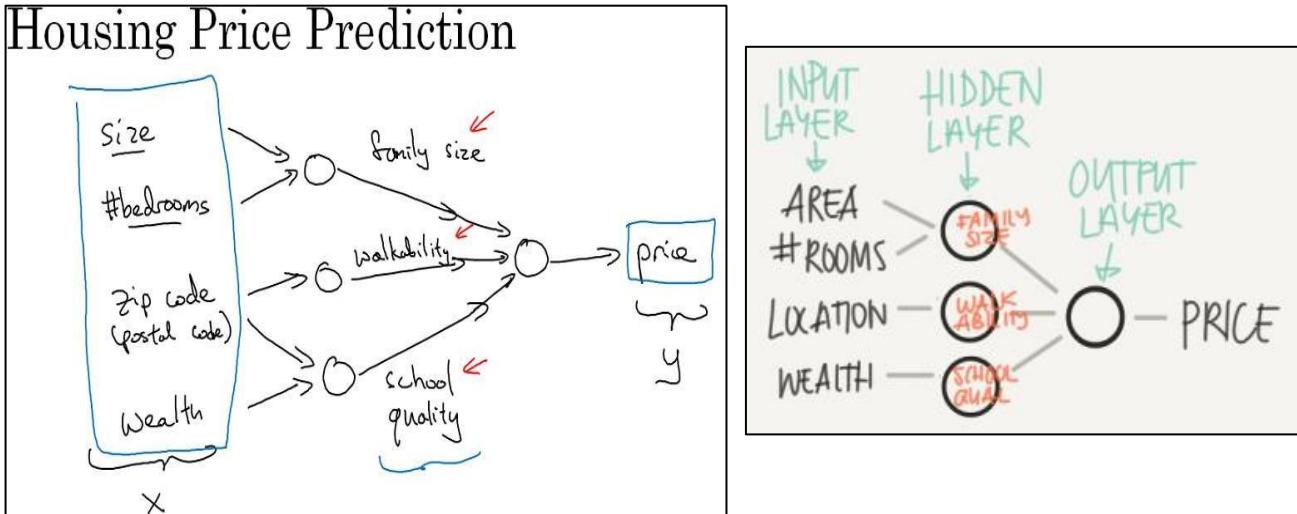


Introduction to Deep Learning

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

Housing Price Prediction



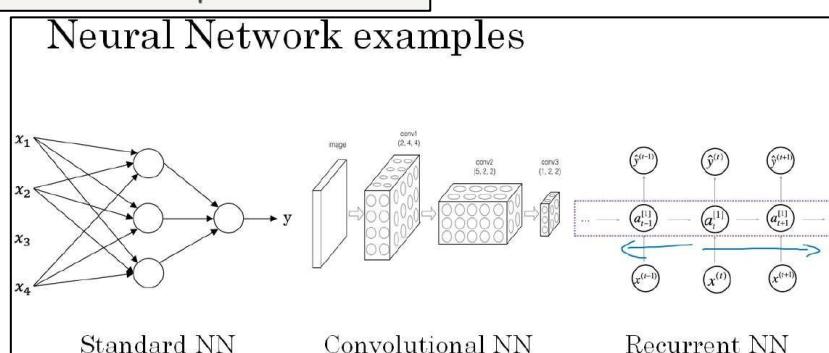
- The magic of a neural network is that when you implement it, you need to give it just the inputs "X" and the output "Y" for a number of examples in your training set and all these things in the middle (hidden layers), they will figure out by itself.

❖ Supervised learning

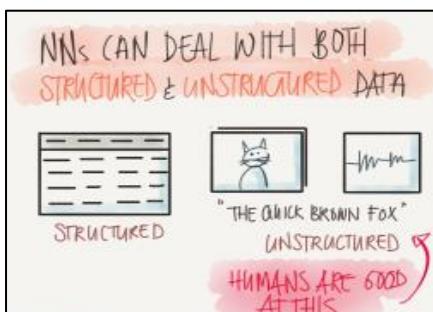
SUPERVISED LEARNING		
INPUT: X	OUTPUT: Y	NN TYPE
HOME FEATURES	PRICE	STANDARD NN
AD+USER INFO	WILL CLICK ON AD (0/1)	
IMAGE	OBJECT (1...1000)	CONV. NN (CNN)
AUDIO	TEXT TRANSCRIPT	RECURRENT NN (RNN)
ENGLISH	CHINESE	
IMAGE/RADAR	POS OF OTHER CARS	CUSTOM/HYBRID

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



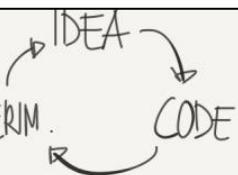
❖ Structured and Unstructured data



- 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 or 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 of deep learning



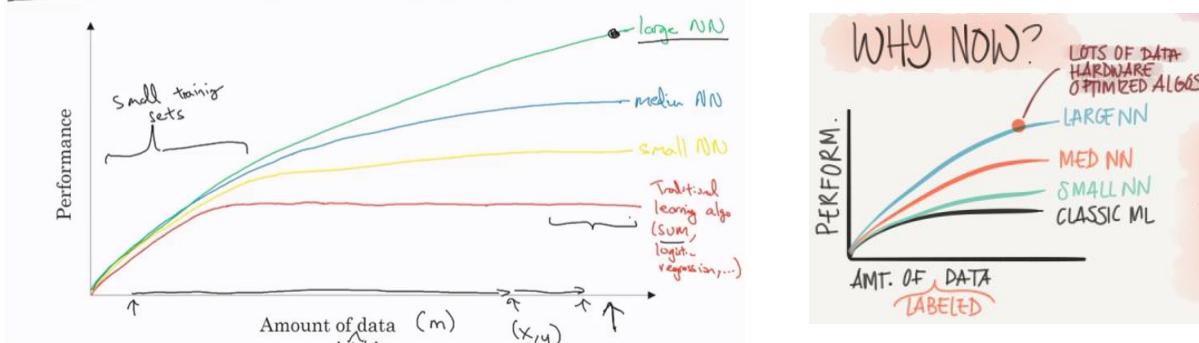
IDEA → **EXPERIMENT** → **CODE**

FASTER COMPUTATION IS IMPORTANT TO SPEED UP THE ITERATIVE PROCESS

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

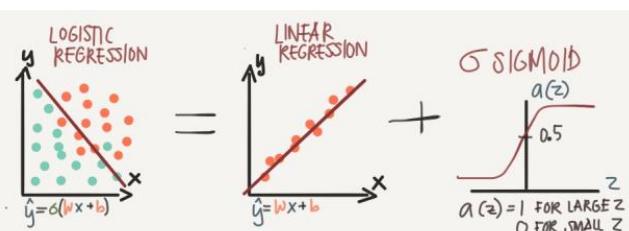
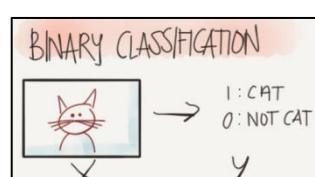
Scale drives deep learning progress



