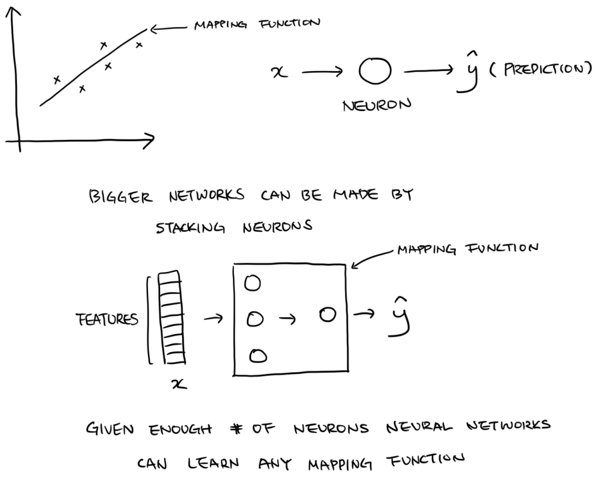
Neural Networks and Deep Neural Networks

Swapnoneel Kayal

1 June-7 June 2021

1 Neural Networks

Neural network is a stack of **neurons** that takes in some value and outputs some value. Given enough number of neurons, neural networks are incredibly good at **mapping x** to **y**. During **supervised learning**, we use this property to learn a **function** that maps **x** to **ŷ**. To learn that function, we need data. There are two types of data: **structured** and **unstructured**. Structured data are well organized into arrays and often comes from a database. Unstructured data are things like audio, image, text, and etc. that does not come in typical structure.


MAPPING FUNCTION
x
NEURON
ŷ (PREDICTION)
BIGGER NETWORKS CAN BE MADE BY STACKING NEURONS
MAPPING FUNCTION
FEATURES
x
ŷ
GIVEN ENOUGH # OF NEURONS NEURAL NETWORKS CAN LEARN ANY MAPPING FUNCTION

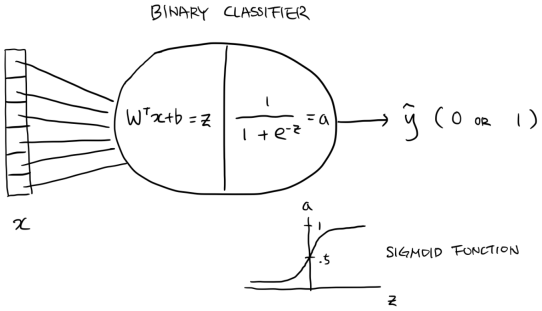
Binary classifier predicts the **class** of an input x by computing **ŷ** (**0** or **1** for two classes). We use a **neuron** that maps **x** to **ŷ** such that:

$$\hat{\mathbf{y}} = \boldsymbol{\sigma}\left(\boldsymbol{W}^T\boldsymbol{x} + \boldsymbol{b}\right)$$

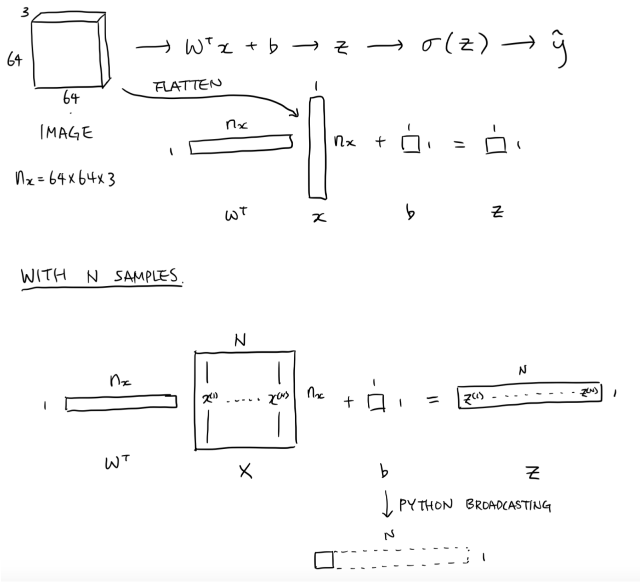
$$\text{where} \quad \boldsymbol{\sigma}(z) = \frac{1}{1+e^{-z}}$$

$$\therefore \ \hat{\mathbf{y}} = \frac{1}{1+e^{-\left(\boldsymbol{W}^T\boldsymbol{x}+\boldsymbol{b}\right)}}$$

Sigmoid function outputs a value between 0 and 1, which happens to work beautifully for our purpose of binary classification.



