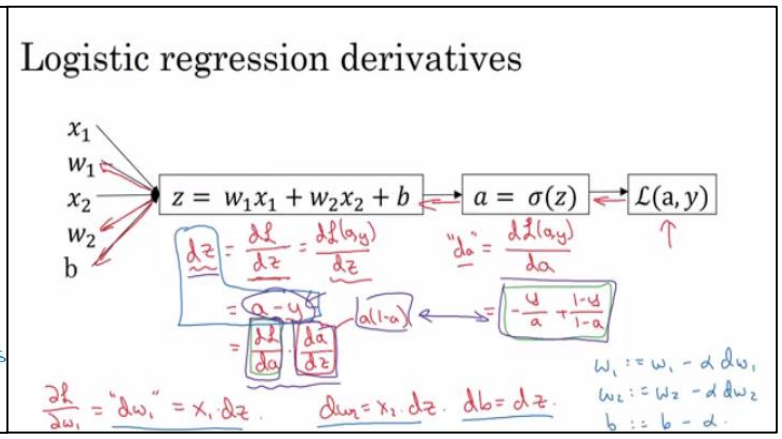
- Calculus chain rule says: If `x -> y -> z` (x effect y and y effects z) Then :

```
d(z)/d(x) = d(z)/d(y) * d(y)/d(x)
```

- 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





Logistic Regression Recap

- **Z**- Logistic Regression Formula: $z = w^T x + b$
- **Y**predictions: $\hat{y} = a = \sigma(z)$
- Loss Function where **a** is the output of logistic regression and **y** is the ground truth label: $\mathcal{L}(a, y) = -(y \log(a) + (1 - y) \log(1 - a))$

How do we compute the derivatives.

- Compute the derivative of $\frac{d\mathcal{L}(a,y)}{da}$ and in code we would denote this as $da = \frac{d\mathcal{L}(a,y)}{da}$

  Note when writing code I assigned $\frac{d\mathcal{L}(a,y)}{da}$ to "**da**" $da = -\frac{y}{a} + \frac{1-y}{1-a}$

- Having computed "**da**", we can go backwards to compute "**dz**"

$$dz = \frac{d\mathcal{L}(a,y)}{dz}$$

$$dz = \frac{d\mathcal{L}(a,y)}{da} * \frac{da}{dz}$$

$$dz = a - y$$

Since we know that :

$$\frac{da}{dz} = a(1-a)$$

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

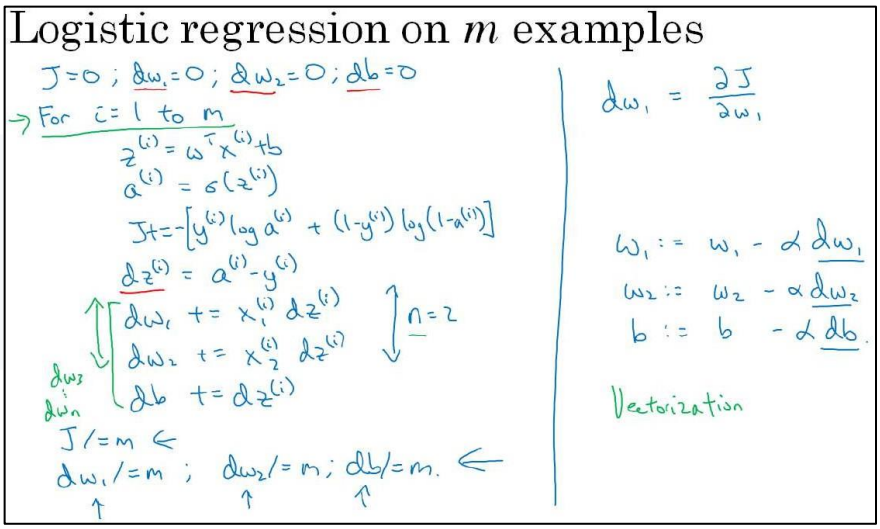
- The final step in back propagation is to go back to compute how much we need to change **w** and **b**

$$dw1 = \frac{d\mathcal{L}(a,y)}{dw1} = x1 * dz \qquad dw2 = \frac{d\mathcal{L}(a,y)}{dw2} = x2 * dz \qquad db = \frac{d\mathcal{L}(a,y)}{db} = dz$$

- so if you want to do gradient descent with respect to just this one example, we would do the following by performing this updates.

$$w1 = w1 - \alpha * dw1 \qquad w2 = w2 - \alpha * dw2 \qquad b = b - \alpha * db$$

❖ Gradient Descent 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

$J = 0; dw_1 = 0; dw_2 = 0; db = 0$

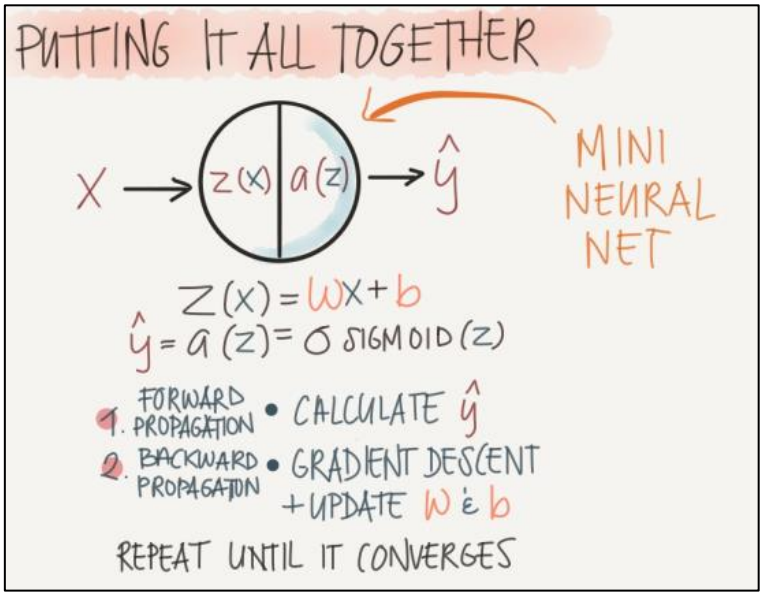
For i = 1 to m

$$Z^{(i)} = w^T x^{(i)} + b$$
$$a^{(i)} = \sigma(Z^{(i)})$$
$$J+ = -[y^{(i)} log a^{(i)} + (1 - y^{(i)}) log(1 - a^{(i)}))]$$
$$dz^{(i)} = a^{(i)} - y^{(i)}$$
$$dw_1+ = x_1^{(i)} dz^{(i)}$$
$$dw_2+ = x_2^{(i)} dz^{(i)}$$
$$db+ = dz^{(i)}$$

