CS70 Final Report:

Least Squares and Neural Network Approaches to Handwritten Digit Classification

Joshua Pfefferkorn

CS70: Foundations of Applied CS

Prof. Bo Zhu

Presentation Video Link:
https://dartmouth.hosted.panopto.com/Panopto/Pages/Viewer.aspx?id=509b3cac-36fc-4021-af41-ad40012ea2ab

Methodology – Appetizer Task

For the appetizer task, we were asked to solve the handwritten digit classification problem using least squares. To accomplish this, I set up the normal equations for 10 separate least squares problems using the 1000 x 785 data matrix X and a column of the true-value matrix Y at a time to find each column of the solution matrix theta, whose maximum value's index indicated the model's guess for the digit's classification.

Methodology – Entrée Task

For the entrée task, we were asked to solve the same classification problem using a neural network approach. For each layer, I implemented two functions: one of forward propagation, and a second of backward propagation. Here, I designed three nonlinear activation layers of my own for a dessert task – a LReLU (Leaky Rectified Linear Unit) layer, FReLU (Flexible Rectified Linear Unit) layer, and a sigmoid function layer. Ultimately, I assembled the objects for linear and nonlinear layers into a network builder, which called the appropriate constructors to build the network framework for an inputted list. In terms of the classification itself, I wrote a one-hot encoder function and modeled a gradient descent algorithm after some sample code given.

Code Implementation – Appetizer Task

In function "classifier":

- Add noise to the training data by looping over the data matrix and adding a random number from 0 to 0.0001 to each entry
- Build Y using one-hot encoding (i.e., making the entry in the appropriate column 1 as indicated in the training label matrix and leaving the other entries as -1)
- Solve 10 least squares problems using X and a column of Y to set up normal equations and find a column of theta

Code Implementation – Entrée Task

In class "LinearLayer"

- Function "forward"
  - Store input
  - Multiply weights by input and return the product to forward data to the next layer
- Function "backward"
  - Store gradient of the weights by multiplying stored input by gradient of the output
  - Return the product of the gradient of the output and the weights

In class "ReLU"

- Function “forward”
    - Perform ReLU function on each entry in the input
    - Store input
    - Return output
- Function “backward”
    - Perform derivative of ReLU function on each entry in the stored input
    - Return the product of this modified input and the gradient of the output

In class “LReLU”

- Function “forward”
    - Perform LReLU function on each entry in the input
    - Store input
    - Return output
- Function “backward”
    - Perform derivative of LReLU function on each entry in the stored input
    - Return the product of this modified input and the gradient of the output

In class “FReLU”

- Function “forward”
    - Perform FReLU function on each entry in the input
    - Store input
    - Return output
- Function “backward”
    - Perform derivative of FReLU function on each entry in the stored input
    - Return the product of this modified input and the gradient of the output

In class “sigmoid”

- Function “forward”
    - Perform sigmoid function on each entry in the input
    - Store input
    - Return output
- Function “backward”
    - Perform derivative of sigmoid function on each entry in the stored input
    - Return the product of this modified input and the gradient of the output

In class “MSELoss”

- Function “forward”
    - Store difference between prediction and truth values
    - Loop over the matrix containing this difference and square each value, cumulatively adding the result to a variable tracking total loss
    - Divide the loss by the number of elements to get mean squared loss
- Function “backward”
    - Perform derivative of loss function on the stored differences

In class “Network”

- Function “__init__”

    - Loop over list of layer types and parse each message, calling the appropriate constructor to add the correct layer type to a list of layers
- Function “forward”
    - Loop over list of layers and call the forward function on the current layer
- Function “backward”
    - Loop over list of layers in reverse and call the backward function on the current layer

In function “One_Hot_Encode”
- Create a labels size x classes size matrix of 0s
- Change the appropriate entry in each row to 1 according to the list of data labels

In class “Classifier”
- In function “Train_One_Epoch”
    - Forward the data to the network, getting a prediction
    - Use the prediction to calculate loss using loss function
    - Propagate the gradient backwards through the network
    - Update the weights with the weight gradients, multiplying by the learning rate

Results Discussion – Appetizer Task

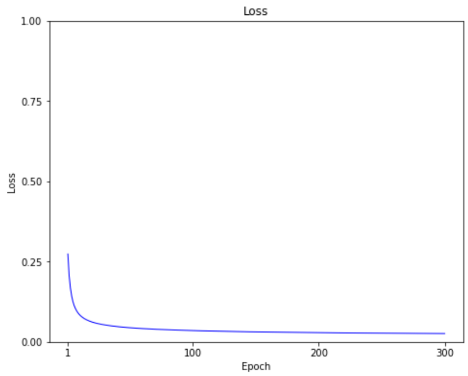
```
Training accuracy is: 0.765
Testing accuracy is: 0.43
```