To use a 64x64x3 image as an input to our neuron, we need to flatten the image into a (64x64x3)x1 vector and to make $\boldsymbol{W}^T\boldsymbol{x}+\boldsymbol{b}$ output a single value **z**, we need **W** to be a (64x64x3)x1 vector: (dimension of input)x(dimension of output), and **b** to be a single value. With **N** number of images, we can make a matrix **X** of shape (64x64x3)x**N**. $\boldsymbol{W}^T\boldsymbol{x}+\boldsymbol{b}$ outputs **Z** of shape 1x**N** containing **z**'s for every single sample, and by passing **Z** through a sigmoid function we get final $\hat{\mathbf{y}}$ of shape 1x**N** that contains predictions for every single sample. We do not have to explicitly create a **b** of 1x**N** with the same value copied **N** times, thanks to Python **broadcasting**.



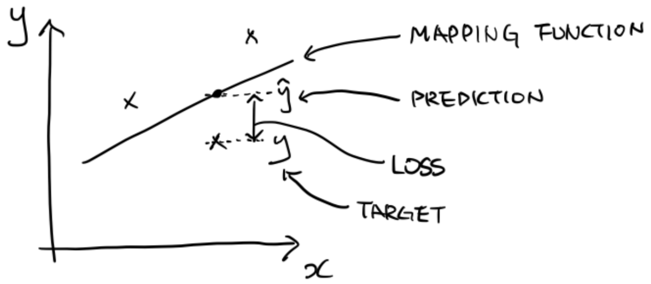
y is our given **target** or **label**. During **supervised learning**, our goal is to try to learn the exact mapping from **x** to **y**. In other words, we want to figure out a mapping function that would give us **prediction** $\hat{\mathbf{y}}$ as close as possible to **target y**. We use **loss function** to measure how close $\hat{\mathbf{y}}$ is to **y**. We could simply use the distance or use something better like **logarithmic/cross-entropy loss**:

$$J(\boldsymbol{\theta}) = -\frac{1}{m}\sum_{i=1}^{m}\left[\mathbf{y}^{(i)}\log\hat{\mathbf{y}} + \left(1-\mathbf{y}^{(i)}\right)\log\left(1-\hat{\mathbf{y}}\right)\right]$$

Now, we can simplify our goal to minimizing the loss function. When we have multiple $\hat{\mathbf{y}}$'s and **y**'s from multiple **x**'s, we take the average of **loss** to get **cost**.

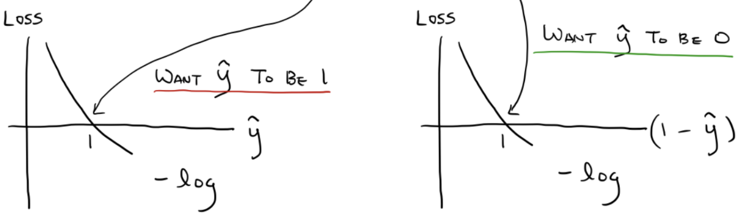
GIVEN $\{(x^{(1)}, y^{(1)}), \cdots, (x^{(N)}, y^{(N)})\}$

WANT $\hat{y}^{(i)} \approx y^{(i)}$



WANT TO MINIMIZE LOSS

$$\text{LOG LOSS} = \begin{cases} -\log \hat{y} & \text{IF } y = 1 \\ -(\log 1 - \hat{y}) & \text{IF } y = 0 \end{cases}$$



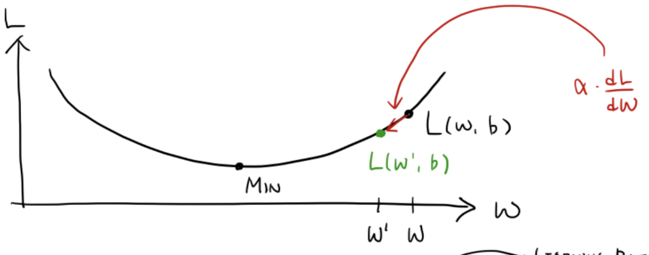
$$\text{COST} = \frac{\sum_{i=1}^{N} \text{LOSS}^{(i)}}{N}$$

We use **gradient descent** to achieve our goal of finding the **W** and **b** that minimizes the loss function given by **L(W,b)** as given below.