n

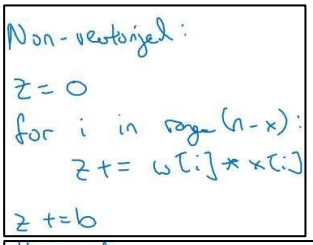
- Then from right to left we will calculate derivations compared to the result:

```
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)
```



❖ Vectorization

- **Vectorization** is a style of computer programming where operations are **applied to whole arrays instead of individual elements**
- Basically the art of **getting rid of explicit for loops** in your code.
- Deep learning algorithms tend to shine when training on large datasets, so it's important that your code runs quickly. Otherwise, your code might take a long time to get your result.
- Vectorization can significantly **speed up your code**.
- The rule of thumb to remember is whenever **possible avoid using explicit for loops in your code.**
- In the next slides, I would be doing a comparison in time using a vectorized and non-vectorized implementation to compute the dot products of two arrays.

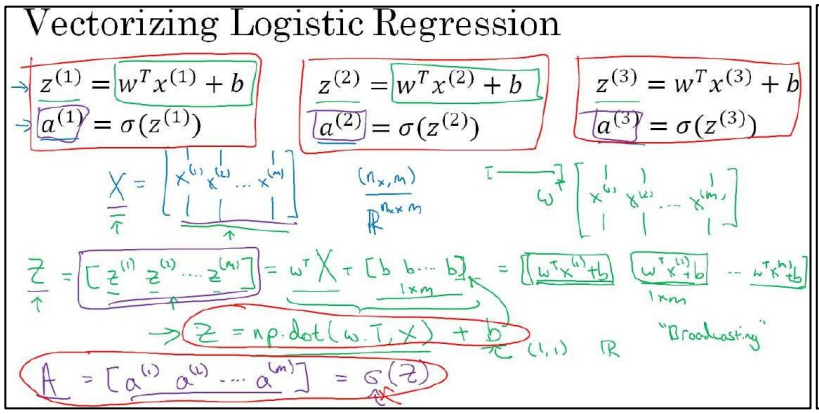


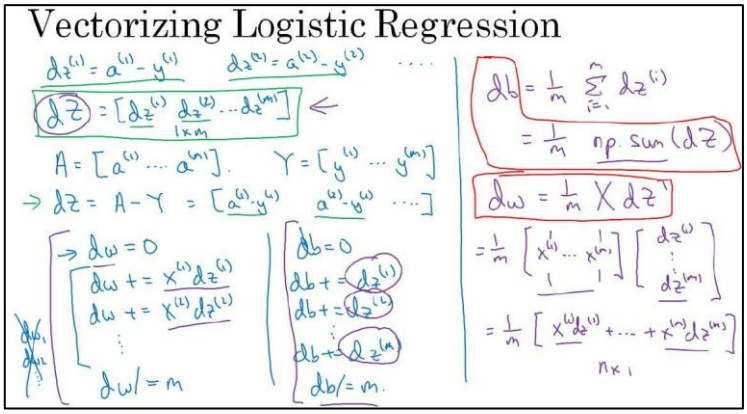
- We will implement Logistic Regression using one for loop then without any for loop.
- As an input we have a matrix `X` and its `[Nx, m]` and a matrix `Y` and its `[Ny, m]`.
- We will then compute at instance `[z1,z2...zm] = W' * X + [b,b,...b]`. This can be written in python as:

```
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)
```





- Now we will vectorize all. Starting with vectorization of db and dw.

$$db = \frac{1}{m} \sum_{i=1}^{m} dz^{(i)} \qquad dw = \frac{1}{m} X dZ^T = \frac{1}{m} [x^{(1)} dz^{(1)} + \dots + x^{(m)} dz^{(m)}]$$

- In python:

$$db = \frac{1}{m} np.sum(dZ) \qquad dw = \frac{1}{m} X dZ^T$$

- Let's pull it all together to actually implement logistic regression.

$$\begin{aligned} Z &= np.dot(w.T, X) + b \\ A &= \alpha(Z) \\ dZ &= A - Y \\ dw &= \frac{1}{m} X dZ^T \\ db &= \frac{1}{m} np.sum(dZ) \\ w &:= w - \alpha dw \\ b &:= b - \alpha db \end{aligned}$$

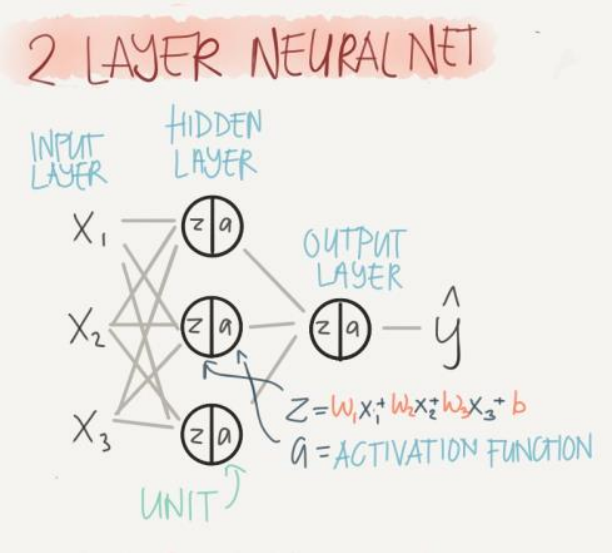
❖ Notes on python and Nupmy

Python/numpy vectors

```
a = np.random.randn(5)
    a.shape = (5,)
    "rank 1 array"
```
Don't use

