Abstract

Creating a Multilayer perceptron from scratch requires much attention to detail to math and intensive studying. Before the project was given, I watched a few videos on neural networks to get a better grasp at the intuition, specifically:

- 3Blue1Brown's video *But what is a neural network? | Deep learning chapter 1* https://youtu.be/aircAruvnKk?si=MhdDyt2Vbdr-K9cj
- Artem Kirsanov's *Backpropagation from the ground up* https://youtu.be/SmZmBKc7Lrs?si=08EG7SIeszs45-KV

My model architecture and layout yield 93-94% accuracy on the MNIST dataset classification problem, and the total loss for the MPG dataset was ~ 2.3.

Methodology

The approach was bottom up. I implemented the activation and loss functions first. Taking a glance at the notes, some of the trivial activation functions Linear, ReLu, and Tanh were simple enough. This allowed for easier testing – I did not have to wait until the full application was created to test. I could test the individual activation and loss functions, ensuring they worked okay before continuing. Originally, the `train_mnist / train_mpg` files were cli with options, but it proved too much. I also started small. For MNIST, I started with 3 maximum layers, with small layer neuron counts. I kept everything as immutable as possible, to reduce bugs and issues with mutating state. This is evident in my `MultilayerPerceptron.forward()` method, using `functools.reduce`, and lack of many changing variables, just functions that take input and return an output. I wanted it to work before getting it fast. The linear regression notebook was crucial for understanding how to implement the MLP. Since I understood linear regression is one dimensional, I understood my implementation is n dimensional, thus making implementation a little easier.

For training on the MNIST dataset, I only introduced 4 total layers. The first three layers had activation functions of ReLu, ReLu, LeakyReLu, and Softmax, in that order. My loss function was CrossEntropy. Dropout, the act of randomly removing activations, is implemented yet was not used since it led to no increased gains in my architecture.

```
model = instantiate_model([
    Layer(fan_in=28*28,  fan_out=64,       activation_function=  Relu()),
    Layer(fan_in=64,    fan_out=87,       activation_function= Relu()),
    Layer(fan_in=87,    fan_out=72,       activation_function= LeakyRelu()),
    Layer(fan_in=72,     fan_out=10,       activation_function= Softmax()),
])
loss = CrossEntropy()
```

Figure 1: Architecture for MNIST training and classification.

Originally, I did not see the template code, and I tried implementing it by just looking at the project description, which made me implement activation functions for scalar values instead of vectors. I had to refactor as thus, turning my scalar functions into vectorized ones. One problem I had was figuring out

Squared Error loss. I had thought it was Mean Squared Error loss, but it should have been Squared Error, which tripped me up a bit. In addition, figuring out backpropagation was a challenge. I read the diagram plenty of times to comprehend the process. Eventually, it took a while, but I essentially understood it. In addition, taking notes in class on the `SoftMax` quirks (specifically the derivative and obtaining `dL_dz` with the Einstein sum) helped create better results for my model. The most aggravating issue was having a bad loss function implementation. I had implemented Cross Entropy incorrectly, which led to poor performance of my model, where it would only descend 1E-3 and was unstable, either vanishing or exploding my gradient calculations, thus crashing my model. I had checked my architecture countless times, from forward to backward, but everything seemed sound. Although it seemed in vain, the constant checking helped reinforce my confidence in my architecture and implement some optimizations to ensure everything was working. I am satisfied with the result of my Multilayer Perceptron.

```
activation = Linear()
model = MultilayerPerceptron(
    layers=[
        Layer(fan_in=7, fan_out=1, activation_function=activation),
    ]
)

```

Figure 2: Architecture for MPG dataset linear regression.

The MPG dataset had fewer troubles. The only issue was originally thinking linear regression could be implemented with more than one layer. Other than that, it was simple.

