CSCI 402 Assignment 2
Artificial Neural Network – Mushroom Classification

Josh Loehr

Artifacts
The following files and directories are included in this submission:
- loehrj-writeup.pdf (this document)
- eigen/ - directory containing Eigen [1] source files
- src/ - directory containing all source C++ files, and input CSV files [2]
- makefile – Makefile for building the program
- README.txt
- output.txt – the script output of an example run of the program
- weights.zip – an archive of the saved weights output during the example run
- biases.zip – an archive of the saved biases output during the example run

Approach
I give a brief description of my development process, followed by an outline of all major components of my program.

Development: As someone not very familiar with C++ at the beginning of this project, I started my development process by first familiarizing myself with I/O and the Eigen [1] library. I first created parse.cpp, and figured out how to load data into Eigen's MatrixXf and VectorXf objects. Once done, I set that aside, and began creating a neural network to solve the toy problem of approximating the XOR function. This took the majority of the development time, but once done, I had generalized my code well enough that I could both easily alter the network topology and feed in differently sized inputs without making significant code changes. Once this was achieved, there was a small period of debugging and refactoring until the program was running as expected. I then spent a couple of days tuning hyperparameters and running experiments with different, randomly initialized weights until I reached convergence within a satisfactory number of epochs. I saved the seed used for that final run, and then finally touched up the code style and output formatting.

I/O: The original mushrooms.csv file is split manually into a training set and a test set, with 7312 and 812 samples in each, respectively – a 90% / 10% split. These files are parsed via the `ParseInputCSV()` method in `parse.cpp`, which iterates line by line through each CSV. The first column of each sample (the training label) is added to a separate vector of outputs, `Y`, while the rest of each row is converted to a one-hot encoded vector and added as a row to a MatrixXf object, `X`. For the mushroom problem, these one-hot encoded vectors are of dimension 126, and the ordering of the one-hot encodings is listed in `proto.hpp` as the constant string array `FEATURE_SPACES`.

Learning Speed: Through limited trial and error, the learning rate was fixed at a constant 0.001. This allowed for convergence in a reasonable amount of epochs for almost every run of the training program.

Initial Weights: The weights are initialized randomly, with mean zero and upper/lower bounds determined by a hyperparamter, which was fixed at 0.8, also via trial and error. The seed for the random number generator is determined by the current system time, unless manually specified as an argument to the program – this allowed me to replicate results between runs by using the same seed and hyperparameters.

Biases: In addition to training the weights for the network, the biases are also trained via backpropagation. Each layer's biases consist of a vector of floats with the same dimension as the layer itself. All bias values across the entire network are initialized to 1.

Topology: The network topology used includes one input layer with dimension equal to the sample vectors (126), followed by one hidden layer of dimension 64, and finally an output layer with dimension 1. Each node across all layers uses an element-wise sigmoid activation function.

Error: Error (and conversely, accuracy) is measured in a variety ways. The following equation is used to calculate Binary Cross Entropy between an output vector, *A*, and the ground truth vector, *Y*:

$$\text{BinaryCrossEntropy(A, Y)} = \frac{-Y \circ \log(A) - (\mathbf{1} - Y) \circ \log(\mathbf{1} - A)}{|Y|}$$

where $\circ$ denotes element-wise multiplication, $|Y|$ denotes the dimension of vector *Y*, and a bolded **1** represents a vector of ones with appropriate dimension. When this binary cross entropy is measured directly from the output of the last hidden layer, it is called "`bce`" within the code. When the output of the final hidden layer is first converted to an output of all ones or zeros, and then the binary cross entropy is calculated, it is referred to as simply "`error`." Accuracy, or "`acc`," is the percentage of matching rows between *A* and *Y* when *A* is first converted to outputs, as before.

The pure `bce` measurement provided the best measure of error between consecutive training epochs, and so it is used to determine divergence and when a new best set of weights has been found. `Error` and `acc` are used to determine convergence, since these are the measures that we actually care about when evaluating performance.

Training: Training uses purely stochastic gradient descent with backpropagation across weights and biases. For each epoch, the network feeds-forward the sample matrix `X`, and saves the hidden states of each layer. Binary cross entropy `bce` is recorded, and the hidden states are passed to the backpropagation algorithm. For each training sample, if the desired output does not match the current prediction, delta values are computed by backpropagating over each layer. These stored deltas are used to calculate gradients, and each weight and bias value is then updated using gradient descent with the set learning rate.

Training continues until one of the following conditions is met:

- *Divergence*: if the `bce` measure decreases consecutively for `max_divergence` epochs, training is considered to have diverged, and the program exits.
- *Convergence*: if the `error` measure falls below `error_threshold`, or `acc` increases above 99.8%, training is considered to have converged, and the training loop breaks.
- *Maximum training epochs exceeded*: if training fails to either diverge or converge within `max_epochs`, the training loop ends.

Saving Weights & Biases: During training, all weights and biases are periodically saved to files. This occurs when `bce` improves upon the previous best `bce`, and when the measured accuracy exceeds an accuracy threshold, which starts at 81% and increases by 3% each time it is achieved. The final best weights and biases are also saved upon convergence.

Summary of Hyperparameters Used:

Hyperparameter	Variable Name	Value	Description
Learning Rate	`lr`	`0.001`	Learning speed used for gradient descent
Initial Weight Range	`max_weight_val`	`0.8`	Maximum range of initial weights, centered around zero
Error Convergence Threshold	`error_threshold`	`0.02`	The threshold used to determine convergence with the `error` measurement
Maximum Training Epochs	`max_epochs`	`1000`	Maximum number of allowed training epochs without convergence/divergence
Maximum Divergent Epochs	`max_divergence`	`50`	Number of consecutively worse epochs before training diverges
Random Number Generator Seed	`seed`	`1510181022`	Unsigned integer used to initialize the random number generator (value given was used for the example run)

Results

As illustrated in the example run in `output.txt`, my network architecture was able to achieve ~99.8% training accuracy, and ~99.26% test accuracy, in just 328 training epochs over the course of ~10 minutes.

References

[1] G. Guennebaud, B. Jacob - URL http://eigen. tuxfamily. org, Accessed, 2017

[2] Lichman, M. (2013). UCI Machine Learning Repository [http://archive.ics.uci.edu/ml]. Irvine, CA: University of California, School of Information and Computer Science.