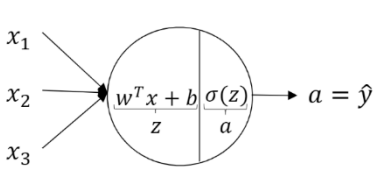
```
a = np.random.randn(5,1) → a.shape = (5,1)   column vector ✓
a = np.random.randn(1,5) → a.shape = (1,5)   row vector ✓
assert(a.shape == (5,1)) ←
    a = a.reshape((5,1))
```

❖ Neural Network representation



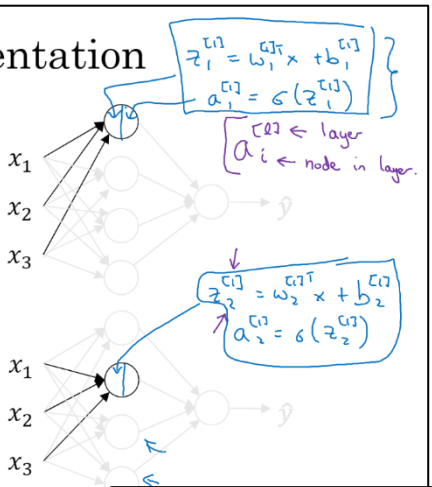
- We will define the neural networks that has one hidden layer.
- NN contains of input layers, hidden layers, output layers.
- Hidden layer means we cant see that layers in the training set.
- $a_0 = x$ (the input layer)
- a_1 will represent the activation of the hidden neurons.
- a_2 will represent the output layer.
- We are talking about 2 layers NN. The input layer isn't counted.

