CS 4710 Final Project:
Multi-layer Perceptron From Scratch

Jamie Fulford

Due: May 2, 2024

Introduction

In this paper we discuss the implementation of a multi-layer perceptron (MLP) from scratch. "From scratch" means that we do not use any machine learning libraries such as TensorFlow or PyTorch. We have two implementations, one in Python and one in Rust.

Design

Our program is designed to be similar to TensorFlow's Keras API, albeit with much less features and flexibility. Here is a brief overview of the abstractions we use:

- `Activation`: We have an abstract class for activation functions.
- `Initializer`: We also have an abstract class for weight initializers.
- `Layer`: A layer is a building block of a neural network. It has a forward pass and a backward pass. Instead of having a list of neurons, we have a matrix of weights and a vector of biases. Layers also have an activation function and a weight initializer. item `Loss`: A loss function is used to calculate the error of the network.
- `MLP`: An MLP is a sequence of layers. It has a forward pass and a backward pass. The forward pass just calls the forward pass of each layer in sequence, and the backward pass works similarly. However, the backward pass must return the gradients and deltas for each layer to the optimizer. The MLP also has a loss function.
- `Optimizer`: An optimizer is responsible for updating the weights and biases of the layers, as well as training the network through it's epochs. The optimizer has a learning rate and an MLP. Per our API, it is the optimizer's responsibility to call the forward and backward passes of the MLP.

Demonstration

There are two implementations of the abstractions, one in Python and one in Rust, and they are mostly the same. We have two examples using the Python implementation and one using

the Rust implementation. We solve MNIST and XOR using the Python implementation, and we solve MNIST using the Rust implementation. Generally, the MNIST models for both achieve around 96% accuracy. The XOR model can be trained to 100% accuracy, but it is not guaranteed to converge.

Further Optimization

There are many ways to optimize our implementation. First, there is some numerical instability in the Python implementation, most likely due to exploding gradients. We can likely fix this with gradient clipping, but we did not want to overcomplicate our implementation of Stochastic Gradient Descent. Regarding speed, we only use the CPU, so we do not benefit from GPU acceleration, but thankfully numpy is natively multithreaded.

We could add more features to our API, such as more activation functions, more initializers, and more optimizers (e.g. Adam). We could also add more layers, such as convolutional layers and recurrent layers. Our implementation only supports dense layer sequential models, but we could add support for more complex architectures.