Report 0: Project 0

By: TJ DiMeola
Course: Computer Vision CSCI 581
Instructor: Hawk (Dr. Wang)
Date: September 07, 2025

Purpose and Objectives

We are to build (write code for) three significant algorithms in the CV field:

- Linear neural networks for classification using the softmax function
- Multilayer perceptrons (MLPs)
- Basic convolutional neural networks, through the implementation of LeNet

It is expected that we will write this code without relying on the textbook architectural elements provided by d2l, but to attempt to complete the assignments such at we gain a much deeper understanding of the the historical effort that has gone (and still does go) into basic image/object identification.

2. Methods and Approach

I have chosen to carry out this project at the lowest possible coding level, in order to ensure that I understand each element that goes into a given architecture, as well as possible. For this reason, I have not only no relied on d2l methods, I have also not relied on the Torch nn class (with a single exception, which will be detailed below).

2.1 The Data

For all exercises, the Fashion-MNIST[1] dataset was used. The dataset includes a training set of 60,000 examples and a test set of 10,000 examples. Each example is a 28x28 grayscale image, associated with a label from 10 classes. Fashion-MNIST shares the same image size and structure for training and testing as MNIST[2].

2.2 Code Design

It was my decision to try first to flatten *all* the code to ensure that I captured both every small nuance as well as to be able to somehow track the evolutionary changes from linear to MLP to NN. In order to do this, I started with a general outline of the elements that go into generic code for CV. This is shown in Figure 1.

1. Load data.
2. define the model forward pass
   a) softmax
   b) forward
3. define Loss Function
   a) cross_entropy
4. define metric
   a) accuracy, mIoU, etc.
5. define training Loop
6. define validation Loop
7. define main workflow
   a) define hyperparameters
      1. batch_size = 256
      2. num_inputs = 28 * 28
      3. num_outputs = 10
      4. lr = 0.1
      5. epochs = 10
   b) call load function
      1. return train and val
         1. train_loader
         2. val_loader
   c) define weights and bias (use torch.normal, torch.zeros respectivel)
   d) run through epochs loop
      1. train
      2. val

Figure 1: Basic outline for building linear, MLP, and neural networks. Note that the hyperparamter values will vary, but these were the basic values for these exercises.

I devised this outline as I worked my way through the linear (SoftMax) classification implementation. It helped me a great deal for both MLP and NN.

2.3 Tools and Resources Used

I was able to write the softmax classifier on my own by following the text in d2l. The same was not true for the MLP.

I started by leveraging the linear classifier, but got into trouble when I had to pass forward W2, b2. So I did turn to GPT4.1. That was a pretty easy fix (just more of the the same).

When I went to write the code for LeNet I now passed *all* the hyperparameters, however, I got hopelessly confused by theie initializations and kept running the back propagation multiple times! So I did turn to Claude[3] in this instance. A record of this conversation is available as a link in my code for LeNet.

1 https://github.com/zalandoresearch/fashion-mnist?tab=readme-ov-file

2 https://huggingface.co/datasets/ylecun/mnist

In the end, I did create a simplified (using Torch.nn) version of LeNet since my looped version was taking almost an hour to get through a single epoch. This version is included with the code.

All code is included in the .zip folder.

2.4 SoftMax Regression/Classifier

2.4.2 Prepare FashionMNIST Dataset

See *FashionMNIST.py in code.*

Question 1*:* Visualize 10 images from the dataset.

Result shown in Figure 2 below.