Neural Network Representation

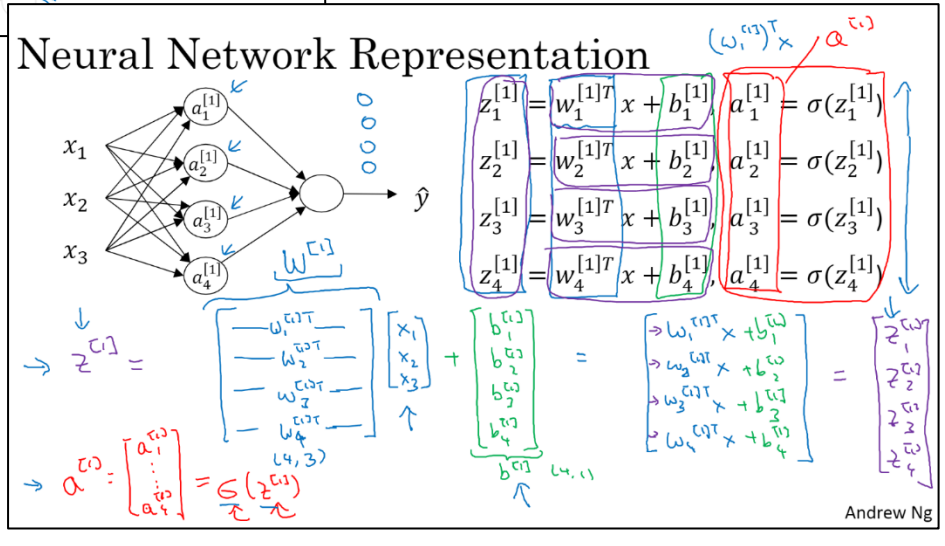


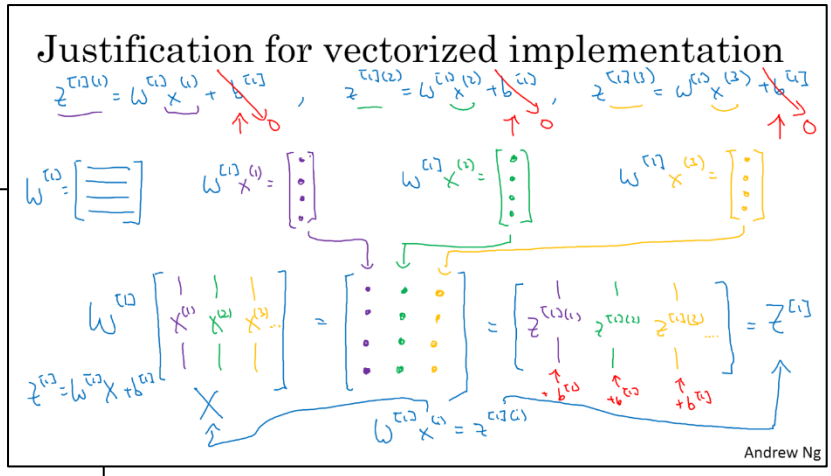
$z = w^Tx + b$

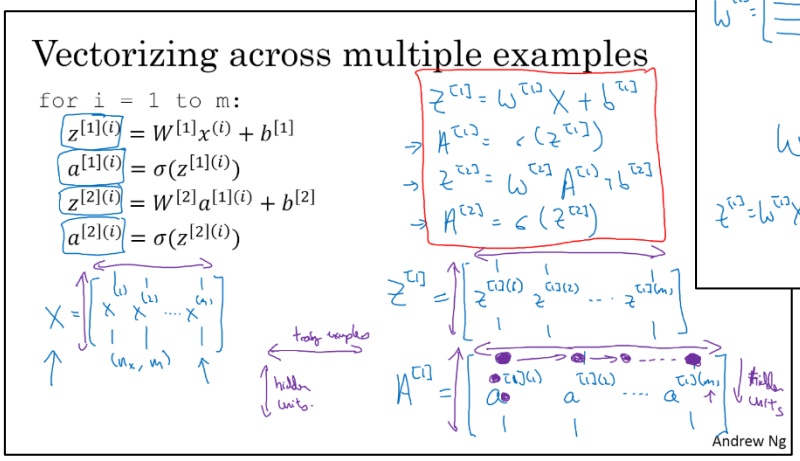
$a = \sigma(z)$

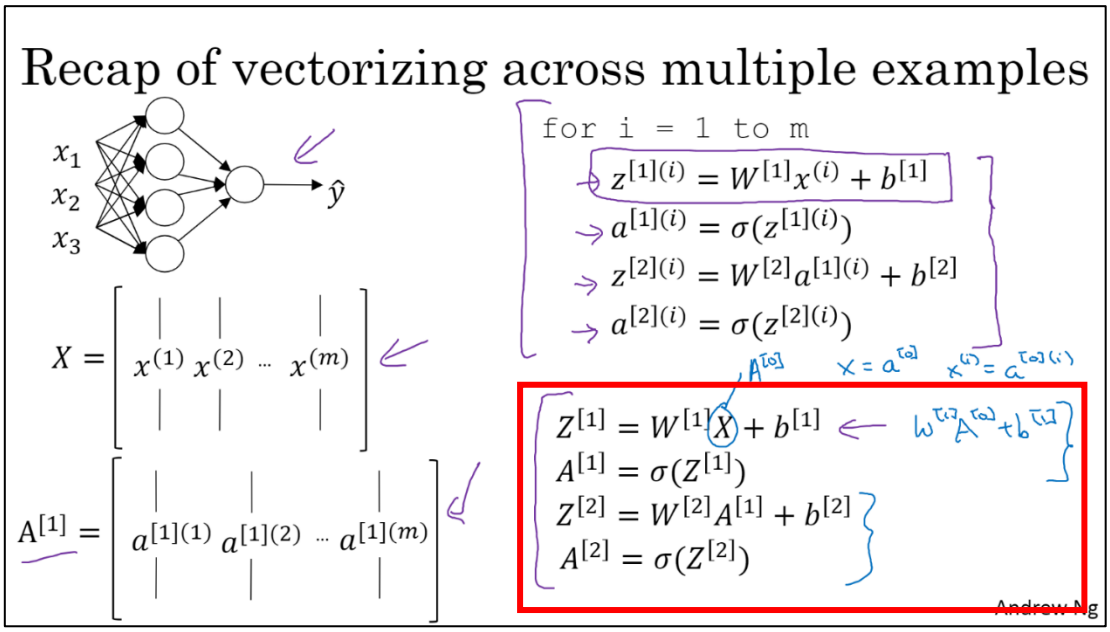


Neural Network Representation

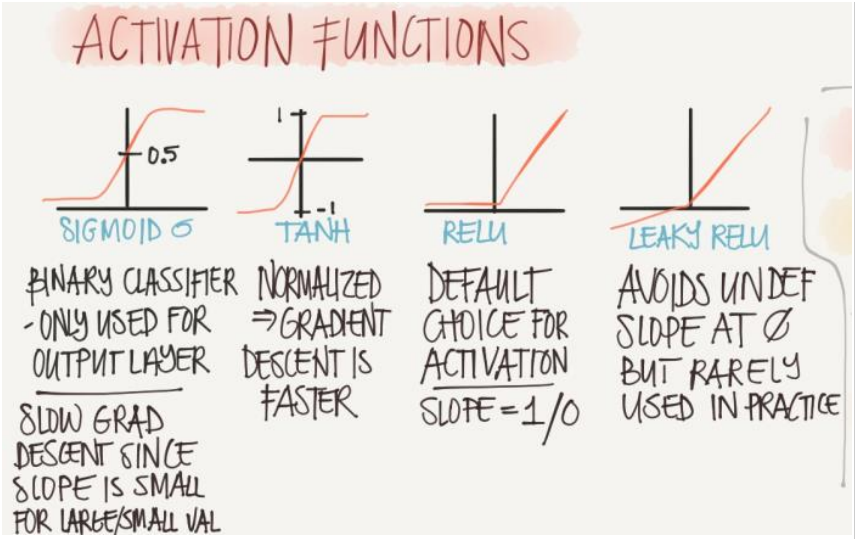








❖ Activation funtions



- 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)

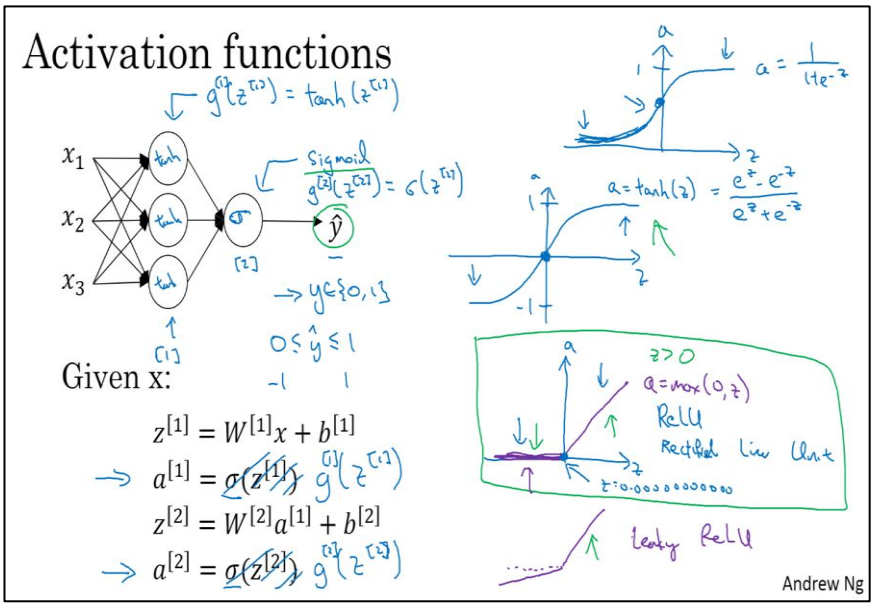
⇨ In NumPy we can implement Tanh using one of these methods:

⇨
```
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.



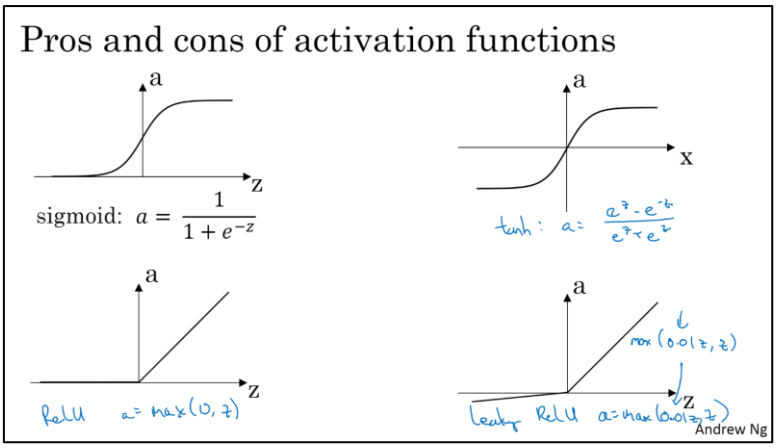
- One of the popular activation functions that solved the slow gradient decent is the RELU function. `RELU = max(0,z)`

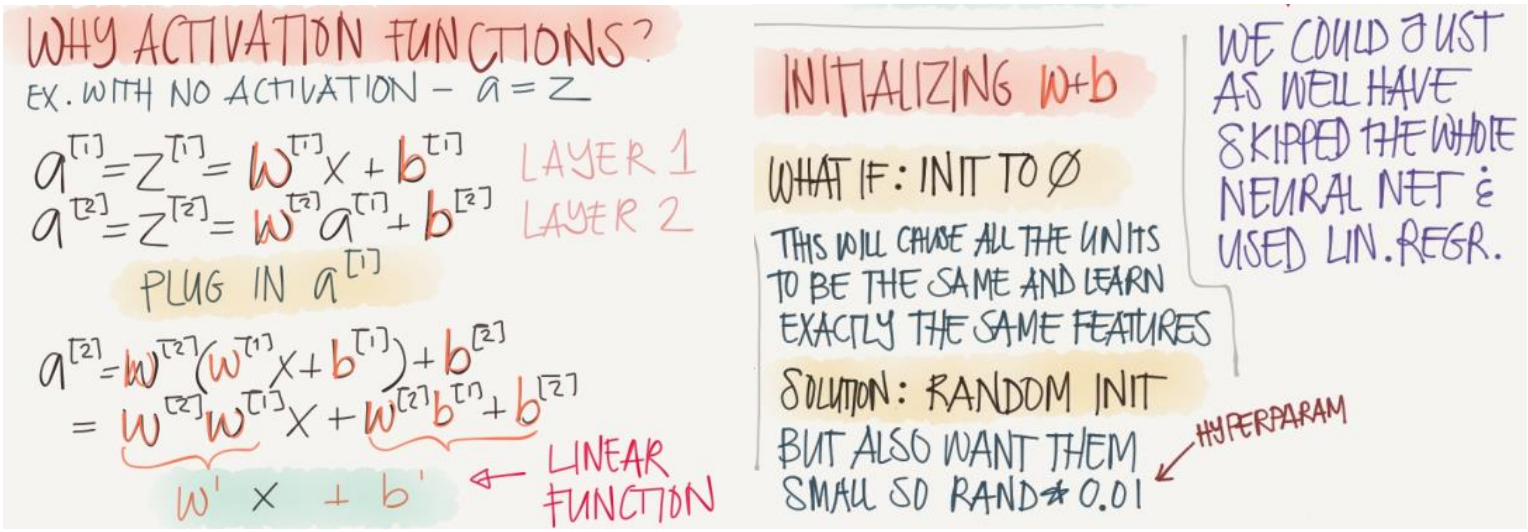
```
# so if z is negative the slope is 0 and if z is positive the slope remains linear.
```

- 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)`

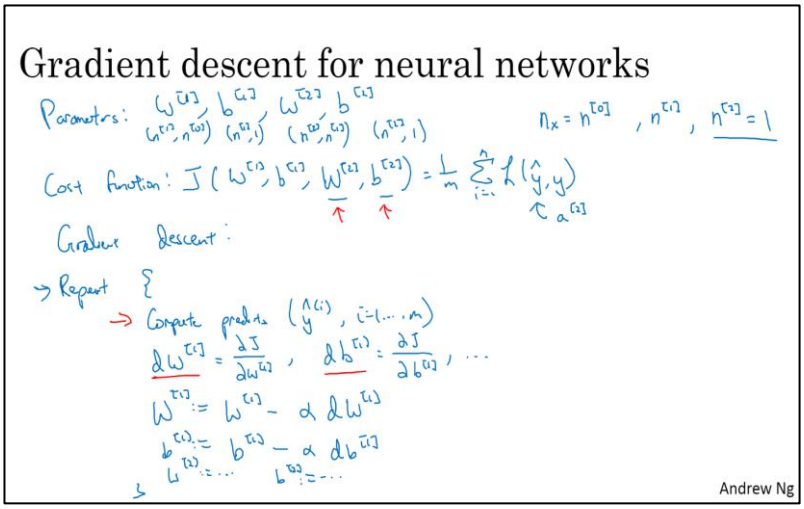
```
#the 0.01 can be a parameter for your algorithm.
```

- 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.





❖ Gradient descent for NNs



- 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:

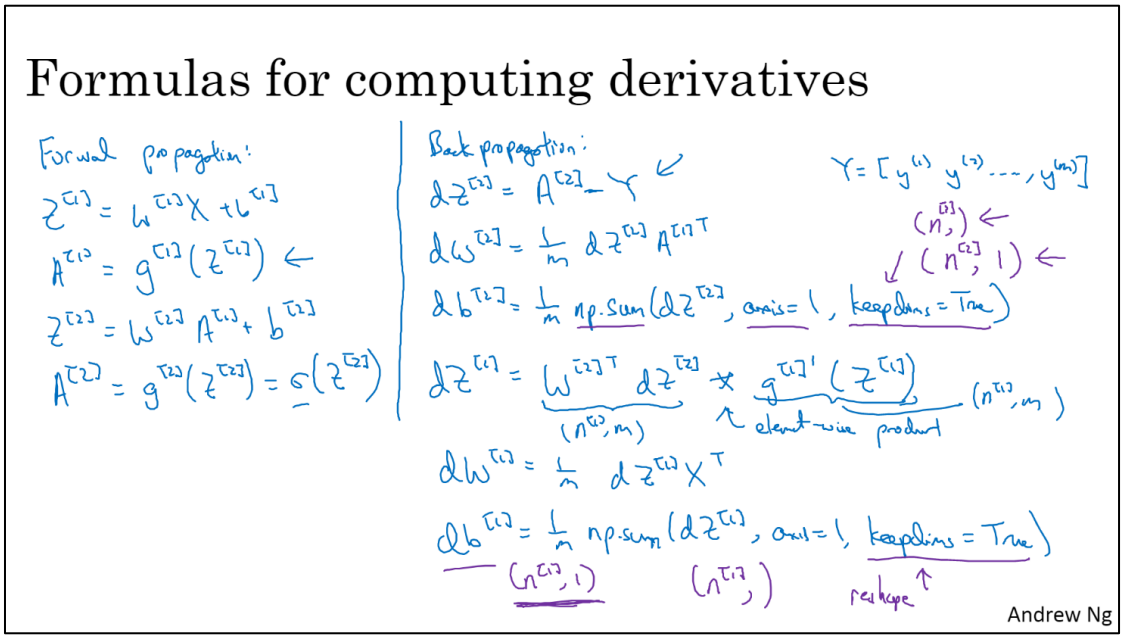
```
Repeat:
        Compute predictions (y'[i], i = 0,...m)
        Get derivatives: dW1, db1, dW2, db2
        Update: W1 = W1 - LearningRate * dW1
                b1 = b1 - LearningRate * db1
                W2 = W2 - LearningRate * dW2
                b2 = b2 - LearningRate * db2
```

- 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
```