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



Some notations (we will see the details later)

- ✓ M is the number of training vectors
- ✓ N_x is the size of the input vector
- ✓ N_y is the size of the output vector
- ✓ $X(1)$ is the first input vector
- ✓ $Y(1)$ is the first output vector
- ✓ $X = [x(1) \ x(2) \dots \ x(M)]$
- ✓ $Y = [y(1) \ y(2) \dots \ y(M)]$

❖ Logistic Regression

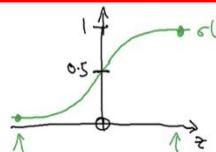
- Algorithm is used for classification algorithm of 2 classes.
- Equations:
 - Simple equation: $y = wx + b$
 - If x is a vector: $y = w(\text{transpose})x + b$
 - If we need y to be in between 0 and 1 (probability): $y = \text{sigmoid}(w(\text{transpose})x + b)$
 - In some notations this might be used: $y = \text{sigmoid}(w(\text{transpose})x)$
 - While b is w_0 of w and we add $x_0 = 1$. 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

Logistic Regression

Given x , want $\hat{y} = P(y=1|x)$
 $x \in \mathbb{R}^{n_x}$ $0 \leq \hat{y} \leq 1$

Parameters: $w \in \mathbb{R}^{n_x}$, $b \in \mathbb{R}$.

$$\text{Output } \hat{y} = \sigma(w^T x + b)$$



$$x_0 = 1, \quad x \in \mathbb{R}^{n_x+1}$$

$$\hat{y} = \sigma(w^T x)$$

$$w = \begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ \vdots \\ w_{n_x} \end{bmatrix} \quad b \leftarrow \begin{bmatrix} b \end{bmatrix}$$

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

$$\text{If } z \text{ large positive number: } \sigma(z) \approx \frac{1}{1+0} = 1$$

$$\text{If } z \text{ large negative number: } \sigma(z) \approx \frac{1}{1+e^{-z}} \times \frac{1}{1+\text{BigNum}} \approx 0$$

Then, we have: $\hat{y} = \sigma(w^T x + b)$

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 $y = 1 \Rightarrow L(y', 1) = -\log(y') \Rightarrow$ we want y' to be the largest $\Rightarrow y'$ biggest value is 1
 - if $y = 0 \Rightarrow L(y', 0) = -\log(1-y') \Rightarrow$ we want $1-y'$ to be the largest $\Rightarrow y'$ to be smaller as possible because it can only has 1 value.
- Then the Cost function will be: $J(w, b) = (1/m) * \text{Sum}(L(y'[i], y[i]))$
- The loss function computes the error for a single training example; the cost function is the average of the loss functions of the entire

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

Logistic Regression cost function

$$\hat{y}^{(i)} = \sigma(w^T x^{(i)} + b), \text{ where } \sigma(z^{(i)}) = \frac{1}{1+e^{-z^{(i)}}} \quad z^{(i)} = w^T x^{(i)} + b$$

Given $\{(x^{(1)}, y^{(1)}), \dots, (x^{(m)}, y^{(m)})\}$, want $\hat{y}^{(i)} \approx y^{(i)}$.

Loss(error) function: $L(\hat{y}, y) = \frac{1}{2} (\hat{y} - y)^2$

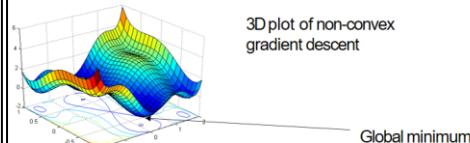
$$L(\hat{y}, y) = -[y \log \hat{y} + (1-y) \log(1-\hat{y})]$$

If $y=1$: $L(\hat{y}, y) = -\log \hat{y} \leftarrow$ want $\log \hat{y}$ large, want \hat{y} large.
 If $y=0$: $L(\hat{y}, y) = -\log(1-\hat{y}) \leftarrow$ want $\log(1-\hat{y})$ large ... want \hat{y} small

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

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.



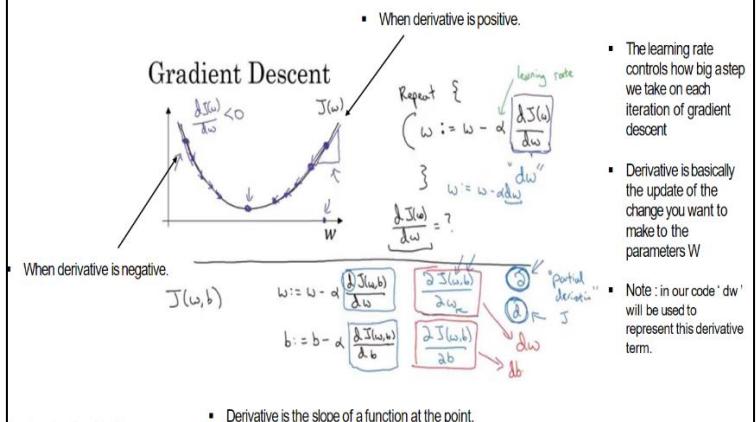
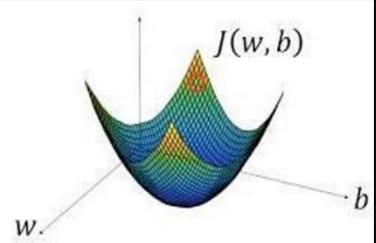
❖ Logistic Regression : Gradient Descent

- 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 to reach minimum value.
- The gradient descent algorithm repeats: $w = w - \alpha * dw$ where α is the learning rate and dw is the derivative of w (Change to w) The derivative is also the slope of w
- 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 b 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 B it seems natural that we want to find W and B that make the cost function $J(W, b)$ as small.



