Milestone 2: Implementation

Kyra Patton (kp483), Agrippa Kellum (ask252)
Luca Leeser (ll698), Stephanie Sinwell (sas575)

1. System Description

We created a program that allows users to draw handwritten digits in a GUI, which are then classified as a digit. The classification is done by the backend, which is a neural network that we built using matrix operation optimization libraries that are available in OCaml. Our project also comprises a simple interface allowing users to generate classifiers by inputting labeled datasets and a loss function.

Key features:

1. A basic interface to allow the construction of simple fully connected neural networks
2. Usage of Lacaml (https://github.com/mmottl/lacaml) BLAS binding to optimize performance
3. Implementation of basic loss functions and backpropagation
4. GUI that allows the user to draw digits. The saved image can be converted to a matrix and fed into the neural network to be classified by a model that was trained on the MNIST dataset for handwritten digits

A neural network is a set of nested matrix operations (tensors) with a nonlinear function applied to their output. We will allow users to specify the nesting using a directed graph structure with tensors as nodes.

In order to make the neural net trainable, nodes in the network can have parameters. We implemented an algorithm for mutating these parameters based on the gradient of the loss function across the parameter space given a learning speed. Iterative mutation will produce parameters that minimize the neural network's loss for the given inputs. To allow for fast matrix computations, we used Lacaml.

After training a neural network on the MNIST dataset, the saved model can be used for classification of images. The GUI functions as a canvas that allows users to draw digits. These drawings can be classified by the trained model as a digit between 0 and 9. The model we use for classification was trained for 1 epoch, where every epoch is 60,000 steps. The accuracy of the classification is not perfect, but it can recognize drawings with some error. Because this is a rather simple neural network with only two layers, the image is misclassified more often than it would be with a more complex model, especially when two numbers are shaped similarly -- for example, 1 and 7. For the most part, though, it recognizes the hand-drawn digits, especially those with unique shapes.

There already exists a library that implements ANNs in OCaml, called Owl. The documentation is not clear about how full the implementation is, but we did not reference this package while developing our project.

2. System Design

a. Activation (actv.ml/i)

Layers consist of a weight matrix, bias matrix, and an activation function that is applied to it. This module contains constructors for different nonlinear activation functions, including both their regular representation and a derivative of said function. We currently have sigmoid, softmax, and softplus implemented.

b. Loss (loss.ml/i)

This module contains a data type representing loss functions. The loss function we have implemented will be described in the Data section below. The process of training a neural network consists of minimizing a loss function through methods like gradient descent.

c. Matrix (matrix.ml/i)

This module contains a data type defining our matrix type. It also contains basic helper functions for initializing and performing simple matrix operations or permutations.

d. Layer (layer.ml/i)

A neural network is made up of layers, where a layer contains a weight matrix, a bias matrix, and an activation function that can be applied to it. This module contains types to represent layers (implemented as matrices) and activation functions. It also supports the creation of new layers with an activation function and dimensions, as well as loading layers from saved matrix files. Functionality for updating the weights and biases of a layer is also implemented here, which is essential for the neural network training process.

e. Model (model.ml/i)

This module contains the model type, which organizes layers in a sequence. It also contains the network type, where a network consists of a model and a loss function to minimize. It will contain a function for propagating signals through the layers, ultimately producing categorizations. It will also contain the function for backpropagation, the process through which layers update themselves towards producing more predictive outputs. Also, it will have support for full passes, which encapsulate forward and backward propagation into one operation. We then define a train function which iteratively updates a network object in memory using full passes. Lastly, an infer function accepts an input and returns a predicted class for a given network; this is what the GUI uses to classify a drawn image.

f. GUI (gui.ml/i)

The main purpose of the GUI is to allow for users to draw handwritten digits, which will then be fed into the neural network and classified. When the user hits the classify button in the GUI, their drawing is saved to an image, which is converted to a matrix and fed into the neural network in the Model module. The Model module then classifies the user-drawn image as a digit; this output is returned to the GUI module, which will display the output of the classification process to the user in a text field.