To do this, we compute **L**, then derive **dL/dW** and **dL/db**, to use them for updating **W** and **b** so that they would give a better **L** when we compute it with the same **x** next time. The derivatives provide the directions, while **learning rate** denoted by α determines the magnitude of the changes we make. The **update rule** has been given below. Using the **chain rule**, we can easily derive **dL/dW** and **dL/db**. In deep learning, we call the chain of computations up to **L**, the **forward propagation**, and the following chain of computations to **dL/dW** and **dL/db**, the **backward propagation**. **Computation graphs** helps us visualize the chains. Finally, **logistic regression** is simply repeating the process of our forward propagation, backward propagation, and update parameters, to find **W** and **b** that maps **x** to $\hat{\mathbf{y}}$ with minimum **logarithmic loss** compared to **y**.

Want to find W, b that minimizes

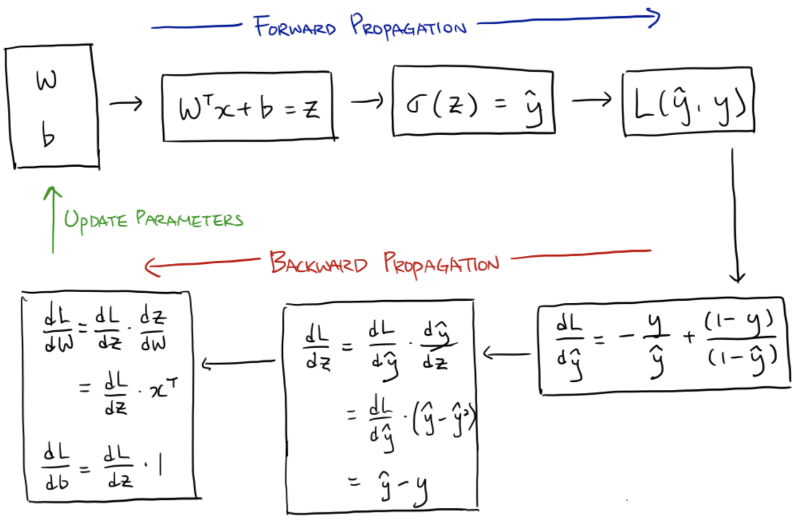
$$L(w,b) = -(y \log \sigma(W^T x + b) + (1-y) \log(1-\sigma(W^T x+b)))$$