These equations are equivalent.

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

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

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

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

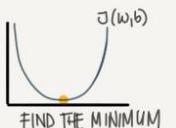
THE TASK IS TO LEARN w & b BUT HOW?

A: OPTIMIZE HOW GOOD THE GUESS IS BY MINIMIZING THE DIFF BETWEEN GUESS (\hat{y}) AND TRUTH (y)

$$\text{LOSS} = L(\hat{y}, y)$$

$$\text{COST} = J(w, b) = \frac{1}{m} \sum_{i=1}^m L(\hat{y}^{(i)}, y^{(i)})$$

COST = LOSS FOR THE ENTIRE DATASET



❖ 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:

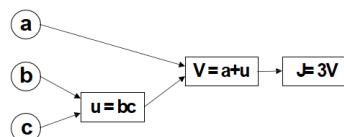
- Compute bc and store it in the variable u , so $u=bc$
- Compute $V=a+u$,
- Output $J=3V$.

$$J(a,b,c)=3(a+bc)$$

$$u=bc$$

$$V=a+u$$

$$J=3V$$



The computations of a neural network are organized in terms of

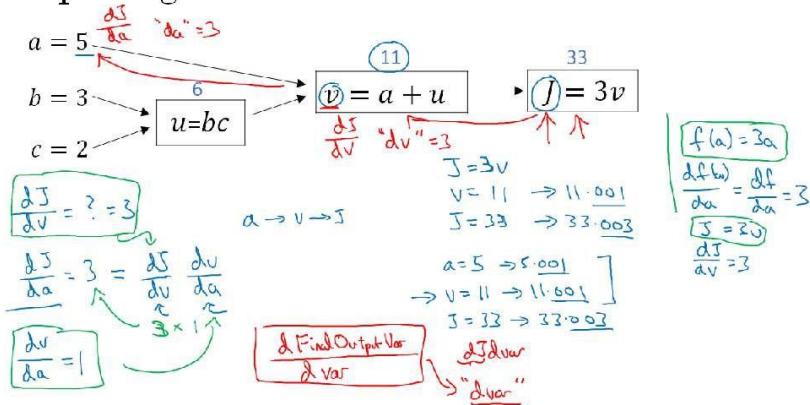
- a **forward pass** or a **forward propagation step**, in which we compute the output of the neural network,
- followed by a **backward pass** or **back propagation step**, which we use to compute gradients or compute derivatives.

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 a Computing graph

Computing derivatives



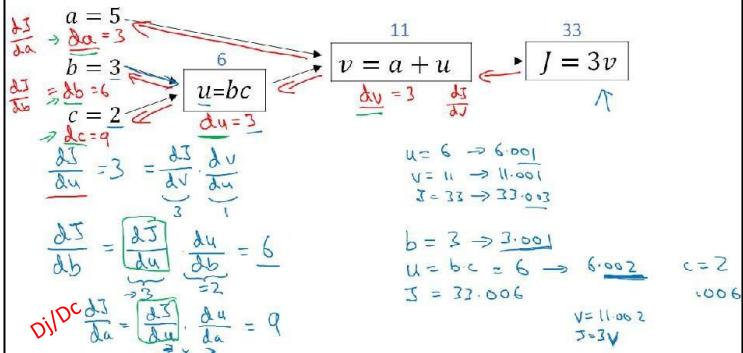
- Calculus chain rule says: If $x \rightarrow y \rightarrow z$ (x effect y and y effects z) Then :

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

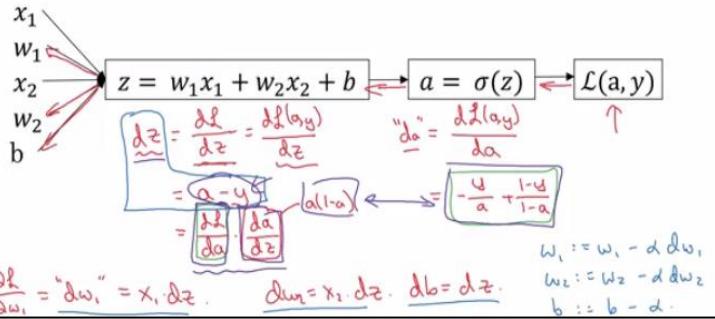
- We compute the derivatives on a graph from right to left and it will be a lot more easier.
- dvar means the derivatives of a final output variable with respect to various intermediate quantities.

❖ Logistic Regression : Gradient Descent

Computing derivatives



Logistic regression derivatives



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

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

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

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

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

$$w1 = w1 - \alpha * dw1$$

$$w2 = w2 - \alpha * dw2$$

$$b = b - \alpha * db$$

❖ 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

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

Logistic regression on m examples

$$\begin{aligned} J &= 0; \underline{dw_1} = 0; \underline{dw_2} = 0; \underline{db} = 0 \\ \text{For } i &= 1 \text{ to } m \\ z^{(i)} &= w^T x^{(i)} + b \\ a^{(i)} &= \sigma(z^{(i)}) \\ J &= -[y^{(i)} \log a^{(i)} + (1 - y^{(i)}) \log(1 - a^{(i)})] \\ dz^{(i)} &= a^{(i)} - y^{(i)} \\ dw_1 &+= x_1^{(i)} dz^{(i)} \quad | \quad n=2 \\ dw_2 &+= x_2^{(i)} dz^{(i)} \\ db &+= dz^{(i)} \\ J &= m \leftarrow \\ dw_1 &= m; \quad dw_2 = m; \quad db = m. \leftarrow \\ \uparrow & \uparrow \quad \uparrow \end{aligned}$$

