COL341

Assignment 2- Neural Networks

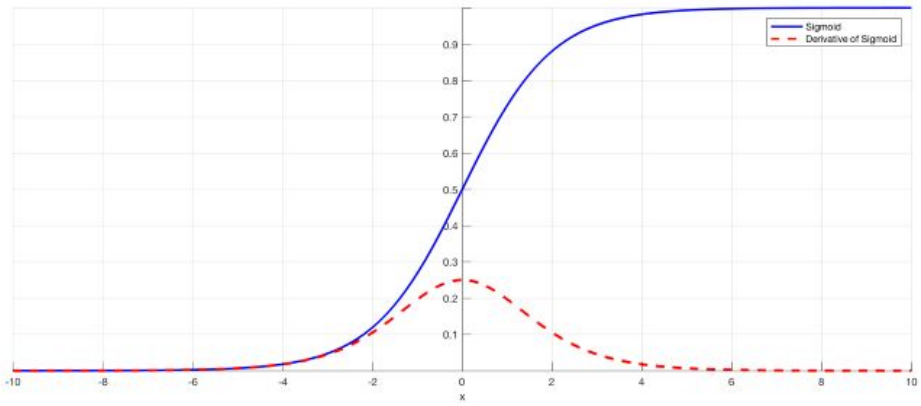
Prabhat Kanaujia
2016CS50789

PART B: Observations

1. The performance hugely depends on the values to which the weights are initialized. This led to a difference of upto **4%** accuracy at times.

2. Performance varies on the data in the following order(lowest to highest):
    a. Raw data
    b. Normalized data
    c. Standardized data

This might be due to the fact that the activation functions we use, i.e. **sigmoid** and **tanh** are symmetric about the y-axis(x = 0) line(and so are their derivatives) and hence constraining the data to behave similarly might give best performance for these kind of activation functions.



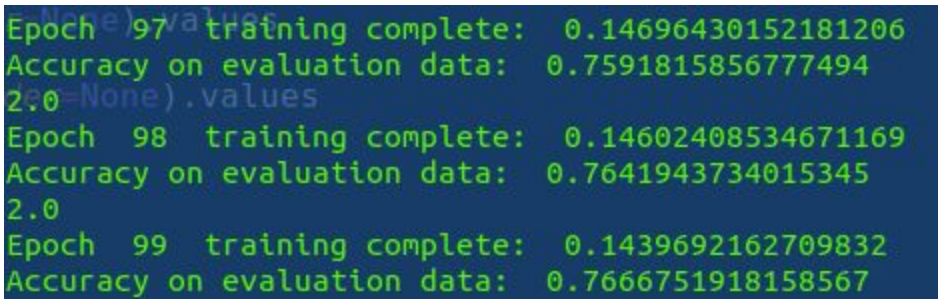
3. In my case, increasing the number of units in a hidden layer, while keeping the number of layers same, didn't give much improvement in performance. Initially, the decrease in cost per epoch was a bit higher but it slowed down towards the end and eventually performed a bit worse than the architecture with half the number of hidden units.
    a. 100 hidden units -> **0.46** (accuracy, not cost)
    b. 200 hidden units -> **0.45**

4. Increasing the number of hidden layers gave a comparatively better performance and led to a faster convergence.
    a. 1 layer, 100 units -> **0.46**
    b. 2 layers, 100 units each -> **0.50**

5. An interesting fact to note is that when the number of units are increased in a hidden layer, **time taken per epoch** increases proportionally. But, if another layer having the same number of units is added to the neural network, then there is negligible change in the time taken. This is because when a new layer is added, the number of matrix operations are just multiplied by a factor **k**(=units in existing layer/units in new layer), considering 1 hidden layer to start with. But when the same number of units are added to the existing layer, the factor shoots up exponentially, with **k** as the base.

6. Using a smaller **batch size** gives slightly better performance because the weights are updated more frequently.

Overall, my best performing architecture is:
Hidden layers: [100,100]
Learning rate: 2
Batch size: 128
Activation function: sigmoid
Epochs: 100
I got an accuracy of **76.66%** using this architecture, for a particular random initialization of weights.



This code runs in 24 minutes on my laptop, 7th gen i5, 8 GB RAM.
I plan on limiting the epochs to 90, as a safety factor.

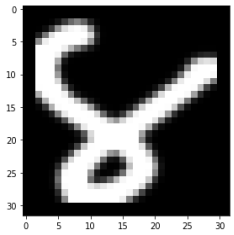
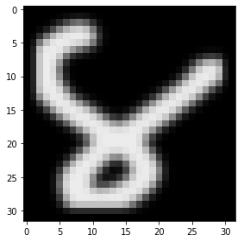
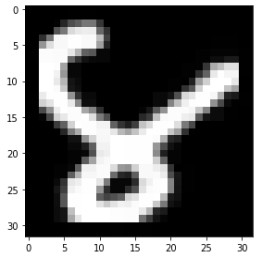
Note: I notices that the actual image is only 28*28 and the rest is padding. It was taught in class that padding is needed for convolution and since we aren't doing any convolution, i chose to remove the padding, thus reducing the dimensionality to 28*28 = 784.

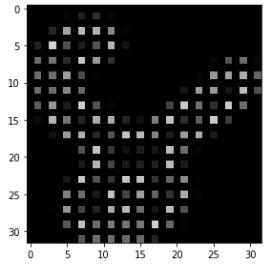
PART C: Observations

1. **Tanh** behaves somewhat similar to sigmoid, but the accuracy always lacked behind sigmoid by a small amount.

2. **ReLU** performs well when the learning rate is low. If the learning rate is high, many weights are updated such that some units have 0 values. These contribute neither to the next layer, nor to the derivative and are called **dead ReLUs.**

3. Feature engineering techniques I tried are:

    - Original Image:
    - 
    - Gabor Filter:
    - 
    - Bayesian Noise elimination:
    - 
    - HOG

- 

  Out of these filters, Gabor gave the best performance.
  Bayesian Noise elimination didn't have any benefit and the results were almost the same.
  HOG improved the performance, but only a little bit.

Hence, I decided to go ahead with **Gabor filter.**

References:

- http://scikit-image.org/docs/dev/api/skimage.filters.html
- https://en.wikipedia.org/wiki/Backpropagation
- http://neuralnetworksanddeeplearning.com/chap1.html
- https://towardsdatascience.com/difference-between-batch-gradient-descent-and-stochastic-gradient-descent-1187f1291aa1
- https://medium.com/datathings/vectorized-implementation-of-back-propagation-1011884df84
- https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/