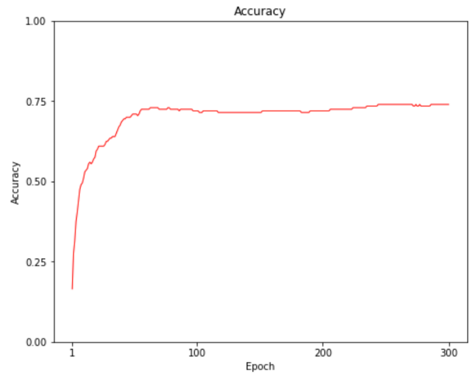
- Accuracy was about 43%, which indicates that the least squares solver was functioning properly but reveals that a more intuitive system (like a neural network) might be a more apt implementation to solve this classification problem

Results Discussion – Entrée Task

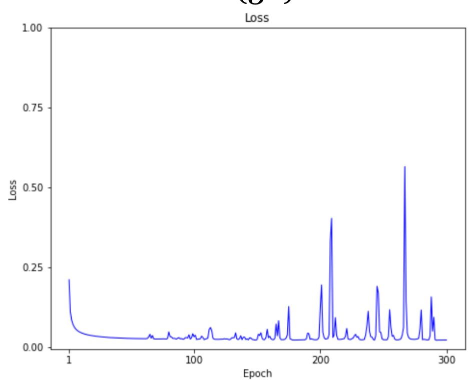
- Accuracy with various parameters and network architectures ranged from about 70% to 80% accuracy
    - I found that optimal parameters for my implementation were around a *learning rate of 0.01, a batch size of 256, and a max epoch of 300*
        - These are the parameters used for the below figures unless otherwise specified
- My dessert nonlinear activation functions in place of ReLU and my new network architectures proved to be marginally more accurate than the given architecture
- Overall, the neural network approach was about twice as accurate as the least squares solution
- The following figures display the loss and accuracy convergence on each of the 7 approaches I used, varying architecture, layer types, and layer order

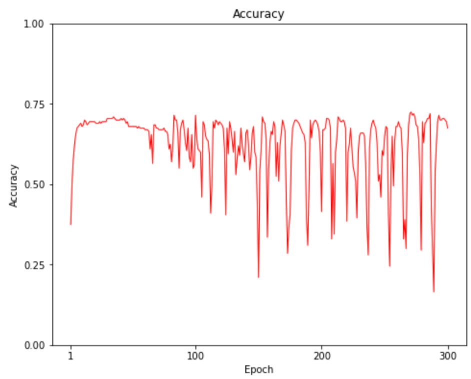
[Linear -> ReLU -> Linear]



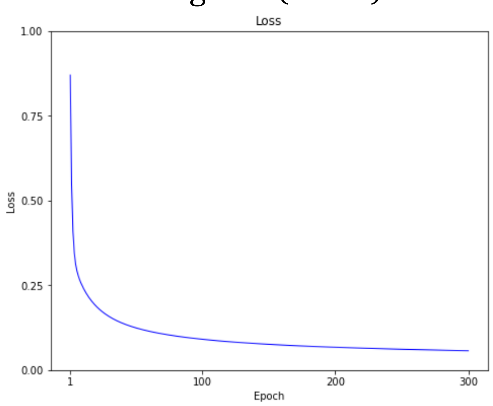


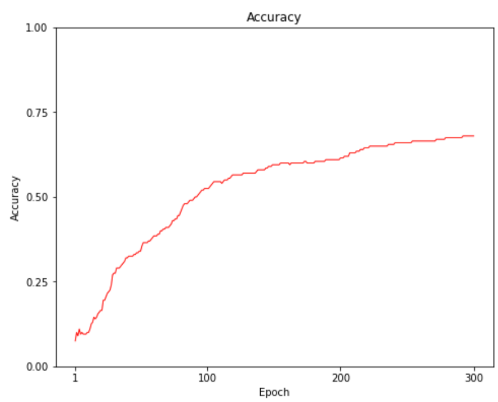
Small batch size (32)