$$dw_1 = \frac{\partial J}{\partial w_1}$$

$$w_1 := w_1 - \alpha \underline{dw_1}$$

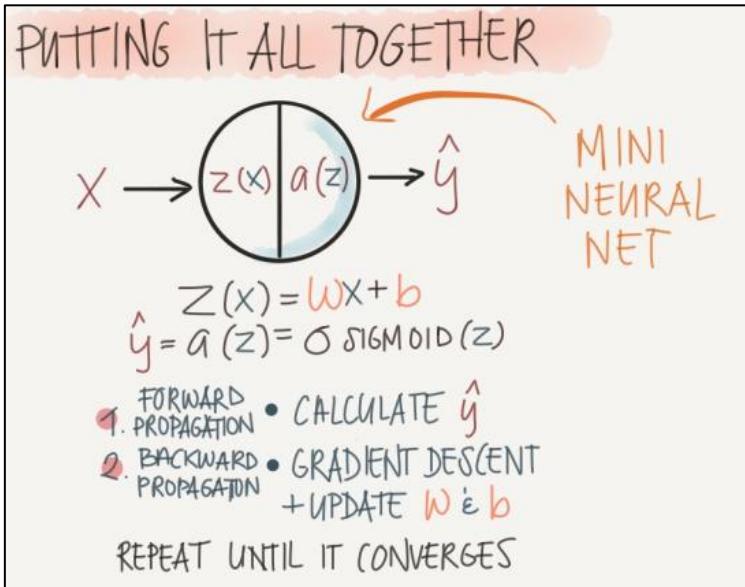
$$w_2 := w_2 - \alpha \underline{dw_2}$$

$$b := b - \alpha \underline{db}$$

Vectorization

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

$$\begin{aligned} d(a) &= d(1)/d(a) = -(y/a) + ((1-y)/(1-a)) \\ d(z) &= d(1)/d(z) = a - y \\ d(W1) &= X1 * d(z) \\ d(W2) &= X2 * d(z) \\ d(B) &= d(z) \end{aligned}$$



❖ 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.

Non-vectorized:

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

Vectorized

$$z = \underbrace{\text{np.dot}(w, x)}_{w^T x} + b$$

→ GPU } SIMD - single instruction multiple data.
→ CPU }

- 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 $[z_1, z_2, \dots, z_m] = w^T * X + [b, b, \dots, b]$. 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, db shape is (1, 1)
```

Vectorizing Logistic Regression

$$\begin{aligned} z^{(1)} &= w^T x^{(1)} + b \\ \hat{a}^{(1)} &= \sigma(z^{(1)}) \\ X &= \begin{bmatrix} x_1^{(1)} & x_2^{(1)} & \dots & x_n^{(1)} \end{bmatrix} \\ \tilde{Z} &= \begin{bmatrix} z^{(1)} & z^{(2)} & \dots & z^{(m)} \end{bmatrix} = w^T X + [b, b, \dots, b] \\ A &= \begin{bmatrix} a^{(1)} & a^{(2)} & \dots & a^{(m)} \end{bmatrix} = \sigma(\tilde{Z}) \end{aligned}$$

Vectorizing Logistic Regression

$$\begin{aligned} dz^{(1)} &= a^{(1)} - y^{(1)} & dz^{(2)} &= a^{(2)} - y^{(2)} & \dots \\ d\tilde{Z} &= \begin{bmatrix} dz^{(1)} & dz^{(2)} & \dots & dz^{(m)} \end{bmatrix} \\ A &= [a^{(1)} \dots a^{(m)}] & Y &= [y^{(1)} \dots y^{(m)}] \\ \Rightarrow dZ &= A - Y = [a^{(1)} - y^{(1)} \quad a^{(2)} - y^{(2)} \quad \dots] \\ \Rightarrow dw &= 0 & db &= 0 \\ dw &+ \frac{1}{m} \sum_{i=1}^m x^{(i)} dz^{(i)} & db &+ \frac{1}{m} \sum_{i=1}^m dz^{(i)} \\ dw &+ \frac{1}{m} X dZ & db &+ \frac{1}{m} \sum_{i=1}^m dz^{(i)} \\ dw &= \frac{1}{m} X dZ & db &= \frac{1}{m} \sum_{i=1}^m dz^{(i)} \end{aligned}$$

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

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

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

- In python:

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

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

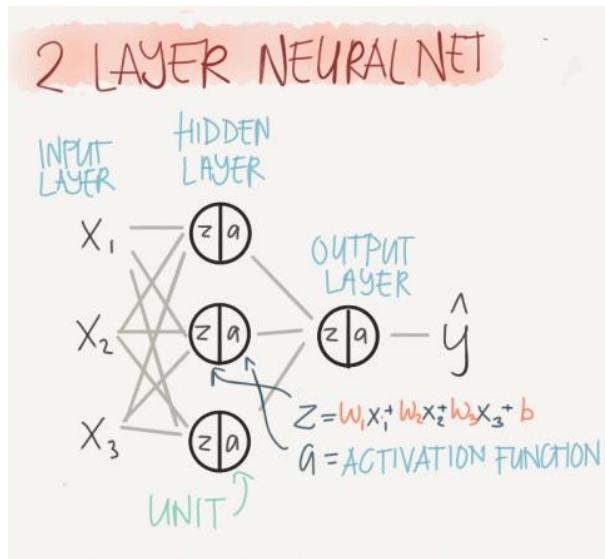
```
Z = np.dot(w.T, X) + b
A = sigma(Z)
dZ = A - Y
dw = 1/m * X dZ^T
db = 1/m np.sum(dZ)
w := w - alpha dw
b := b - alpha db
```

❖ Notes on python and Numpy

Python/numpy vectors

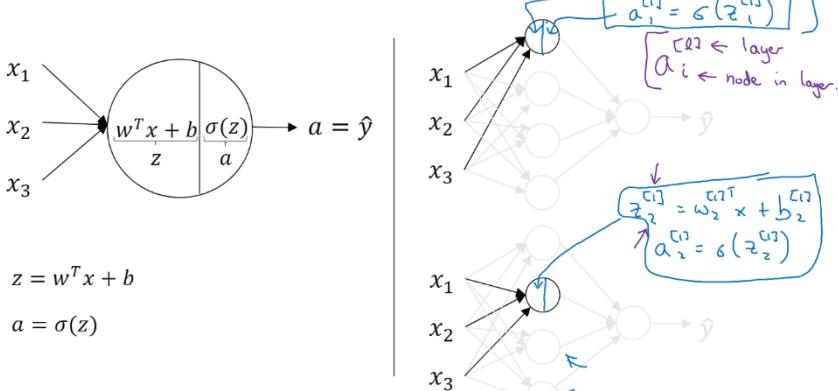
```
a = np.random.randn(5)
a.shape = (5,)           } Don't use
"rank 1 array"
a = np.random.randn(5,1) → a.shape = (5,1)   column ✓
a = np.random.randn(1,5) → a.shape = (1,5)   row ✓
assert(a.shape == (5,1)) ←
a = a.reshape((5,1))
```

❖ 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 can't 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



Neural Network Representation

x_1 x_2 x_3

$\rightarrow z^{[1]} = \begin{bmatrix} w_1^{[1]T} \\ w_2^{[1]T} \\ w_3^{[1]T} \\ w_4^{[1]T} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \\ b_4^{[1]} \end{bmatrix}$

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

$\rightarrow z^{[2]} = \begin{bmatrix} w_1^{[2]T} \\ w_2^{[2]T} \\ w_3^{[2]T} \\ w_4^{[2]T} \end{bmatrix} \begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \\ a_4^{[1]} \end{bmatrix} + \begin{bmatrix} b_1^{[2]} \\ b_2^{[2]} \\ b_3^{[2]} \\ b_4^{[2]} \end{bmatrix}$

$\rightarrow a^{[2]} = \begin{bmatrix} a_1^{[2]} \\ a_2^{[2]} \\ a_3^{[2]} \\ a_4^{[2]} \end{bmatrix} = \sigma(z^{[2]})$

$= \begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \\ z_4^{[1]} \end{bmatrix} + \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \\ b_4^{[1]} \end{bmatrix} = \begin{bmatrix} w_1^{[1]T} x + b_1^{[1]} \\ w_2^{[1]T} x + b_2^{[1]} \\ w_3^{[1]T} x + b_3^{[1]} \\ w_4^{[1]T} x + b_4^{[1]} \end{bmatrix} = \begin{bmatrix} z_1^{[2]} \\ z_2^{[2]} \\ z_3^{[2]} \\ z_4^{[2]} \end{bmatrix}$

Justification for vectorized implementation

$$\begin{aligned}
 z^{(1)} &= w^{(1)} x + b^{(1)}, & z^{(2)} &= w^{(2)} x + b^{(2)}, & z^{(3)} &= w^{(3)} x + b^{(3)} \\
 w^{(1)} &= \begin{bmatrix} \vdots \\ \vdots \end{bmatrix}, & w^{(2)} &= \begin{bmatrix} \vdots \\ \vdots \end{bmatrix}, & w^{(3)} &= \begin{bmatrix} \vdots \\ \vdots \end{bmatrix} \\
 w^{(1)} x &= \begin{bmatrix} \vdots \\ \vdots \end{bmatrix}, & w^{(2)} x &= \begin{bmatrix} \vdots \\ \vdots \end{bmatrix}, & w^{(3)} x &= \begin{bmatrix} \vdots \\ \vdots \end{bmatrix} \\
 w^{(1)} x^{(1)} &= \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \end{bmatrix}, & w^{(2)} x^{(1)} &= \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \end{bmatrix}, & w^{(3)} x^{(1)} &= \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \end{bmatrix} = z^{(1)} \\