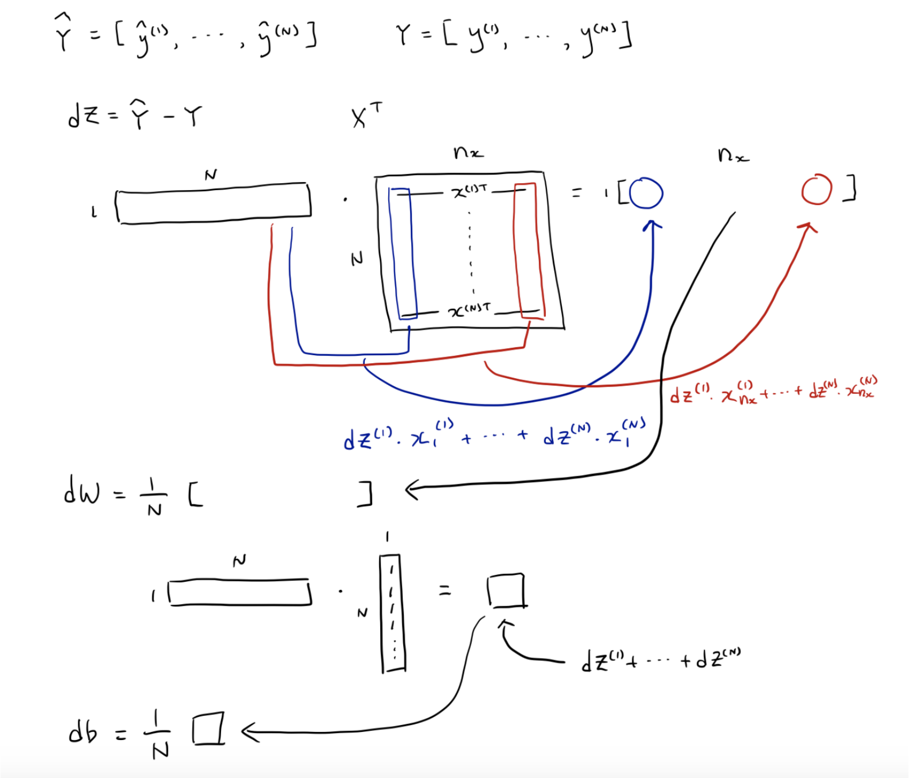
$$\text{Update Rule}: W' = W - \overset{\text{Learning Rate}}{\alpha} \frac{dL}{dW}$$

$$\longrightarrow L(w',b) < L(w,b)$$

Logistic Regession

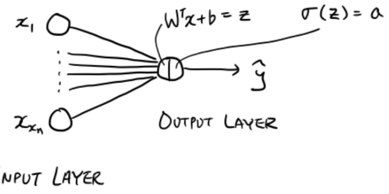

Forward Propagation
W
b
$W^T x + b = z$
$\sigma(z) = \hat{y}$
$L(\hat{y}, y)$
Update Parameters
Backward Propagation
$\frac{dL}{d\hat{y}} = -\frac{y}{\hat{y}} + \frac{(1-y)}{(1-\hat{y})}$
$\frac{dL}{dz} = \frac{dL}{d\hat{y}} \cdot \frac{d\hat{y}}{dz}$
$= \frac{dL}{d\hat{y}} \cdot (\hat{y} - \hat{y}^2)$
$= \hat{y} - y$
$\frac{dL}{dW} = \frac{dL}{dz} \cdot \frac{dz}{dW}$
$= \frac{dL}{dz} \cdot x^T$
$\frac{dL}{db} = \frac{dL}{dz} \cdot 1$

What happens to **dL/dW**(**dW** from here on) and **dL/db**(**db** from here on) when we have multiple x's? As mentioned before with **N** number of **x**'s, we create a **nx** by **N** matrix **X** that has **x**'s as columns. Similarly, its target **Y** is a 1x**N** row vector with **N** number of **y**'s stacked horizontally. Forward propagation of **X** computes $\hat{\mathbf{Y}}$ with the same shape as **Y** and their **cost** by taking an average over multiple losses. Likewise, we will have to take an average over multiple **dW**'s and **db**'s. With 1x**N** $\hat{\mathbf{Y}}$ and **Y**, **dZ** will be 1x**N** as well, carrying **dz**'s for N samples. Taking the dot product between 1x**N** **dZ** and **N**x**nx** **X.T** results in 1x**nx** row vector just like **W**. Even with just one sample, when dz is 1x1 and x.T is 1x**nx**, the dot product between the two outputs 1x**nx**. Then what's the difference? With **N** number of samples, each element in the row vector becomes a sum of **dW**'s from **N** samples, due to dot product. To take an average, all we have to do is dividing the 1x**nx** row vector by **N**. Unfortunately and obviously, there is no Python broadcasting for dot products. To mimic the operation with **dW** for **db**, I prefer to take a dot product between 1x**N** **dZ** and **N**x1 vector of ones, then divide the resulting number with **N** to take an average. But in the course, we take the sum of **dZ** over **N** samples and divide it by **N**.