❖ Random Initialization

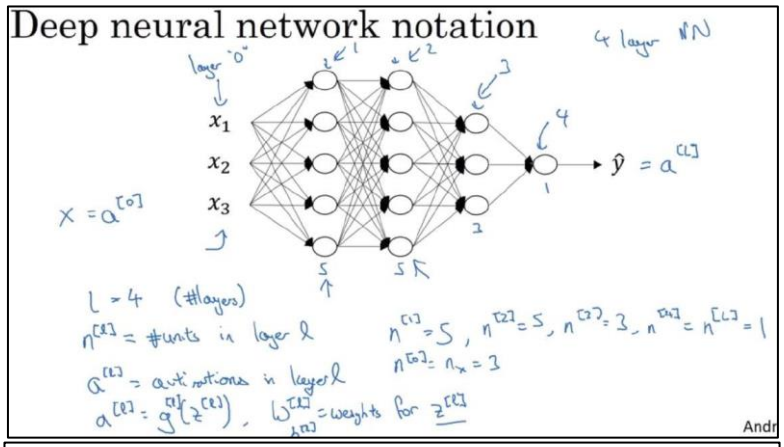
- In logistic regression it wasn't important to initialize the weights randomly, while in NN we have to initialize them randomly.
- If we initialize all the weights with zeros in NN it won't work (initializing bias with zero is OK):
  - all hidden units will be completely identical (symmetric) - compute exactly the same function
  - on each gradient descent iteration all the hidden units will always update the same