 w^{(1)} x^{(2)} &= \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \end{bmatrix}, & w^{(2)} x^{(2)} &= \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \end{bmatrix}, & w^{(3)} x^{(2)} &= \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \end{bmatrix} = z^{(2)} \\
 w^{(1)} x^{(m)} &= \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \end{bmatrix}, & w^{(2)} x^{(m)} &= \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \end{bmatrix}, & w^{(3)} x^{(m)} &= \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \end{bmatrix} = z^{(m)}
 \end{aligned}$$

Andrew Ng

Vectorizing across multiple examples

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

$X = \begin{bmatrix} x^{(1)} & x^{(2)} & \dots & x^{(m)} \end{bmatrix}$
 (n_x, m) samples
 hidden units.

$Z^{(1)} = \begin{bmatrix} z^{(1)(1)} & z^{(1)(2)} & \dots & z^{(1)(m)} \end{bmatrix}$
 $A^{(1)} = \begin{bmatrix} a^{(1)(1)} & a^{(1)(2)} & \dots & a^{(1)(m)} \end{bmatrix}$
 hidden units.

Andrew Ng

Recap of vectorizing across multiple examples

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

for $i = 1$ to m
 $\quad z^{(1)(i)} = W^{[1]} x^{(i)} + b^{[1]}$
 $\quad a^{(1)(i)} = \sigma(z^{(1)(i)})$
 $\quad z^{(2)(i)} = W^{[2]} a^{(1)(i)} + b^{[2]}$
 $\quad a^{(2)(i)} = \sigma(z^{(2)(i)})$
 $\quad A^{[1]} = \sigma(W^{[1]} X + b^{[1]})$
 $\quad Z^{[1]} = W^{[1]} X + b^{[1]}$
 $\quad A^{[1]} = \sigma(Z^{[1]})$
 $\quad Z^{[2]} = W^{[2]} A^{[1]} + b^{[2]}$
 $\quad A^{[2]} = \sigma(Z^{[2]})$

Andrew Ng

❖ Activation functions

ACTIVATION FUNCTIONS

SIGMOID σ	TANH	RELU	LEAKY RELU
BINARY CLASSIFIER ONLY USED FOR OUTPUT LAYER <u>SLOW GRAD</u> DESCENT SINCE SLOPE IS SMALL FOR LARGE/SMALL VAL	NORMALIZED \Rightarrow GRADIENT DESCENT IS FASTER	DEFAULT CHOICE FOR ACTIVATION SLOPE = 1/0	AVoids UNDEF SLOPE AT 0 BUT RARELY USED IN PRACTICE

- 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.

- 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:

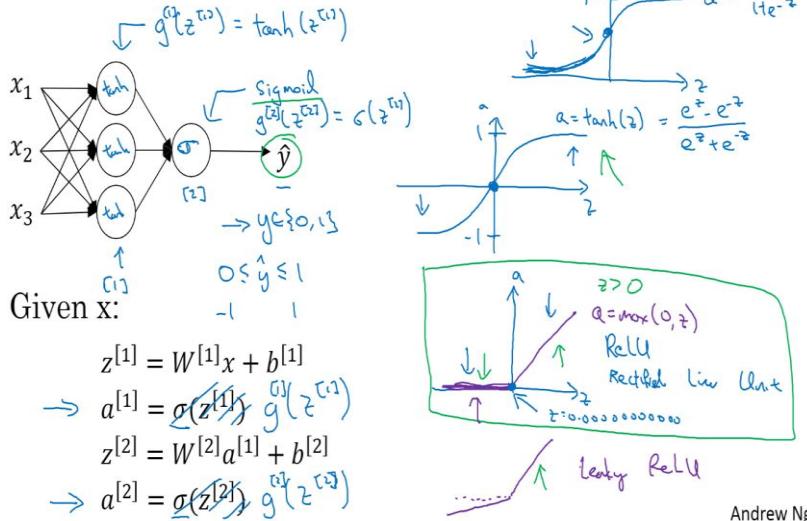
$A = (np.exp(z) - np.exp(-z)) / (np.exp(z) + np.exp(-z))$

Where z is the input matrix

Or $A = np.tanh(z)$

Where z is the input matrix

Activation functions



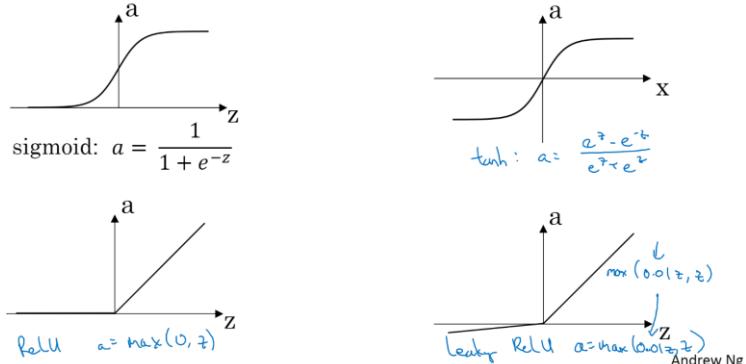
- One of the popular activation functions that solved the slow gradient decent is the RELU function. $\text{RELU} = \max(0, z)$

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. $\text{Leaky RELU} = \max(0.01z, z)$ #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.

Pros and cons of activation functions



WHY ACTIVATION FUNCTIONS?

Ex. with no activation - $a = z$

$$a^{[1]} = z^{[1]} = W^{[1]}X + b^{[1]} \quad \text{LAYER 1}$$

$$a^{[2]} = z^{[2]} = W^{[2]}a^{[1]} + b^{[2]} \quad \text{LAYER 2}$$

PLUG IN $a^{[1]}$

$$a^{[2]} = W^{[2]}(W^{[1]}X + b^{[1]}) + b^{[2]}$$

$$= W^{[2]}W^{[1]}X + W^{[2]}b^{[1]} + b^{[2]} \quad \leftarrow \text{LINEAR FUNCTION}$$

INITIALIZING $W+b$

WHAT IF: INIT TO \emptyset

THIS WILL CAUSE ALL THE UNITS TO BE THE SAME AND LEARN EXACTLY THE SAME FEATURES

SOLUTION: RANDOM INIT
BUT ALSO WANT THEM SMALL SO RAND * 0.01

WE COULD JUST AS WELL HAVE SKIPPED THE WHOLE NEURAL NET & USED LIN. REGR.

HYPERPARAM

❖ Gradient descent for NNs

Gradient descent for neural networks

$$\text{Parameters: } \begin{pmatrix} \omega_{11} & b_{11} \\ \omega_{12}, n_{12} & (n_{12})' \\ \omega_{21}, n_{21} & (n_{21})' \end{pmatrix}, \quad \begin{pmatrix} \omega_{22} & b_{22} \\ n_{22}, n_{22} & (n_{22})' \\ n_{12}, n_{21} & (n_{12}, 1) \end{pmatrix}, \quad n_x = n^{[0]}, \quad n^{[1]}, \quad \underline{n^{[2]} = 1}$$

$$\text{Cost Function: } J(\underset{\substack{\leftarrow \\ \text{W}^{[1]}}}{{\color{blue}w}^{[1]}}, \underset{\substack{\leftarrow \\ \text{b}^{[1]}}}{{\color{blue}b}^{[1]}}, \underset{\substack{\leftarrow \\ \text{W}^{[2]}}}{{\color{red}W}^{[2]}}, \underset{\substack{\leftarrow \\ \text{b}^{[2]}}}{{\color{red}b}^{[2]}}) = \frac{1}{m} \sum_{i=1}^m \ell(\hat{y}_i, y_i)$$

Graham Descent

Report {
 → Compute predict. ($\hat{y}^{(i)}$, $i=1 \dots m$)
 $\frac{\partial L}{\partial w^{(j)}} = \frac{\partial J}{\partial w^{(j)}}, \quad \frac{\partial L}{\partial b^{(j)}} = \frac{\partial J}{\partial b^{(j)}}, \dots$
 $w^{(j)} := w^{(j)} - \alpha \frac{\partial L}{\partial w^{(j)}}$
 $b^{(j)} := b^{(j)} - \alpha \frac{\partial L}{\partial b^{(j)}}$
 $u^{(j)} := \dots \quad b^{(j)} := \dots$

Andrew Ng

- 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'(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