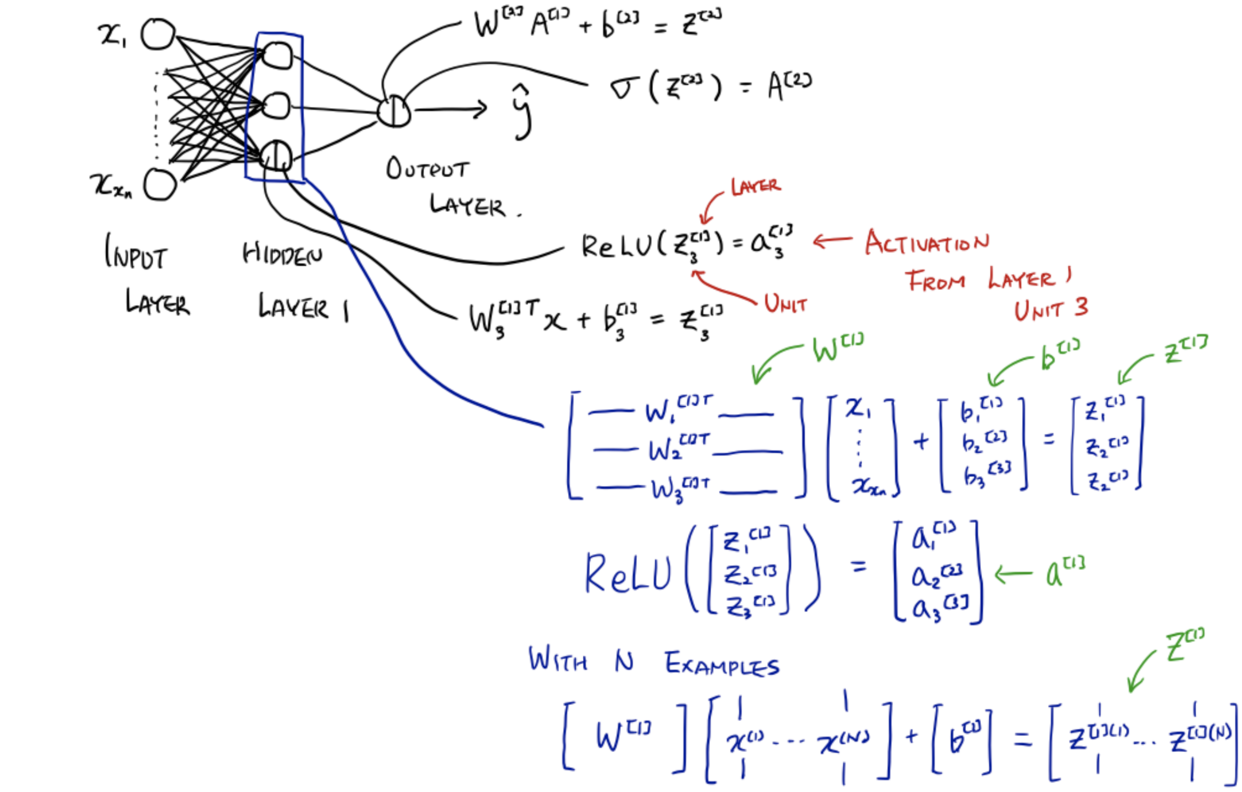

$\hat{Y} = [\hat{y}^{(1)}, \cdots, \hat{y}^{(N)}]$
$Y = [y^{(1)}, \cdots, y^{(N)}]$
$dZ = \hat{Y} - Y$
X^T
n_x
n_x
N
1
N
$x^{(1)T}$
$x^{(N)T}$
$= 1[$
$]$
$dz^{(1)} \cdot x_{n_x}^{(1)} + \cdots + dz^{(N)} \cdot x_{n_x}^{(N)}$
$dz^{(1)} \cdot x_1^{(1)} + \cdots + dz^{(N)} \cdot x_1^{(N)}$
$dW = \frac{1}{N}[\quad]$
N
1
1
N
=
$dz^{(1)} + \cdots + dz^{(N)}$
$db = \frac{1}{N}$

