Coursera Deep Learning Specialization Notes: Neural Networks and Deep Learning

Amir Masoud Sefidian

Version 1.0, November 2022

Contents

Preface

A couple of years ago I completed Deep Learning Specialization taught by AI pioneer Andrew Ng. I found this series of courses immensely helpful in my learning journey of deep learning. After years, I decided to prepare this document to share some of the notes which highlight key concepts I learned in the first course of this specialization, Neural Networks and Deep Learning. This course teaches you the theory behind deep learning and applies this technology in the real world, such as creating and training a simple neural network and understanding the key parameters of deep learning. Notes are based on lecture videos and supplementary material provided and my own understanding of the topics.

The content of this document is mainly adapted from this GitHub repository. I have added some explanations, illustrations, and visualization to make some complex concepts easier to grasp for readers. This document could be a good reference for Machine Learning Engineers, Deep Learning Engineers, and Data Scientists to refresh their minds on the fundamentals of deep learning. Please don't hesitate to contact me via my website (sefidian.com) if you have any questions.

Happy Learning!
Amir Masoud Sefidian

1 Neural Networks and Deep Learning

1.1 Logistic regression as a neural network

- A small-scale example of a neural network with a single neuron (and still useful as a classifier).
- n: number of features, m: number of samples
- Inputs: $X = \begin{bmatrix} x^{(1)} & x^{(2)} & \cdots & x^{(m)} \end{bmatrix}_{n \times m}$, $Y = \begin{bmatrix} y^{(1)} & y^{(2)} & \cdots & y^{(m)} \end{bmatrix}_{1 \times m}$
- Logistic Loss / Binary Cross Entropy Loss:
  $L(y, a) = -[y \log a + (1 - y) \log(1 - a)]$
- Vectorized Optimization:
  $Z = W^T X + b$

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

  $dZ = A - Y$

  $dW = \frac{1}{m} X dZ^T$

  $db = \frac{1}{m} \sum dZ$

  $W \leftarrow W - \alpha \cdot dW, b \leftarrow b - \alpha \cdot db$
- **Note:** d stands for "derivative" where there is only a single variable. ∂ stands for "partial derivative" where there are multiple variables.

1.2 Neural networks

- Typically the input layer is not counted as a layer when counting the layers of a neural network, only the hidden layers and the output layer are considered. The output layer is NOT a hidden layer.
- $A^{[l]}$ is the activation for layer l, $W^{[l]}$ is the weights matrix, and $b^{[l]}$ is the bias term. $g(z)$ is the chosen activation function.
- The learning rate α scales the size of the gradient descent steps, and therefore determines how fast it converges (though values too high can actually make it not converge).
- Regarding notation $a_j^{[l](i)\{k\}}$ means the activation (could be any other parameter) on the j^{th} unit/neuron of the l^{th} network layer, for the i^{th} example/data point of the k^{th} minibatch.
- Notations:
  L: # of layers
  $n^{[l]}$: # of units in layer l
  $b^{[l]}$: biases in layer l (size: $(n^{[l]}, m)$)
  $Z^{[l]}$: outputs in layer l (size: $(n^{[l]}, m)$)
  $A^{[l]}$: activations in layer l (size: $(n^{[l]}, m)$)
  $W^{[l]}$: Weights connecting layer $l-1$ to l (size: $(n^{[l]}, n^{[l-1]})$)
  $n^{[0]} = n_X$: # of input features

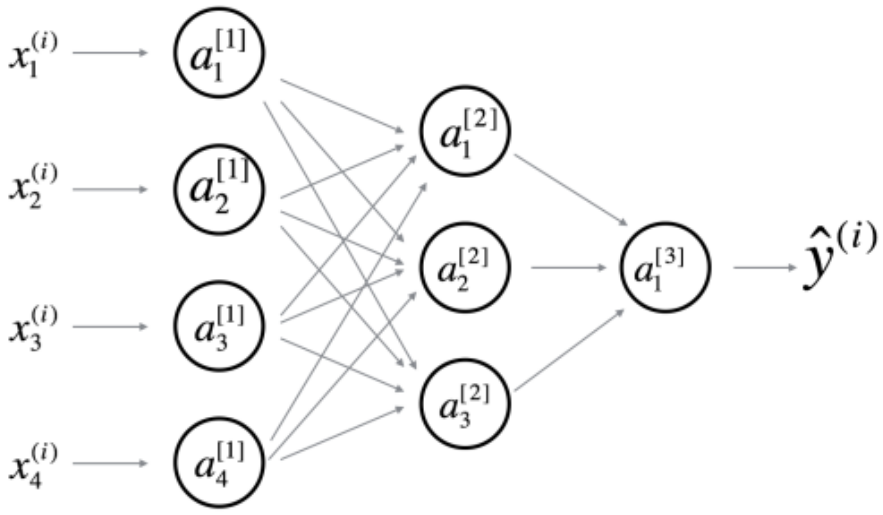

Figure 1:

Forward Propagation for layer l (vectorized):

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

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

Backward Propagation for layer l (vectorized):

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

$$dW^{[l]} = \frac{1}{m} dZ^{[l]}(A^{[l-1]})^\intercal$$

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

$$dA^{[l-1]} = (W^{[l]})^\intercal dZ^{[l]}$$

Note: $*$ is the element-wise multiplication

Parameter Update for layer l (vectorized):

$W^{[l]} \leftarrow W^{[l]} - \alpha \cdot dW^{[l]}$, where α is the learning rate

$b^{[l]} \leftarrow b^{[l]} - \alpha \cdot db^{[l]}$, where α is the learning rate

1.3 Activation functions

- Sigmoid (or logistic function) - usually used for the output layer only (because it outputs either 1 or 0), and not for the hidden layers because its derivative can be close to 1 for values of z further away from the origin.

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

- *tanh* is better than sigmoid for hidden layers (though it is an offset sigmoid) only because it is defined between -1 and 1, which means that for the mean of zero, the tanh will be close to 0 as well, which can be useful for computation.

$$tanh(z) = \frac{2}{1+e^{-2z}} - 1,\ tanh'(z) = 1 - tanh(z)^2$$

- Rectified linear unit (ReLU) - $ReLU(z) = max(0, z)$ - is the de facto standard for linear units nowadays, its derivative is easy to calculate (not defined for z = 0, but we can work around that with a convention on what 0 means), and does not suffer from slow convergence of gradient descent due to derivative being close to 0.

$$ReLU(z) = \begin{cases} 0 & \text{if } z < 0 \\ z & \text{if } z \geq 0 \end{cases},\ ReLU'(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases}$$

- Leaky ReLU - $max(0.01 * z, z)$ - Similar to the ReLU, but instead of being zero for values of z lower than 0, it actually has a small positive slope in that section, so its derivative is not 0, making it easier for the optimizer (e.g. gradient descent) to converge. In reality normal RelU are still the standard though.

$$LReLU(z) = \begin{cases} 0.01z & \text{if } z < 0 \\ z & \text{if } z \geq 0 \end{cases},\ LReLU'(z) = \begin{cases} 0.01 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases}$$

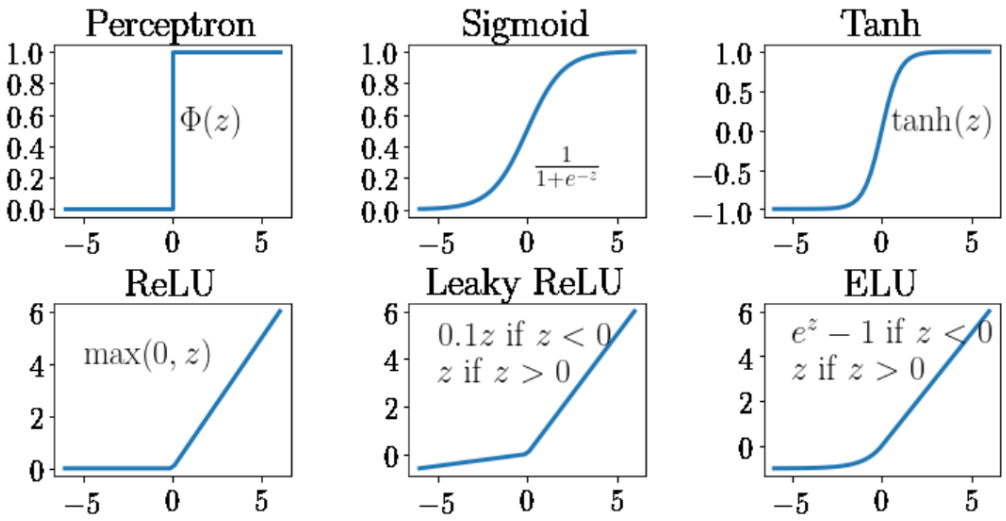

Figure 2: Activation Functions

Why do we need some sort of activation function anyway?

- If we had none, the activation would be linear, and all layers would have linear activation functions, resulting in the NN activation being itself a linear activation function, which negates the usefulness of the hidden layers. There must always be a hidden layer.

1.4 Random initialization

- Two neurons are considered symmetric if they are doing the exact same computation.
- We avoid that by using random initialization rather than zero initialization, otherwise all neurons/units will keep on being symmetric throughout all backprop iterations.
- b (bias) doesn't require random initialization like the weights W.
- W should be initialized to small random values (multiply by 0.01 for example), to allow faster convergence of gradient descent with sigmoid or $tanh$ activation functions.