```

- 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 $J = J(w_1, b_1, w_2, b_2) = \frac{1}{m} * \text{Sum}(L(Y, A2))$

- Then Gradient descent:

Repeat:

Compute predictions ($y'[i]$, $i = 0, \dots, m$)

Get derivatives: dw1, db1, dw2, db2

Update: $W_1 = W_1 - \text{LearningRate} * dW_1$

```
b1 = b1 - LearningRate * db1
```

$$W_2 = W_2 - \text{LearningRate} * dW_2$$

Formulas for computing derivatives

$$\begin{aligned} \text{Forward propagation:} \\ z^{[1]} &= w^{[1]}X + b^{[1]} \\ A^{[1]} &= g^{[1]}(z^{[1]}) \leftarrow \\ z^{[2]} &= w^{[2]}A^{[1]} + b^{[2]} \\ A^{[2]} &= g^{[2]}(z^{[2]}) = \underline{o}(z^{[2]}) \end{aligned}$$

Back propagation:

$$\frac{\partial E^{(2)}}{\partial A^{(2)}} = A^{(2)} - Y \leftarrow$$

$$\frac{\partial W^{(2)}}{\partial} = \frac{1}{m} \frac{\partial E^{(2)}}{\partial A^{(2)}} A^{(1)T}$$

$$\frac{\partial b^{(2)}}{\partial} = \frac{1}{m} \text{np.sum}(\frac{\partial E^{(2)}}{\partial}, \text{axis}=1, \text{keepdims=True})$$

$$\frac{\partial Z^{(1)}}{\partial} = \underbrace{W^{(2)T} \frac{\partial E^{(2)}}{\partial}}_{(n^{(2)}, m)} \times \underbrace{g^{(2)'}(Z^{(2)})}_{\text{element-wise product}} \quad (n^{(2)}, m)$$

$$\frac{\partial W^{(1)}}{\partial} = \frac{1}{n} \frac{\partial E^{(2)}}{\partial} X^T$$

$$\frac{\partial b^{(1)}}{\partial} = \frac{1}{n} \text{np.sum}(\frac{\partial E^{(2)}}{\partial}, \text{axis}=1, \text{keepdims=True})$$

