Week3 - Neural Network

Freitag, 17. Juli 2020 14:21

Standard Notation:

Standard notations for Deep Learning

This document has the purpose of discussing a new standard for deep learning mathematical notations.

1 Neural Networks Notations.

General comments:

- superscript (i) will denote the i^{th} training example while superscript [l] will denote the l^{th} layer

Sizes:

- m : number of examples in the dataset
- n_x : input size
- n_y : output size (or number of classes)
- $n_h^{[l]}$: number of hidden units of the l^{th} layer

In a for loop, it is possible to denote $n_x = n_h^{[0]}$ and $n_y = n_h^{[\text{number of layers }+1]}$.

- L : number of layers in the network.

Objects:

- $X \in \mathbb{R}^{n_x \times m}$ is the input matrix
- $x^{(i)} \in \mathbb{R}^{n_x}$ is the i^{th}example represented as a column vector
- $Y \in \mathbb{R}^{n_y \times m}$ is the label matrix
- $y^{(i)} \in \mathbb{R}^{n_y}$ is the output label for the i^{th} example
- $W^{[l]} \in \mathbb{R}^{\text{number of units in next layer} \times \text{number of units in the previous layer}}$ is the weight matrix,superscript [l] indicates the layer
- $b^{[l]} \in \mathbb{R}^{\text{number of units in next layer}}$ is the bias vector in the l^{th} layer
- $\hat{y} \in \mathbb{R}^{n_y}$ is the predicted output vector. It can also be denoted $a^{[L]}$ where L is the number of layers in the network.

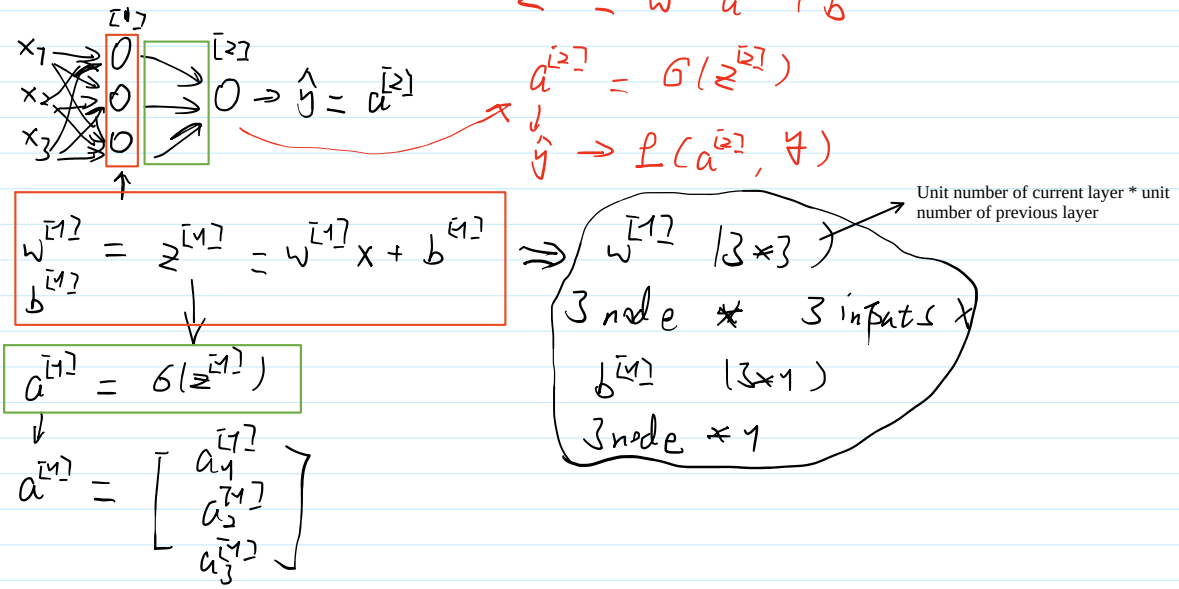
Common forward propagation equation examples:

- $a = g^{[l]}(W_x x^{(i)} + b_1) = g^{[l]}(z_1)$ where $g^{[l]}$ denotes the l^{th} layer activation function
- $\hat{y}^{(i)} = softmax(W_h h + b_2)$
- General Activation Formula: $a_j^{[l]} = g^{[l]}(\sum_k w_{jk}^{[l]} a_k^{[l-1]} + b_j^{[l]}) = g^{[l]}(z_j^{[l]})$
- $J(x, W, b, y)$ or $J(\hat{y}, y)$ denote the cost function.