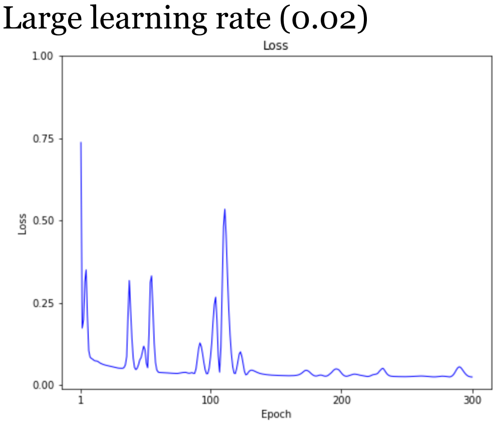


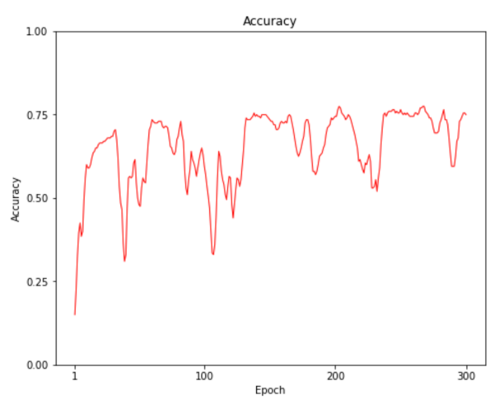
Small learning rate (0.001)



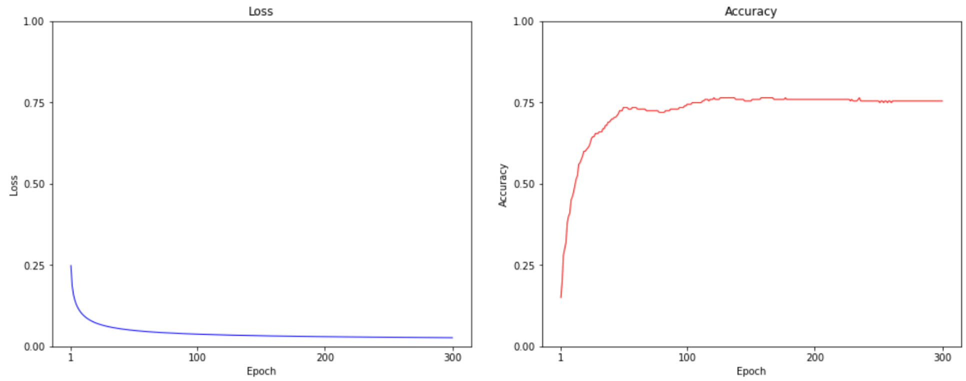


Large learning rate (0.02)

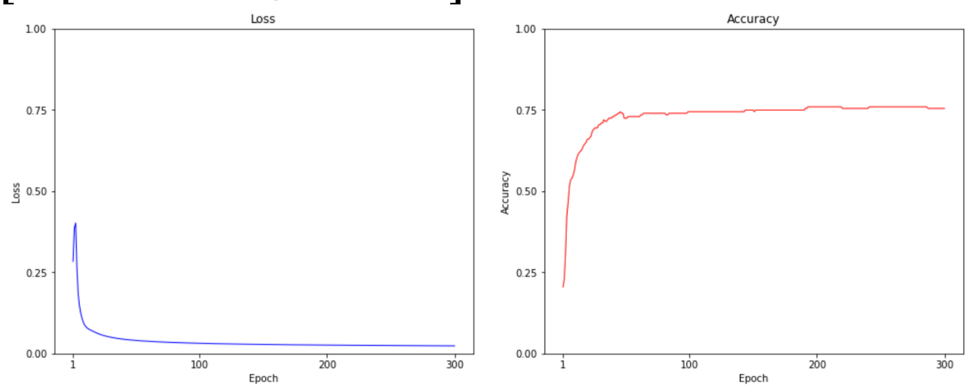




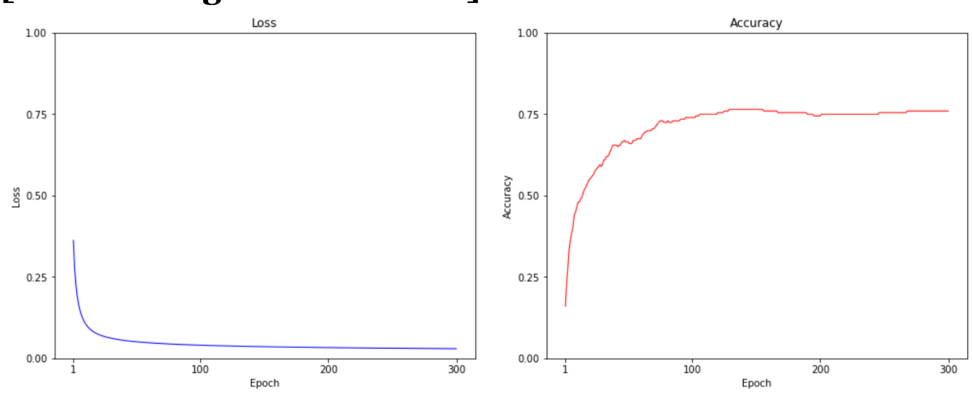
[Linear -> LReLU -> Linear]