The beauty of the **forward** and **backward propagation** is that it works seamlessly even when we stack bunch of **neurons** to make a **neural network**. For **input layer** is just a placeholder for inputs and does not compute anything, we do not count it as **neurons**. So, in our **logistic regression model**, we only had a single **neuron** in **output layer**. To turn it into a **neural network binary classifier**, we add hidden layers in between the **input** and **output layers**. Each **hidden layer** have multiple number of **neurons**, and each **neuron**'s output is called an **activation**. **Activations** from a **hidden layer** become **inputs** for the next layer. The **activation** of the **neuron** in **output layer** is $\hat{\mathbf{y}}$. And it's **activation function** that wraps $W^T\mathbf{x} + \mathbf{b}$ is a **sigmoid function**. Likewise, every **neuron in hidden layers (hidden units)** computes **linear** operation $W^T\mathbf{x} + \mathbf{b}$ then pass it to a **nonlinear activation function** to output an **activation**. **ReLU function** is a better choice of **activation function** for **hidden units** because unlike in the **output layer**, we do not need to squeeze **activations** from **hidden units** to be in between **0** and **1**. Additionally, $\mathbf{ReLU(z)} = \max(0, \mathbf{z})$; therefore, it does not suffer from having very small gradients at the ends, helping the **neural network** to learn much faster. **Activation functions** have to be **nonlinear** because composition of **linear** functions only result in another linear function, which is are not complex enough to solve complex problems. Repeatedly computing activations individually takes a lot of for-loops that makes things run very slow. So, we use **vectorization** to summarize for-loops into one matrix dot product. This is not a new concept for us because we already used it to make matrix **X** out of multiple **x**'s by just stacking them as columns. Likewise, we will stack multiple **W**'s for each units in the same layer as columns and transpose it to get the vectorized $W^{[1]}$. The shape of $W^{[1]}$ is **number of units in current layer** x **number of units from previous layer**. With vectorized $b^{[1]}$ and **X**, $W^{[1]}\mathrm{X} + b^{[1]}$ will give **number of units** x **N** shaped $Z^{[1]}$. And, passing it through activation function will give $A^{[1]}$ with the same shape. $A^{[1]}$ is passed onto the next layer (output layer), which computes $W^{[2]}A^{[1]} + b^{[2]} = Z^{[2]} and A^{[2]} = \sigma(Z^{[2]})$.

Logistic Regression Model

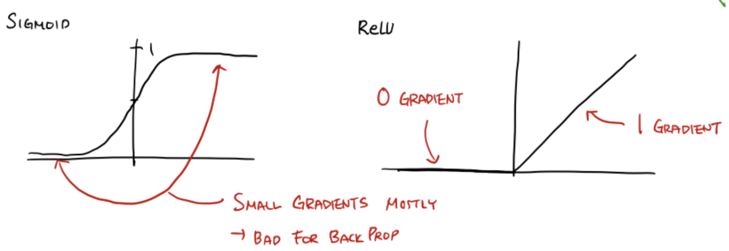


Neural Network (Single Hidden Layer)