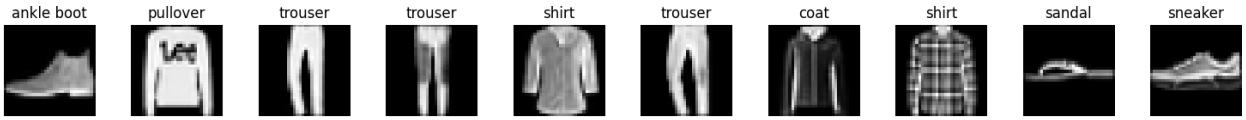
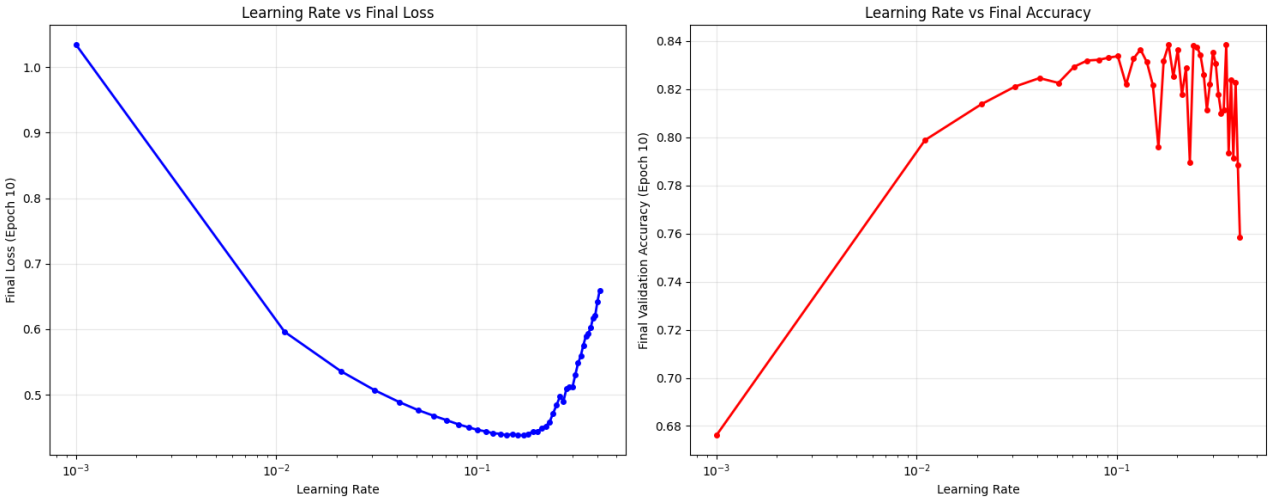


Figure 2: 10 images from the FashionMNIST dataset.

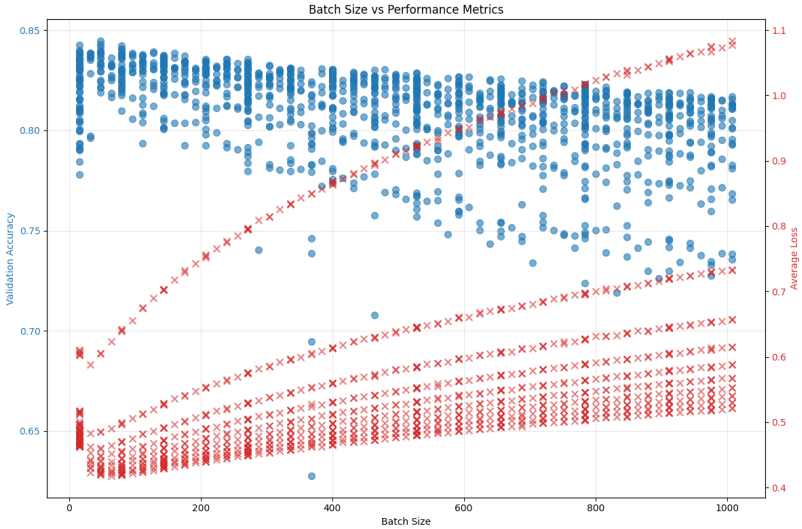
2.4.3 Question 2: Hyperparameter analysis

Original results (lr = .10, Batch_Size = 256): Epoch 10/10: Loss: 0.4472, Val Accuracy: 0.8316

2.1 Try to use different learning rates, plot how validation loss changes.



2.2 Try to use different batch sizes, how validation and training loss changes?



3 claude.ai

2.4.2 Question 3: Explain the overflow and underflow problems in softmax computation

If input values are very large and positive, the term the softmax term can exceed the maximum value that can be represented by a floating-point number resulting in NAN. If the input values are very large and negative, the softmax term can become so small that it gets rounded down to zero, and a division by zero error occurs. This is an underflow.

One possible solution to this problem is to subtract o_bar from all entries, explain how this potentially can solve the overflow problem. Is this solution perfect or you think this solution may cause some other problems (e.g., NaN)?

This solution helps solve the overflow problem because because o_bar is the maximum value, so an element subtracted from it will be less than or equal to it. Since exp(0) = 1, the overflow cannot occur.

This solution is not perfect because if the terms o_bar and the o_j elements are all negative (and thus, very small), then the denominator might be so small that it would be interpreted as zero, leading to a division by zero anyway (underflow).