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

Andrew Ng

❖ 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

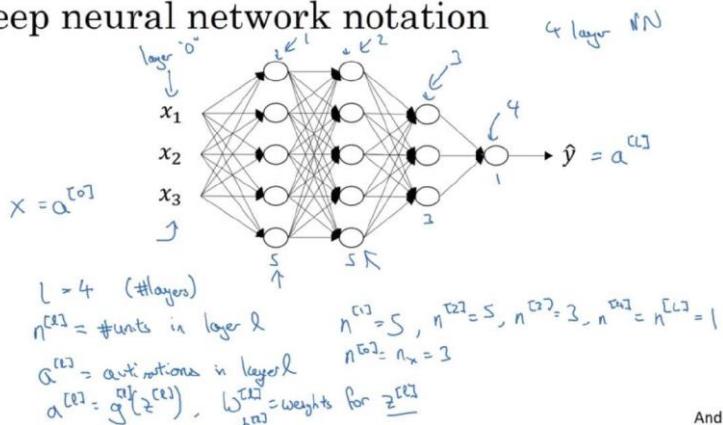
- To solve this we initialize the W's with a small random numbers:

```
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.

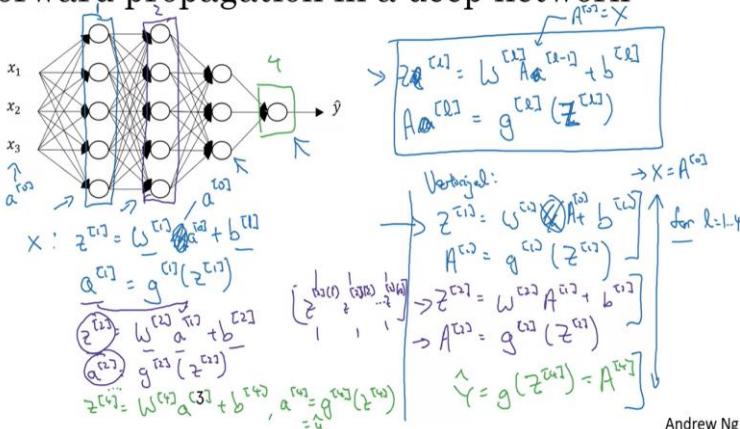
❖ Deep Neural Network

Deep neural network notation



- 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.
- $n[1]$ is the number of neurons in a specific layer 1.
- $n[0]$ denotes the number of neurons input layer. $n[L]$ denotes the number of neurons in output layer.
- $g[1]$ is the activation function.
- $a[1] = g[1](z[1])$
- $w[1]$ weights is used for $z[1]$
- $x = a[0], a[1] = y'$

Forward propagation in a deep network



- These were the notation we will use for deep neural network.
- So we have:
 - A vector n of shape $(1, \text{NoOfLayers}+1)$
 - A vector g of shape $(1, \text{NoOfLayers})$
 - 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:

$$z[1] = W[1]a[1-1] + b[1]$$

$$a[1] = g[1](a[1])$$

- Forward propagation general rule for m inputs:

$$Z[1] = W[1]A[1-1] + B[1]$$

$$A[1] = g[1](A[1])$$

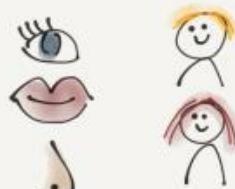
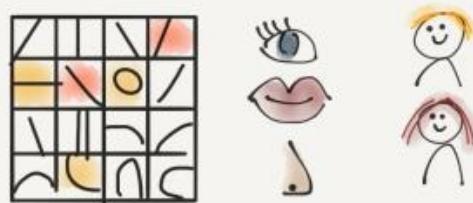
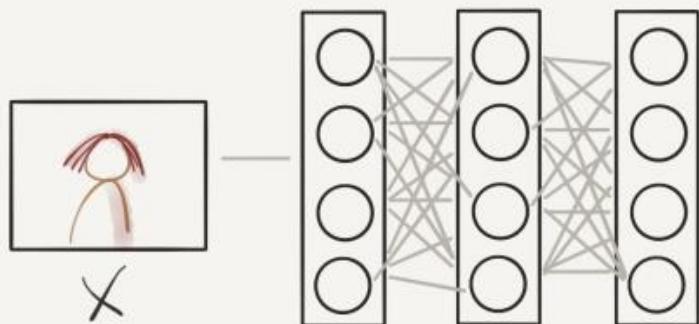
We can't compute the whole layers forward propagation without a for loop so its OK to have a for loop here.

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 $(n[1], n[1-1])$. Can be thought by right to left.
- Dimension of b is $(n[1], 1)$
- dw has the same shape as W , while db is the same shape as b
- Dimension of $Z[1]$, $A[1]$, $dZ[1]$, and $dA[1]$ is $(n[1], m)$

WHY DEEP NEURAL NETS?



LOW LEVEL
AUDIO WAVE
FEATURES
↗ PITCH

CAT

PHONEMES

WORDS

SENTENCES

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

HIDDEN UNITS

ITERATIONS

CHOICE OF ACTIVATION

HIDDEN LAYERS

MOMENTUM

MINI-BATCH SIZE

REGULARIZATION

Parameters $W^{[l]}$ and $b^{[l]}$

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

$z^{[1]} = \underbrace{w^{[1]} \cdot x}_{(3,1) \leftarrow (3,2) \cdot (2,1)} + \underbrace{b^{[1]}_{(2,1)}}_{(n^{[1]}, 1)}$

$z^{[2]} = \underbrace{w^{[2]} \cdot a^{[1]}_{(3,1)}}_{(5,1) \leftarrow (5,3) \cdot (2,1)} + \underbrace{b^{[2]}_{(2,1)}}_{(n^{[2]}, 1)}$

$z^{[3]} = \underbrace{w^{[3]} \cdot a^{[2]}_{(5,1)}}_{(4,1) \leftarrow (4,5) \cdot (2,1)} + \underbrace{b^{[3]}_{(1,1)}}_{(n^{[3]}, 1)}$

$z^{[4]} = \underbrace{w^{[4]} \cdot a^{[3]}_{(4,1)}}_{(2,1) \leftarrow (2,4) \cdot (1,1)} + \underbrace{b^{[4]}_{(1,1)}}_{(n^{[4]}, 1)}$

$z^{[5]} = \underbrace{w^{[5]} \cdot a^{[4]}_{(2,1)}}_{(1,2) \leftarrow (1,2) \cdot (1,1)} + \underbrace{b^{[5]}_{(1,2)}}_{(n^{[5]}, 1)}$

$L=5$

$$\begin{cases} w^{[1]}: (n^{[1]}, n^{[L-1]}) \\ b^{[1]}: (n^{[1]}, 1) \\ dw^{[1]}: (n^{[1]}, n^{[L-1]}) \\ db^{[1]}: (n^{[1]}, 1) \end{cases}$$

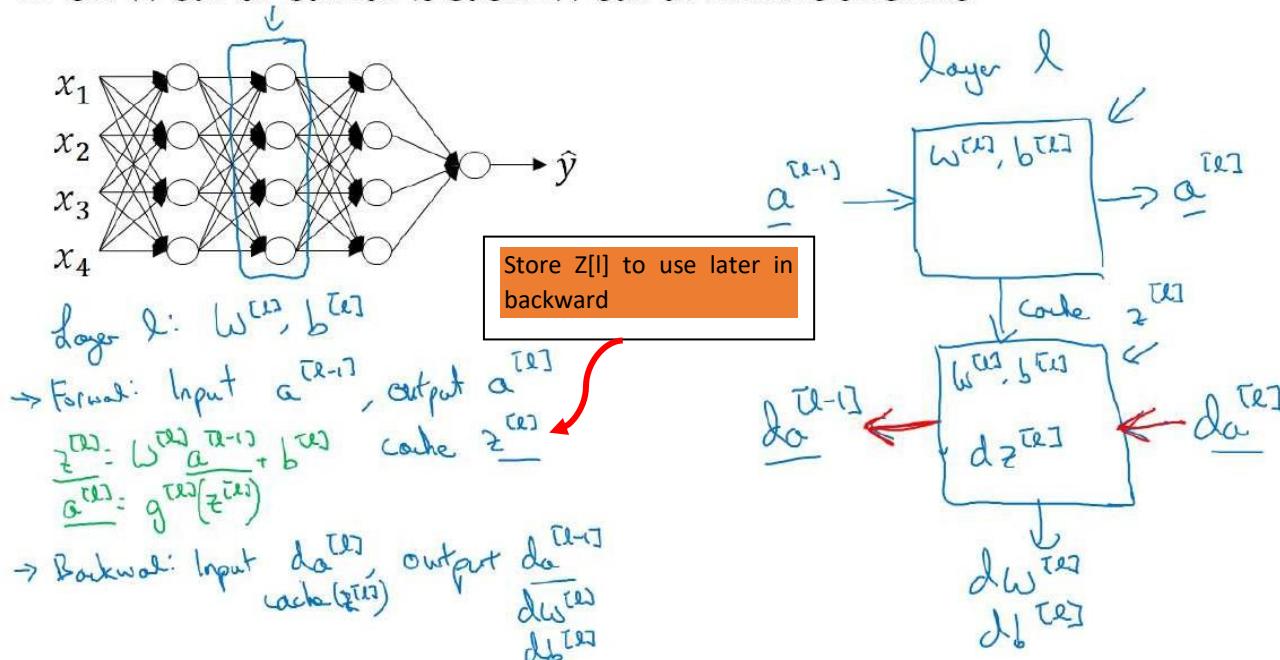
$$\begin{cases} w^{[2]}: (n^{[1]}, n^{[2]}) \\ b^{[2]}: (n^{[2]}, 1) \\ dw^{[2]}: (n^{[2]}, n^{[L-1]}) \\ db^{[2]}: (n^{[2]}, 1) \end{cases}$$

$$\begin{cases} w^{[3]}: (n^{[2]}, n^{[3]}) \\ b^{[3]}: (n^{[3]}, 1) \\ dw^{[3]}: (n^{[3]}, n^{[L-1]}) \\ db^{[3]}: (n^{[3]}, 1) \end{cases}$$

$$\begin{cases} w^{[4]}: (n^{[3]}, n^{[4]}) \\ b^{[4]}: (n^{[4]}, 1) \\ dw^{[4]}: (n^{[4]}, n^{[L-1]}) \\ db^{[4]}: (n^{[4]}, 1) \end{cases}$$

$$\begin{cases} w^{[5]}: (n^{[4]}, n^{[5]}) \\ b^{[5]}: (n^{[5]}, 1) \\ dw^{[5]}: (n^{[5]}, n^{[L-1]}) \\ db^{[5]}: (n^{[5]}, 1) \end{cases}$$

Forward and backward functions



Forward propagation for layer l

$$\rightarrow \text{Input } a^{[l-1]} \leftarrow$$

$$\rightarrow \text{Output } a^{[l]}, \text{cache } (z^{[l]})$$

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

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

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

$$x = h^{[l]} \rightarrow \square \rightarrow \square \rightarrow \square \rightarrow$$

$$a^{[0]} \quad A^{[0]}$$

Verw:

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

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

Backward propagation for layer l

$$\rightarrow \text{Input } da^{[l]}$$

$$\rightarrow \text{Output } \underline{da^{[l-1]}}, \underline{dW^{[l]}}, \underline{db^{[l]}}$$

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

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

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

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

$$dz^{[l]} = W^{[l+1]T} \cdot dz^{[l+1]} * g^{[l+1]}'(z^{[l+1]})$$

$$dz^{[l]} = \underline{dA^{[l]}} * g^{[l]}'(z^{[l]})$$

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

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

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

Initialize all parameters

Initialize
W1, b1 ... Initialize
WL, bL

loop for num_iterations