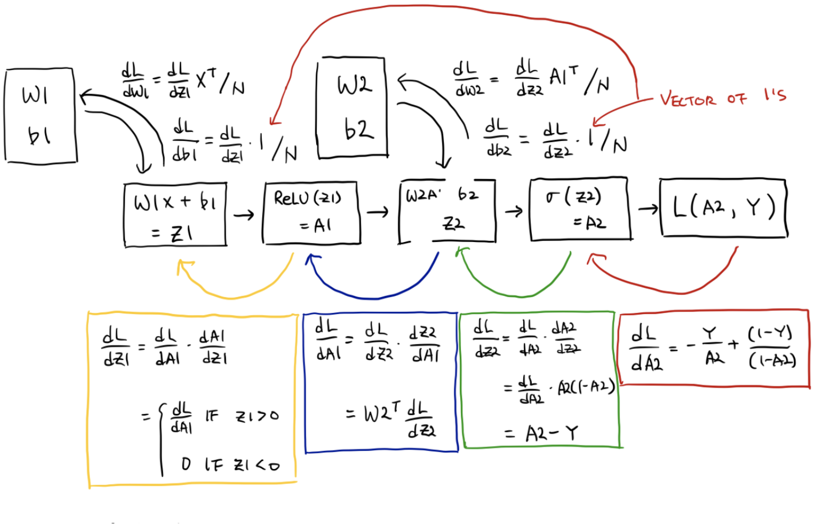

x_1
x_{x_n}
Input Layer
Hidden Layer 1
Output Layer.
$W^{[2]}A^{[1]}+b^{[2]}=z^{[2]}$
$\sigma(z^{[2]})=A^{[2]}$
$\hat{y}$
Layer
$\mathrm{ReLU}(z_3^{[1]})=a_3^{[1]}$
← Activation From Layer 1 Unit 3
Unit
$W_3^{[1]T}x+b_3^{[1]}=z_3^{[1]}$
$W^{[1]}$
$b^{[1]}$
$z^{[1]}$
$\begin{bmatrix} — W_1^{[1]T} — \\ — W_2^{[1]T} — \\ — W_3^{[1]T} — \end{bmatrix}\begin{bmatrix} x_1 \\ \vdots \\ x_{x_n}\end{bmatrix}+\begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]}\end{bmatrix}=\begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_2^{[1]}\end{bmatrix}$
$\mathrm{ReLU}\left(\begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]}\end{bmatrix}\right)=\begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]}\end{bmatrix}$ ← $a^{[1]}$
With N Examples
$Z^{[1]}$
$\begin{bmatrix} W^{[1]} \end{bmatrix}\begin{bmatrix} | & & | \\ x^{(1)} & \cdots & x^{(N)} \\ | & & | \end{bmatrix}+\begin{bmatrix} b^{[1]} \end{bmatrix}=\begin{bmatrix} | & & | \\ z^{[1](1)} & \cdots & z^{[1](N)} \\ | & & | \end{bmatrix}$

$$\mathrm{ReLU}\left(\begin{bmatrix} | & & | \\ z^{[1](1)} & \cdots & z^{[1](N)} \\ | & & | \end{bmatrix}\right)=\underbrace{\begin{bmatrix} | & & | \\ a^{[1](1)} & \cdots & a^{[1](N)} \\ | & & | \end{bmatrix}}_{A^{[1]}}$$

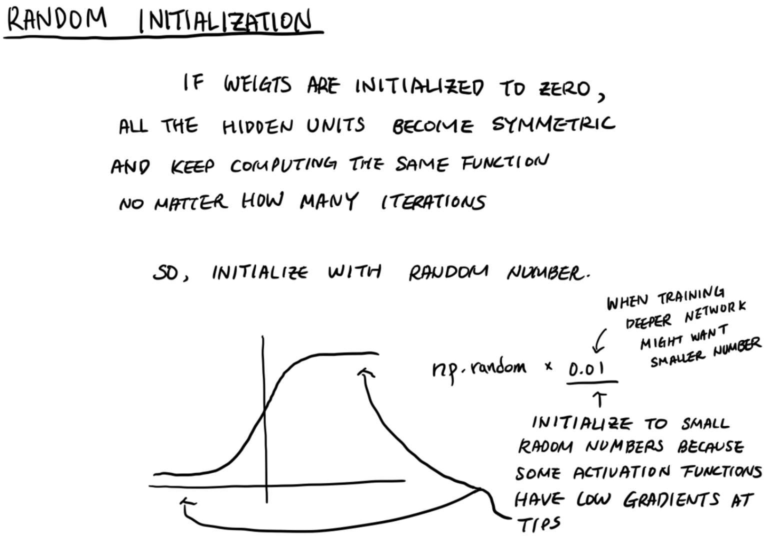
Activation Functions



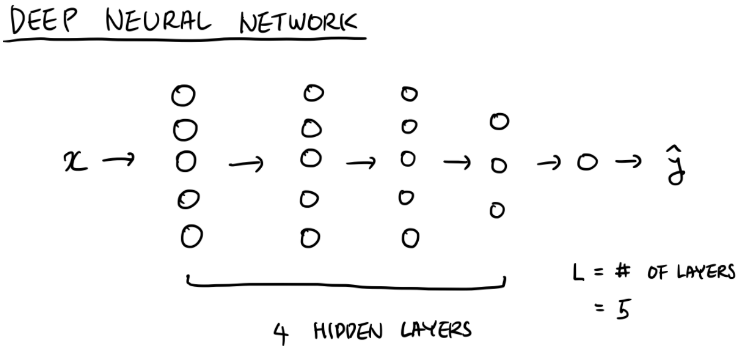
For **backward propagation**, the derivative of **a = sigmoid(z)** is **a(1-a)**, and the derivative of **a = ReLU(z)** is **0 if z < 0** and **1 if z ≥ 0**. Let's take a look at the **computation graph** of a **neural network** with a **single hidden layer**.