2.5 Multilayer Perceptron

2.5.1 Hidden layers and activation functions

Question 1: Come up with an example where the gradients vanish for the sigmoid activation function

Answer: If the initial weight (selected randomly – randn) starts small then, as the weights traverse the network, if they get smaller then during backpropagation, when the derivatives are taken, these values will become vanishingly small.

2.5.2 Implementation

Code: *tjd_mlp.py*

Question 2: Give it a try to add one hidden layer, show how it affects the results. What if adding a hidden layer with one single neuron, what's the results? Please discuss what you get and why.

Answer: ***Not Complete***

2.5.3 Forward, backward propagation, and computational graphs

Question 3: What's the memory footprint for training and prediction in the model described in the example shown in section 5.3?

The size of intermediate values is roughly proportional to the number of network layers and the batch size. Therefore for the model given (a one-hidden-layer MLP)

Number of parameters x bytes per parameter = memory of parameters
And the bytes per parameter depend on the data type, which we have specified as ssingle-precision float or 4 bytes per parameter. Input: 784 features, Hidden layer: 256 neurons, Output: (10 classes) .

Parameters:

W1 (input to hidden): 784 × 256 = 200,704 weights **b1 (hidden layer bias):** 256 biases

W2 (hidden to output): 256 × 10 = 2,560 weights
b2 (output layer bias): 10 biases

Total: 200,704 + 256 + 2,560 + 10 = **203,530 parameters**

at 4 bytes per parameter = 814120 bytes memory

Loss and Accuracy

- Epoch 10: Val accuracy: 0.849

2.6 Basic Convolutional Neural Networks (LeNet)

2.6.1 Implementation

Code (flattened, unvectorized): *tjd_LeNet_flattened.py*

Loss and Accuracy

- Using Sigmoid activation: Epoch 10/10: Loss: 0.7538, Val Accuracy: 0.7022 (AvgPool)

Question 1: Use different epochs (5, 10, 20) to train your LeNet, plot the training loss, validation loss, and validation accuracy of each training process, and discuss how changing training epochs affects the performance.

- 5 epochs, sigmoid, avgpool: Epoch 5/5: Loss: 1.8875, Val Accuracy: 0.4875
- 20 epochs, sigmoid, avgpool: Epoch 20/20: Loss: 0.6082, Val Accuracy: 0.7623

Question 2: Modifying LeNet: Replace average pooling with max-pooling, replace the Sigmoid layer with ReLU. After doing so, redo the above experiment (Using different epochs 5, 10, 20), plot the training loss, validation loss, and validation accuracy for each training process. Discuss the changes compared to the results of original LeNet training.

- Applying ReLu *and* MaxPool: Epoch 10/10: Loss: 0.2798, Val Accuracy: 0.8791
- Applying ReLu *and* MaxPool: Epoch 5/5: Loss: 0.3534, Val Accuracy: 0.8674
- Applying ReLu *and* MaxPool: Epoch 20/20: Loss: 0.2152, Val Accuracy: 0.8907

These results are unsurprising as we would expect increasingly good results over many epochs using reLu.

Code: *tjd_LeNet_Vectorized_ReLu_MaxPooling.py*

Question 3: Use a different dataset, MNIST(PyTorch, TensorFlow), to do training and plot training loss/accuracy and test accuracy, and discuss the results.

Results:
Starting optimized LeNet training...
Epoch 1/10: Loss: 0.3342, Val Accuracy: 0.9619
Epoch 2/10: Loss: 0.0945, Val Accuracy: 0.9719
Epoch 3/10: Loss: 0.0678, Val Accuracy: 0.9806
Epoch 4/10: Loss: 0.0539, Val Accuracy: 0.9839
Epoch 5/10: Loss: 0.0442, Val Accuracy: 0.9857
Epoch 6/10: Loss: 0.0387, Val Accuracy: 0.9862
Epoch 7/10: Loss: 0.0338, Val Accuracy: 0.9858
Epoch 8/10: Loss: 0.0297, Val Accuracy: 0.9881
Epoch 9/10: Loss: 0.0263, Val Accuracy: 0.9880
Epoch 10/10: Loss: 0.0231, Val Accuracy: 0.9872

Code: *tjd_LeNet_Vectorized_MNIST.py*