Deliverables

Your core task for this assignment is to complete the Julia programs `activations_and_losses.jl` and `dense_network_training.jl`. As you work, you should test your implementations in the notebook `scratchwork.ipynb`, which includes our Julia programs and the classic machine learning data set MNIST for recognizing handwritten digits.

When your implementation is complete, you should perform experiments to (1) optimize the model's architecture and other hyperparameters on the MNIST problem, and (2) compare timing results for the CPU and GPU versions of the model. The results of these experiments should be cleaned up for presentation in the notebook `experimental_results.jl`.

Partners and Groups

You are required to work with your assigned partner on this project. You are responsible for ensuring that you and your partner both fully understand all aspects of your submission; you are expected to work collaboratively and take responsibility for each other's learning.

You will also be assigned a code-review partner next week. To prepare for that code review, make sure you understand all of your code and that it is cleaned up to be readable by people outside your group. See the code review guidelines for more information.

Objectives

This project has two primary goals; it aims to give you experience with:

1. vectorizing neural network computations, and
2. running vectorized computations on the GPU.

The end result of this project should be a neural network implementation that is efficient enough to train on real data sets. Your implementation probably won't reach runtime parity with libraries like TensorFlow and PyTorch—they have made many additional optimizations—but it should definitely be possible to get to the same order of magnitude. This week, efficiency is the goal, which means thinking carefully about how each operation can be vectorized, and avoiding computations on single data elements at all costs.

Provided Functionality

Start by reading through the code in `dense_network_model.jl`, which provides the data structures you will be manipulating. Note that our network is no longer comprised of nodes and edges, and is instead built from a sequence of layers. Each layer stores a matrix of weights and a vector of biases, and also stores batch-matrices of activations and deltas. You should not modify this file, but you should make sure you understand what's going on!

As you work on your vectorized neural network implementation you should, as usual, test your code using a Jupyter notebook. The notebook `scratchwork.ipynb` contains a starting point for testing your network on perhaps the best-known data set in all of machine learning: the MNIST handwritten digit classification task.

Vectorized Dense Network

Your primary implementation task is to build a densely-connected network for multi-label classification. Your network should be capable of using any of the activation functions we've studied so far, including softmax activations for the output layer. We will need to re-write the activation functions in `activations_and_losses.jl`, because they now need to operate on an entire layer at a time. None of your implementations in this file should contain any loops, and it should be possible to implement each of the functions in just a couple of lines of vectorized code.

The `predict`, `backpropagation`, and `fit` functions are similar in their objectives to the ones you wrote in project 2. The only difference in *functionality* is that they operate on a batch of data rather than on one data point at a time. However, the internal implementation of `predict` and `backpropagation` will be substantially different because of the different underlying neural network object and because the goal is now to vectorize all operations.

The functions in `train_dense_network.jl` like `predict!`, `gradient!`, `update!`, etc. all operate similarly to those we saw in the previous project, except that we are now operating on a batch of data (represented as a matrix) rather than a single data point (represented as a vector). The starting-point code also provides some suggestions for useful helper functions; it is recommended (but not required) that you implement these.

Your goal is to vectorize as much of the implementation as possible, which means that the functions that operate on a batch should contain no more than one loop. Some of your

functions will of course call other functions that contain a loop, but the only place where you should have any explicitly-nested loops is in `train!` where you can have an outer loop over epochs and an inner loop over batches.

Since we'll be training on much larger data sets than in project 2, it would be helpful for the `train!` function to provide some intermediate updates. If `verbose` is `true`, then at the start of each epoch, print the epoch number, and at the end of each epoch, print the current loss. You should call `flush(stdout)` after each print to ensure they show up when they're supposed to. Here's an example of what it would look like part way through epoch # 4:

```
epoch #1 ... average loss = 1.8778878
epoch #2 ... average loss = 1.0410771
epoch #3 ... average loss = 0.77345747
epoch #4 ...
```

Training and Tuning

Once you have a working implementation, you should experiment with the model's architecture and hyperparameters to find the most effective and/or efficient ways to train on the MNIST data set. You should try varying the each of the following hyperparameters, and come up with the best options in terms of test-set accuracy, subject to taking a reasonable amount of time to train (you shouldn't brute-force it by running for a ridiculously long time).

- number of hidden layers
- sizes of hidden layers
- learning rate
- number of epochs
- batch size
- hidden layer activations
- output layer activation/loss

Include a summary of your findings in the `experimental_results.ipynb` notebook for code review and submission.

Using the Graphics Processor

Once your vectorized neural network is working and correctly classifying most of the MNIST images, we can try to speed up the computation by running it on the GPU. The starting-

point code already provides everything you need to run on the GPU, just by creating a `DenseNetworkGPU` instead of a `DenseNetworkCPU`. If you have implemented all of your functions entirely in terms of vectorized operations, then all we need to do is transfer the training/testing data to the GPU using the `cu()` function; but if you've done any direct indexing, that will result in errors that you'll need to debug.

Once you have a working `DenseNetworkGPU`, you should perform timing comparisons between the CPU and GPU implementations using the BenchmarkTools library. Note that you'll need to use `CUDA.@sync` to get valid timing results on the GPU (see here for more info).

Add a demonstration of your timing results to the `experimental_results.ipynb` notebook for code review and submission.