W1
b1
$\frac{dL}{dW1} = \frac{dL}{dZ1} X^T / N$
$\frac{dL}{db1} = \frac{dL}{dZ1} \cdot 1 / N$
W2
b2
$\frac{dL}{dW2} = \frac{dL}{dZ2} A1^T / N$
$\frac{dL}{db2} = \frac{dL}{dZ2} \cdot 1 / N$
VECTOR OF 1'S
W1x + b1 = Z1
ReLU(Z1) = A1
W2A' b2 Z2
σ(Z2) = A2
L(A2, Y)
$\frac{dL}{dZ1} = \frac{dL}{dA1} \cdot \frac{dA1}{dZ1} = \begin{cases} \frac{dL}{dA1} \text{ IF } Z1 > 0 \\ 0 \text{ IF } Z1 < 0 \end{cases}$
$\frac{dL}{dA1} = \frac{dL}{dZ2} \cdot \frac{dZ2}{dA1} = W2^T \frac{dL}{dZ2}$
$\frac{dL}{dZ2} = \frac{dL}{dA2} \cdot \frac{dA2}{dZ2} = \frac{dL}{dA2} \cdot A2(1-A2) = A2 - Y$
$\frac{dL}{dA2} = -\frac{Y}{A2} + \frac{(1-Y)}{(1-A2)}$

As mentioned, **gradient descent** is an iterative process that steps toward the point with minimum loss. To find **W**'s and **b**'s that give the minimum loss, we need to set the starting point. If we initialize the parameters (**W**'s and **b**'s) with zeros, all the **hidden units** will compute the same **activations** and during **backward propagation**, get the same **gradients**. Basically, all the **hidden units** will become symmetric and keep computing the same function no matter how may iterations. So, we initialize the parameters with random number. When using **sigmoid** as **activation functions**, it is important to keep the numbers small because gradients are nearly 0 at the ends.


RANDOM INITIALIZATION
IF WEIGTS ARE INITIALIZED TO ZERO,
ALL THE HIDDEN UNITS BECOME SYMMETRIC
AND KEEP COMPUTING THE SAME FUNCTION
NO MATTER HOW MANY ITERATIONS
SO, INITIALIZE WITH RANDOM NUMBER.
np.random × 0.01
WHEN TRAINING DEEPER NETWORK MIGHT WANT SMALLER NUMBER
INITIALIZE TO SMALL RADOM NUMBERS BECAUSE SOME ACTIVATION FUNCTIONS HAVE LOW GRADIENTS AT TIPS

By putting everything together, we can build a **binary classifier**. First, we **load data** and **initialize parameters**, then repeat **forward-backward propagation** and **parameter updates**. We have seen in detail what goes inside each neurons from a **logistic regression** model and a **single hidden layer neural network model**. As mentioned in the beginning, we can learn any x to y mapping function given enough number of neurons. Therefore, **deep networks** with multiple **hidden layers** can learn functions that the shallow ones cannot. The **hidden layer 2** will take $A^{[1]}$ in and output $A^{[2]}$, and pass it onto the **hidden layer 3**. Repeating the process until the output layer is reached.The deeper the layer is the higher level features it learns. If we have a face recognizer, the first layer will figure out how to detect edges, the next layer will learn how to group those edges to detect eyes, ears, nose, and etc. And the later layer will learn how to group those informations to recognize different faces. Let's build a **deep neural network**.



Learning rate, number of epochs (number of iterations), number of hidden units, number of hidden layers, choice of activation functions, etc. are **hyper-parameters** that control the character of **parameters** W and b. We can build some insights but the only way to figure out good **hyper-parameters** is by experimenting.