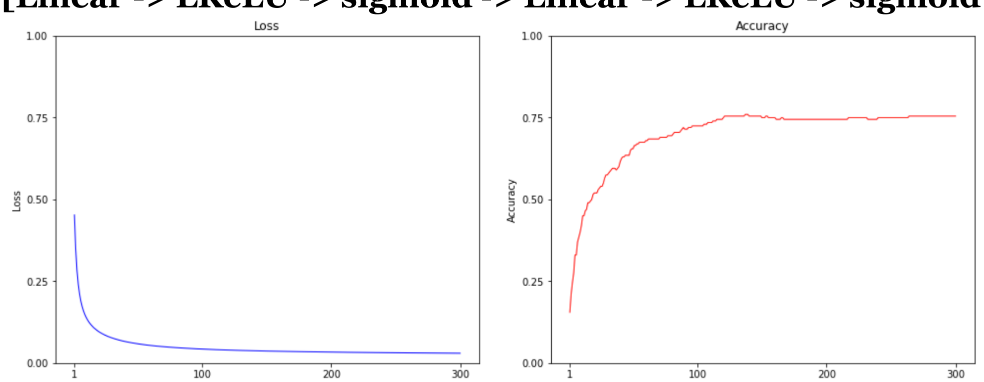
[Linear -> FReLU -> Linear]



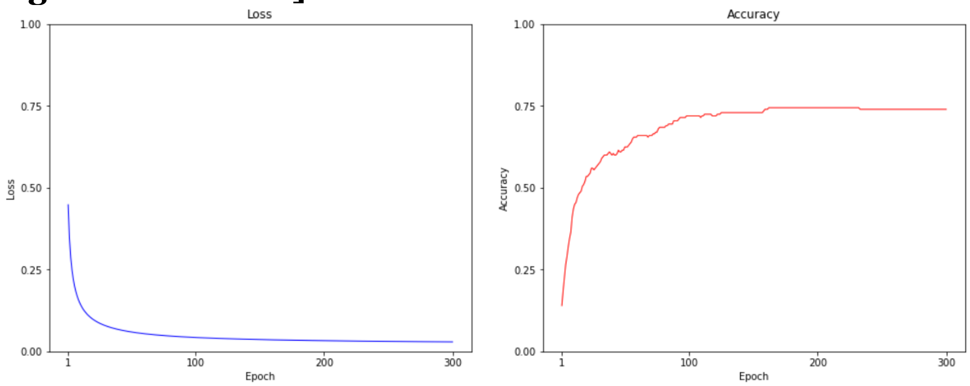
[Linear -> sigmoid -> Linear]



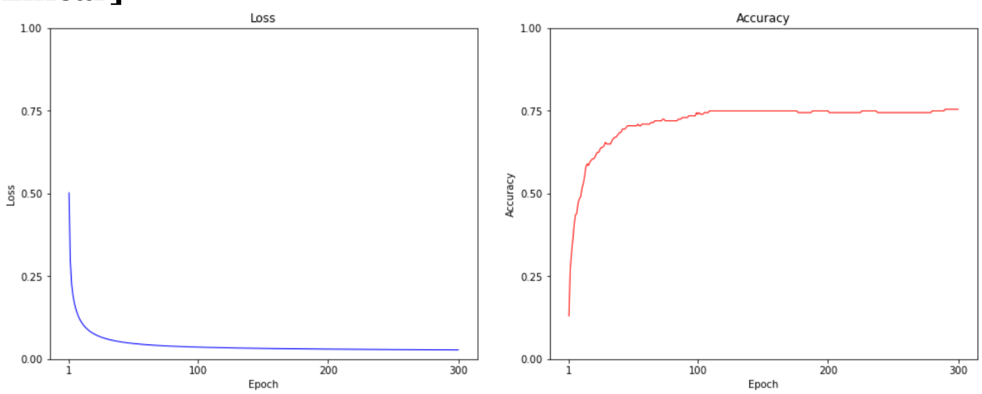
[Linear -> LReLU -> sigmoid -> Linear -> LReLU -> sigmoid -> Linear]



[Linear -> sigmoid -> LReLU -> sigmoid -> Linear -> sigmoid ->LReLU -> sigmoid -> Linear]



[Linear -> FReLU -> sigmoid -> Linear -> Linear -> FReLU -> sigmoid -> Linear]



Main Challenges

I found the code in this project to be fairly straightforward. For me, the main challenge came in understanding how different parameter tweaks applied to the results – accuracy and loss. I found fine-tuning the parameters to be a tedious process of trial-and-error. I also found that experimenting with different network architectures was largely a guess-and-check process to discover which frameworks offered the best results consistently. Ultimately, I was pleased with the accuracy I achieved but not confident I understood conceptually why certain activation and linear layers in certain orders worked better than other combinations.