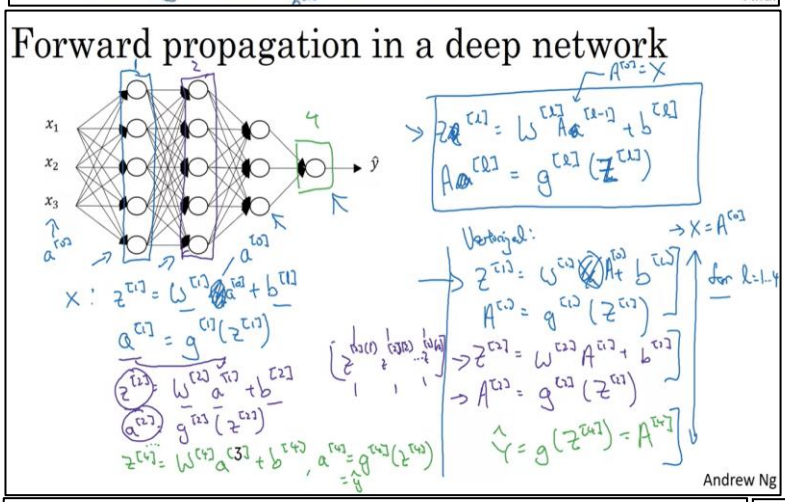
- To solve this we initialize the W's with a small random numbers:

```
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 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:

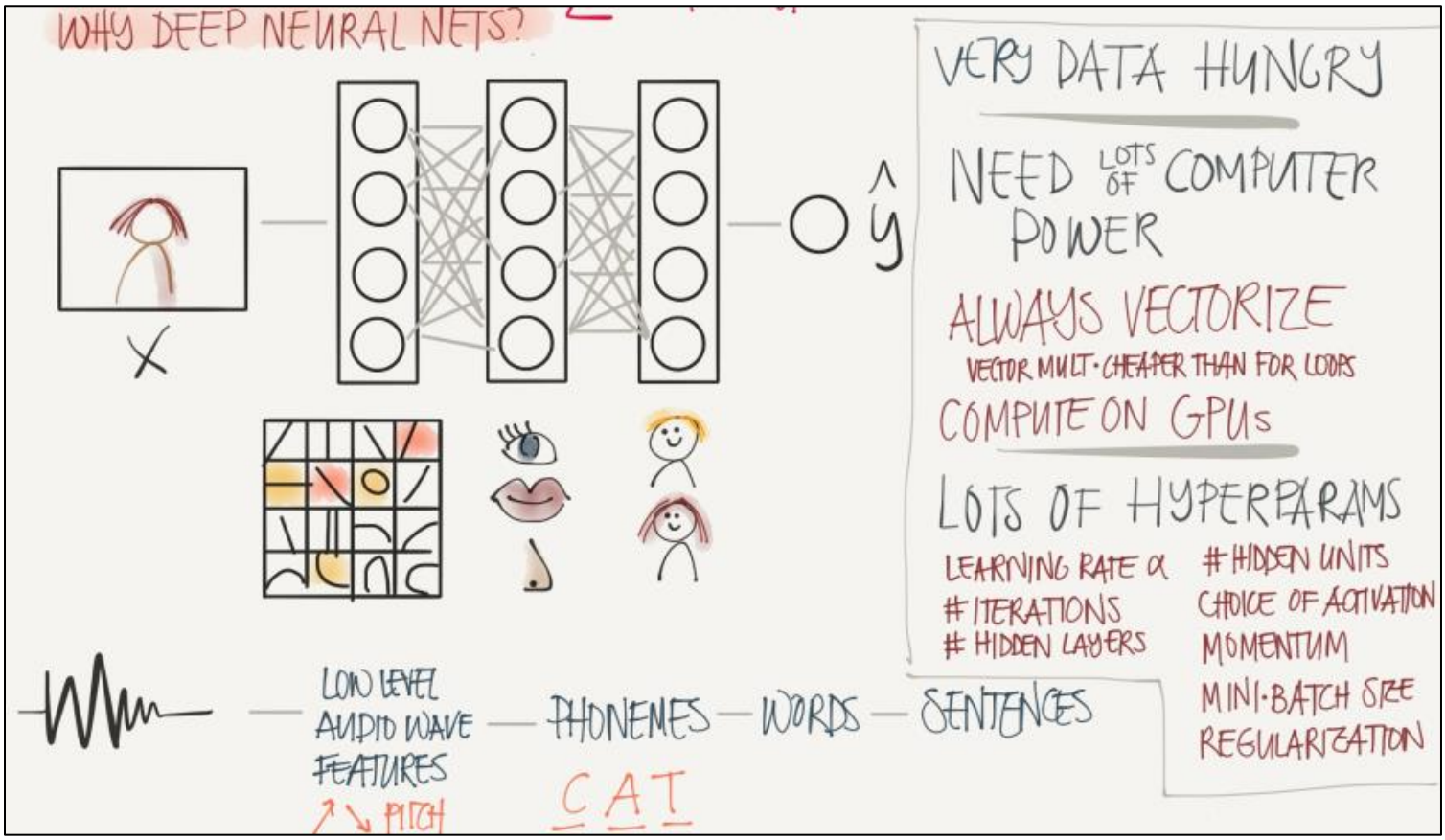
```
Z[l] = W[l]A[l-1] + B[l]
A[l] = g[l](A[l])
```

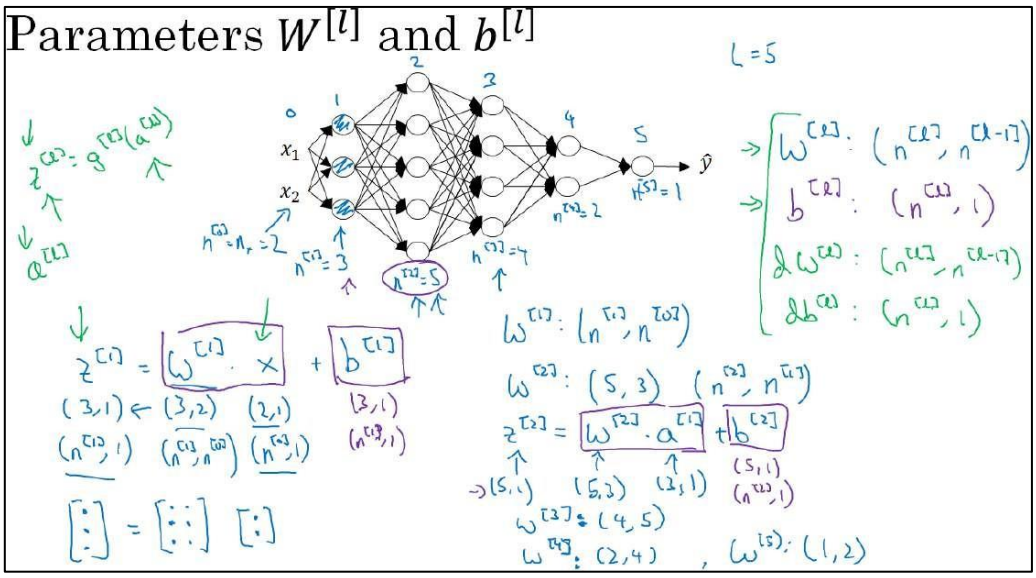
We can't compute the whole layers forward propagation without a for loop so its OK to have a for loop here.

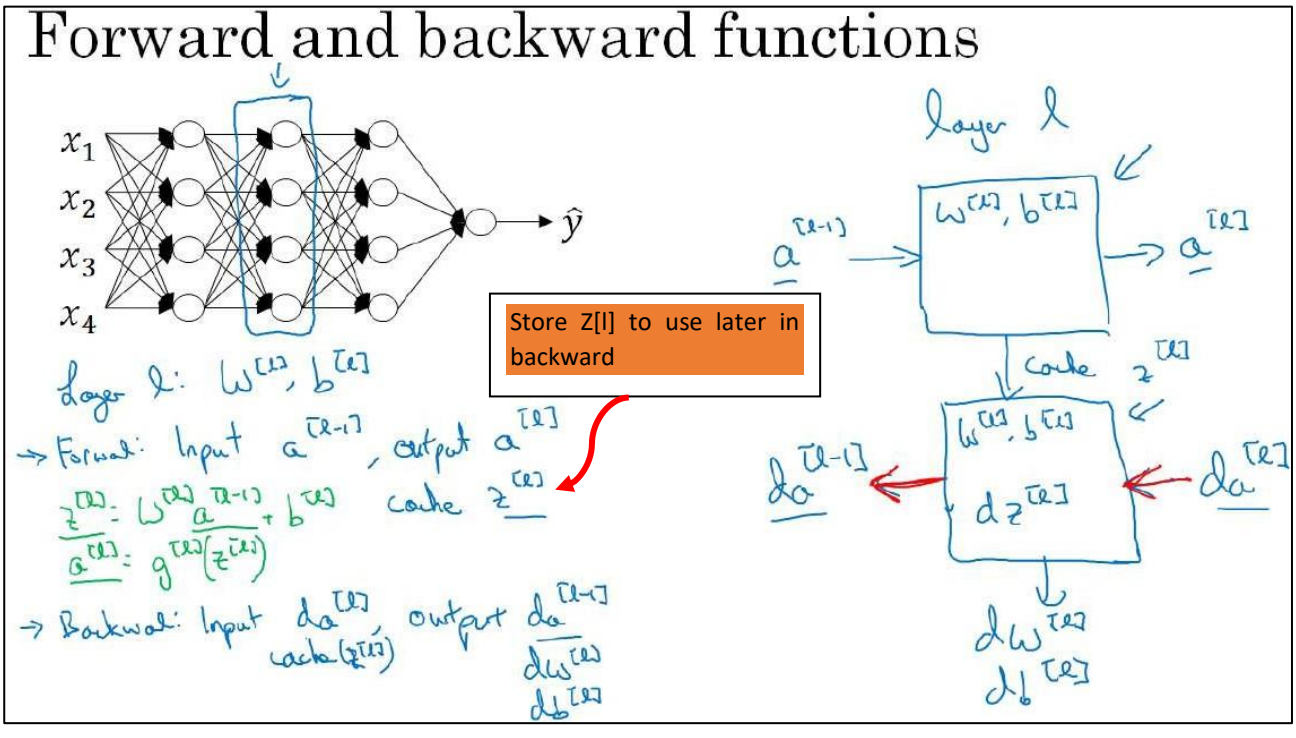
The dimensions of the matrices are so important you need to figure it out.

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 NEURAL NETS?
X
ŷ
VERY DATA HUNGRY
NEED LOTS OF COMPUTER POWER
ALWAYS VECTORIZE
VECTOR MULT · CHEAPER THAN FOR LOOPS
COMPUTE ON GPUs
LOTS OF HYPERPARAMS
LEARNING RATE α
ITERATIONS
HIDDEN LAYERS
HIDDEN UNITS
CHOICE OF ACTIVATION
MOMENTUM
MINI-BATCH SIZE
REGULARIZATION
LOW LEVEL AUDIO WAVE FEATURES
↗ ↘ PITCH
PHONEMES
CAT
WORDS
SENTENCES


Parameters $W^{[l]}$ and $b^{[l]}$
L = 5
$z^{[l]} = g^{[l]}(a^{[l]})$
$a^{[l]}$
x_1
x_2
$n^{[0]} = n_x = 2$
$n^{[1]} = 3$
$n^{[2]} = 5$
$n^{[3]} = 4$
$n^{[4]} = 2$
$n^{[5]} = 1$
$\hat{y}$
$W^{[l]} : (n^{[l]}, n^{[l-1]})$
$b^{[l]} : (n^{[l]}, 1)$
$dW^{[l]} : (n^{[l]}, n^{[l-1]})$
$db^{[l]} : (n^{[l]}, 1)$
$z^{[1]} = W^{[1]} \cdot x + b^{[1]}$
$(3,1) \leftarrow (3,2) \quad (2,1)$
$(n^{[1]}, 1) \quad (n^{[1]}, n^{[0]}) \quad (n^{[1]}, 1)$
$W^{[1]} : (n^{[1]}, n^{[0]})$
$W^{[2]} : (5, 3) \quad (n^{[2]}, n^{[1]})$
$z^{[2]} = W^{[2]} \cdot a^{[1]} + b^{[2]}$