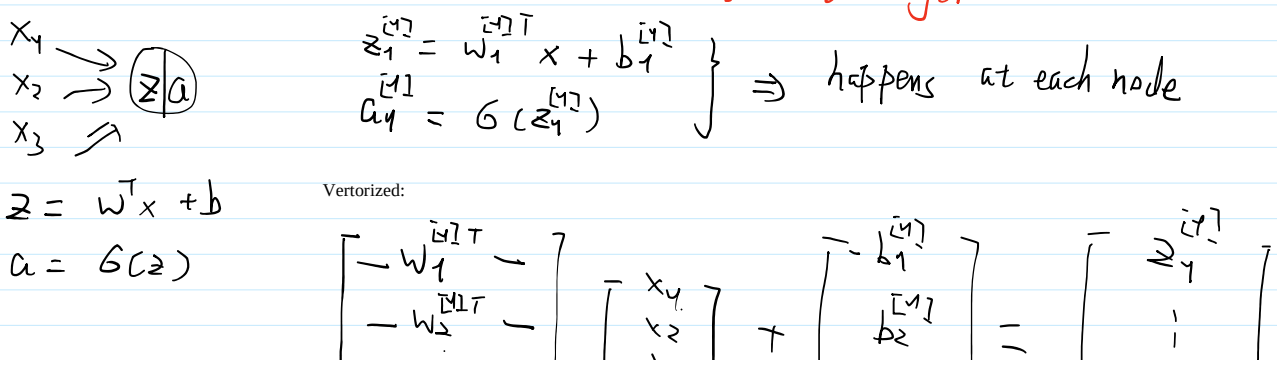
Examples of cost function:

- $J_{CE}(\hat{y}, y) = -\sum_{i=0}^{m} y^{(i)} \log \hat{y}^{(i)}$
- $J_1(\hat{y}, y) = \sum_{i=0}^{m} | y^{(i)} - \hat{y}^{(i)} |$

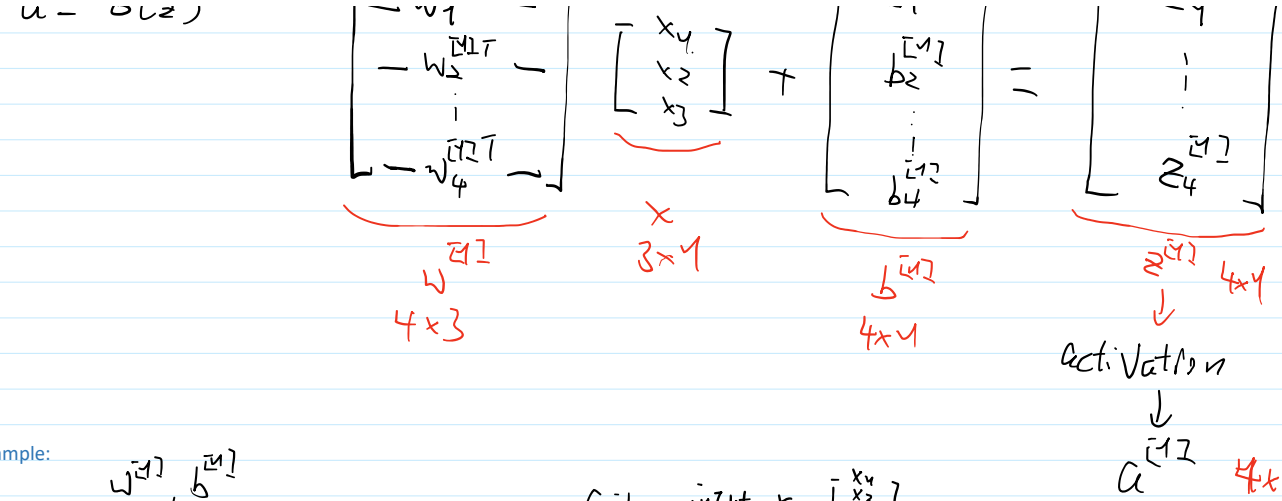
Structure of NN with one hidden layer (shallow NN)



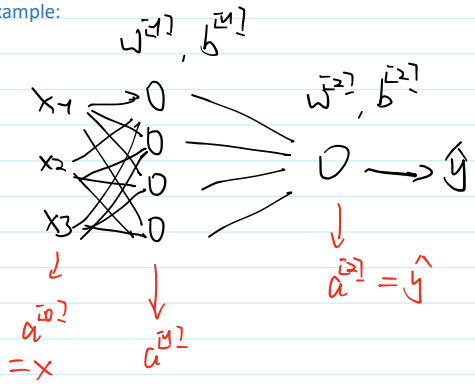
What happens in one node / Mathematical representation of one node:



Vertorized:



Example:



Given input x $\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$

$$\underset{4*1}{z^{[1]}} = \underset{4*3}{W^{[1]}} \underset{3*1}{x} + \underset{4*1}{b^{[1]}}$$

$$\underset{4*1}{a^{[1]}} = \sigma(z^{[1]})$$

$$\underset{1*1}{z^{[2]}} = \underset{1*1}{W^{[2]}} \underset{4*1}{a^{[1]}} + \underset{1*1}{b^{[2]}}$$

$$a^{[2]} = \underset{1*1}{\hat{y}} = \sigma(z^{[2]})$$

For m training examples

$x^{(1)} \rightarrow a^{[2](1)}$

$x^{(2)} \rightarrow a^{[2](2)}$

$x^{(3)} \rightarrow a^{[2](3)}$

$x^{(m)} \rightarrow a^{[2](m)}$

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

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

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

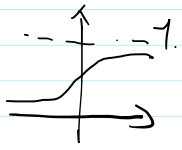
units

m examples

Different Activation Functions:

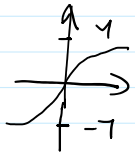
1. Sigmoid

$$\frac{1}{1+e^{-z}}$$



2. Tanh

$$g = \tanh(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{z}}$$



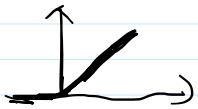
Mean of tanh == 0 and the dataset has usually also mean of 0 (standardlized), increases the process; However the output layer has to be sigmoid because (0,1)

Yet, one problem for tanh(z):
When z -> infinite, gradient really small which slow down grandient descent.

3. ReLU

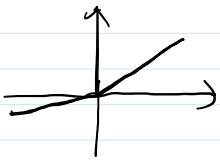
3. ReLU

$a = \max(0, z)$



4. Lecky ReLU

$a = \max(0.01z, z)$



Default setting:

For output layer: sigmoid

For hidden layer: ReLU

Gradient of different activation function:

Sigmoid:

$$g'(z) = g(z)(1-g(z)) = a(1-a)$$

Tanh:

$$g'(z) = 1-(\tanh(z))^2 = 1-a^2$$

ReLU

$$g'(z) = \begin{cases} 0 & z<0 \\ 1 & z \geq 0 \end{cases}$$

Lecky ReLU:

$$g'(z) = \begin{cases} 0.01 & z<0 \\ 1 & z \geq 0 \end{cases}$$

Implementation of 2 layer NN:

Parameters: $w^{[1]}$ $b^{[1]}$ $w^{[2]}$ $b^{[2]}$ $(n^{[1]}, n^{[0]})$ $(n^{[1]}, 1)$ $(n^{[2]}, n^{[1]})$ $(n^{[2]}, 1)$

- $n_x = n^{[0]}$ input
- $n^{[1]}$ - number of hidden units
- $n^{[2]}$ - number of output units

cost: $J(w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]}) = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}, y)$

Repeat i iterations:

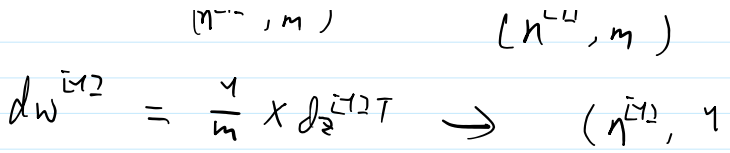
compute prediction $\hat{y} = A^{[2]}$

$dz^{[2]} = A^{[2]} - y \rightarrow (n^{[2]}, m)$

$dw^{[2]} = \frac{1}{m} A^{[1]} dz^{[2]T} \rightarrow (n^{[2]}, n^{[1]})$

$db^{[2]} = \frac{1}{m}$ np.sum$(dz^{[2]}, axis=1, keepdim=True) \rightarrow (n^{[2]}, 1)$

$dz^{[1]} = \underbrace{w^{[2]T} dz^{[2]}}_{(n^{[1]}, m)} * \underbrace{g^{[1]'}(z^{[1]})}_{(n^{[1]}, m)} \rightarrow (n^{[1]}, m)$



How to implement NN

Random Initionlization

Take out from the programming assignment:

The general guidence to build a NN:

1. Define the neural network structure (# of input units, # of hidden units, etc).
2. Initialize the model's parameters
3. Loop:
   - Implement forward propagation
   - Compute cost
   - Implement backward propagation to get the gradients
   - Update parameters (gradient descent)

After all these, the model can be used to predict.

Usually build helper functions to compute and then merge them into one function / nn_model(). Once you've built nn_model() and learnt the right parameters, you can make predictions on new data calling the nn_model().

During the implementation, i personally think paying attention to the demensions of each matrix is extremly helpful.

Last but not least, the design of hidden layer size was observed with different numbers of units in the hidden layer: 1,2,3,4,5,20,50. The result shows that more than 5 units in hidden layer doesnt helps to increase the accuracy, because the model is overfitted to the training set, which wont perform well on the other datasets.