Results

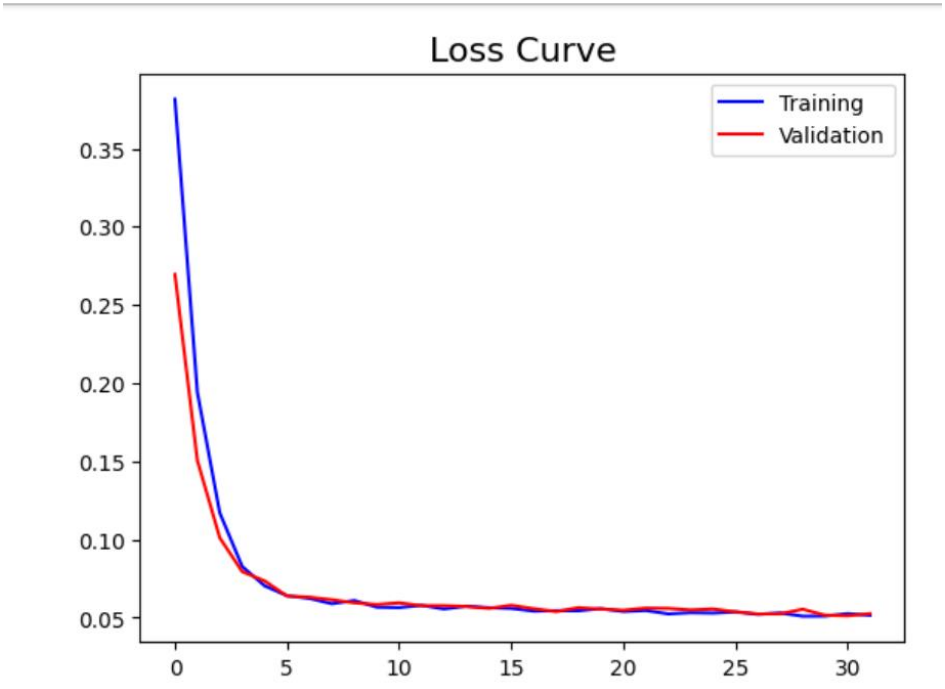

Figure 3: Loss curve for the MPG dataset, using my Multilayer Perceptron

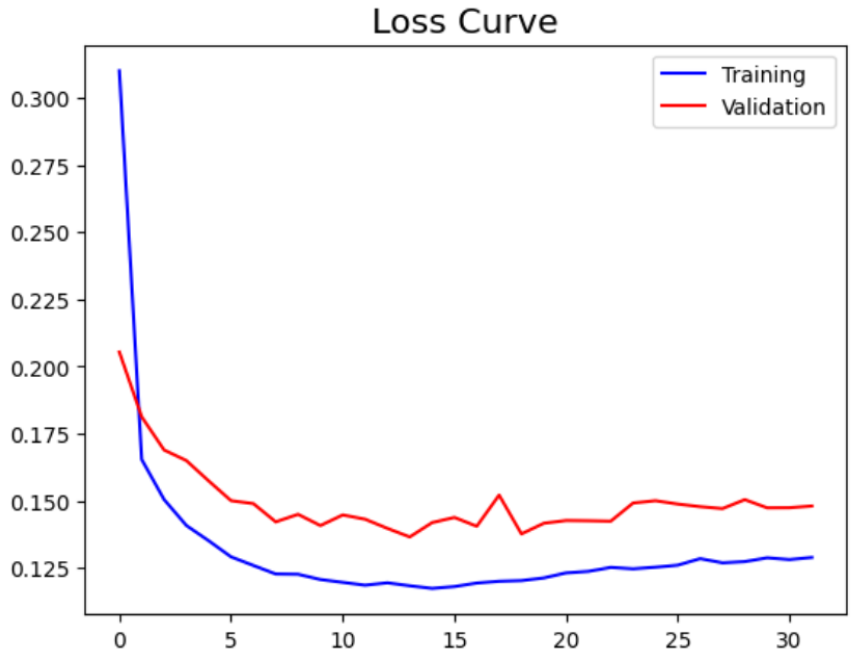

Figure 4: Loss curve for MNIST dataset, using my Multilayer Perceptron.