$\rightarrow (5,1) \quad (5,3) \quad (3,1) \quad (5,1)$
$(n^{[2]}, 1)$
$W^{[3]} : (4, 5)$
$W^{[4]} : (2, 4) \quad , \quad W^{[5]} : (1, 2)$


Forward and backward functions
x_1
x_2
x_3
x_4
$\hat{y}$
layer l: $W^{[l]}, b^{[l]}$
→ Forward: Input $a^{[l-1]}$, output $a^{[l]}$
$z^{[l]} = W^{[l]} a^{[l-1]} + b^{[l]}$ cache $z^{[l]}$
$a^{[l]} = g^{[l]}(z^{[l]})$
→ Backward: Input $da^{[l]}$, cache ($z^{[l]}$), output $da^{[l-1]}$
$dW^{[l]}$
$db^{[l]}$
Store Z[l] to use later in backward
layer l
$W^{[l]}, b^{[l]}$
$a^{[l-1]}$
$a^{[l]}$
cache $z^{[l]}$
$dz^{[l]}$
$da^{[l-1]}$
$da^{[l]}$
$dW^{[l]}$
$db^{[l]}$

Forward propagation for layer l

→ Input $a^{[l-1]}$

→ Output $a^{[l]}$, cache $(z^{[l]})$

$$z^{[l]} = W^{[l]} \cdot a^{[l-1]} + b^{[l]}$$

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

Vectorized:

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

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

$a^{[0]}$ $A^{[0]}$ $X = A^{[0]}$ → [ ] → [ ] → [ ] →

Backward propagation for layer l

→ Input $da^{[l]}$

→ Output $da^{[l-1]}, dW^{[l]}, db^{[l]}$

$$dz^{[l]} = da^{[l]} * g^{[l]\prime}(z^{[l]})$$

$$dW^{[l]} = dz^{[l]} \cdot a^{[l-1]T}$$

$$db^{[l]} = dz^{[l]}$$

$$da^{[l-1]} = W^{[l]T} \cdot dz^{[l]}$$

$$dz^{[l]} = W^{[l+1]T} dz^{[l+1]} * g^{[l]\prime}(z^{[l]})$$

Vectorized:

$$dz^{[l]} = dA^{[l]} * g^{[l]\prime}(Z^{[l]})$$

$$dW^{[l]} = \frac{1}{m} dZ^{[l]} \cdot A^{[l-1]T}$$

$$db^{[l]} = \frac{1}{m} np.sum(dZ^{[l]}, axis=1, keepdims=True)$$

$$dA^{[l-1]} = W^{[l]T} \cdot dZ^{[l]}$$