The GUI module contains all the functions necessary to support the interactive drawing application. The GUI itself is a window containing a drawing pane, a classify button, a reset button, and a text field that displays the output of the neural network after classification. The drawing pane has a black background, and users are able to draw with a fixed-width white drawing tool to mimic the appearance of the MNIST images. MNIST images are 28x28, but the drawing panel is 280x280 to make the interface more user-friendly; the drawing, upon being saved, is scaled down to 28x28 to later be used for classification.

g. Image (image.ml/i)

The Image module contains functions that process images loaded into OCaml. After loading an image, it must be converted to a matrix so that it can be run through the classifier. Our implementation supports converting images with the .bmp extension.

h. Load MNIST (load_mnist.ml/i)

This module allows the GUI to load our trained MNIST model. This model is saved in a specific directory, which is required to exist for the GUI to be able to classify. We decided to put this in its own module to further abstract the workings of the neural network training from the GUI. With this module, the GUI does not need to manually load the neural network or call on any of the functions in the Model module besides infer.

i. Train MNIST (train_mnist.ml)

This module contains the functions and definitions necessary to train the MNIST dataset. The number of epochs and steps per epoch can be changed, allowing the neural network to be retrained with different parameters when re-run by the user. The trained model can be saved and later re-loaded.

j. Visualization (visualisation.ml/i)

The Visualization module contains a static representation of what a sample neural network looks like. Each node is shaded according to its activation, darker nodes having a lower activation. Every possible pathway through the neural network is shown, with each connection between two neurons colored on a scale from red (negative weights) to green (positive weights) according to its weights.

3. Module Design

See our .mli files for more details about each module.

4. Data

a. Matrix

A matrix data structure is used to represent our neural network model. We have extended the Lacaml matrix type for our utilization, allowing for fast matrix computations and syntactically simple code. Matrices can be saved into and loaded, given a user-selected file path. The matrices we have saved are .txt files in the "matrices" directory; this includes matrix_user.txt, which is the matrix version of the saved image from the GUI.

b. Loss

The loss data type contains constructors for different types of loss functions. The loss we implemented is categorical cross-entropy, which is the difference in probability distributions of the two vectors. Each element in these vectors is a probability distribution of a certain category. Our implementation allows for very easy inclusion of additional loss functions.

c. Activation

The activation data type contains constructors for different nonlinear activation functions. The data type contains a regular representation and a derivative of said function. We currently have

sigmoid, softmax, and softplus implemented. The mathematical representations of these functions and their derivatives are written in the .mli file.

d. Layer

The layer data type represents a layer in the neural network. A layer contains a matrix and an activation function. Layers can also be loaded given a user-inputted activation function and two previously-saved matrices to represent the biases and weights. The previously-saved matrices can be loaded given their file paths.

e. Model/Network

A neural network consists of several layers. Our model data type represents an entire neural network, and is therefore a list of layers. The network data type is a model combined with a loss function that it will minimize through training. Models and networks can both be saved with given IDs; they can be found in the “matrices” directory with tags "model" and "saved_net", respectively. The model that we trained for use in the GUI is saved in the "mnist" folder within the "matrices" directory, to keep it distinct from other models that could be generated through training.

f. Images

The image saved by the GUI can be found in the "images" directory in our repository. This image is named "num.bmp". There are additional images in this directory, which we utilized in our testing of saving/loading images and converting them to matrices.

g. MNIST Dataset

We will use the MNIST dataset of handwritten numbers for training and testing our neural networks. MNIST is available in OCaml through this Github repository:
https://github.com/rleonid/dsfo

5. External Dependencies

a. Lacaml

Lacaml (Linear Algebra for OCaml) is a BLAS binding for OCaml which we will use to implement the matrix functions. High performance linear algebra software such as BLAS is many orders of magnitude faster than natively implemented functions for linear algebra; a native solution would not be viable for a regular sized neural net. All relevant usage of Lacaml will be exposed in the Layer module.

b. Lablgtk

We utilized Lablgtk to build the whole GUI. Lablgtk is the OCaml interface to GTK+, a toolkit for making graphical applications. The specific modules we used were included in Lablgtk's GMain and GMisc modules. Lablgtk provided functionality for drawing images on the canvas, resetting the canvas, and saving drawings as images.

c. Camlimages

To implement loading the drawings saved from the GUI back into Ocaml, we used Camlimages. Camlimages has support for loading images, so we used it to convert images into an OCaml type which we later converted to a matrix.

d. DSFO

This dependency was a very simple interface with the MNIST dataset, allowing for easy training and inference.

e. Bigarray

Since Lacaml matrix type is a Bigarray matrix type, we utilized Bigarray for several operations not supported by Lacaml.

6. Testing

a. Testing Plan

In the initial implementation phase of the neural network, we wrote black box tests against the matrix interface to ensure the operations within were working properly. This entailed testing that the creation and saving/loading of matrices worked as expected; the matrix operations, such as matrix multiplication, were handled using Lapack, so we did not specifically test these.

Because loss and activation functions are all mathematical functions, we tested the output of our OCaml functions against the output of the actual mathematical functions. This was done in the top-level because these functions return irrational numbers, and results can differ very slightly due to floating-point arithmetic and other factors.

When basic matrix operations and the loss functions were functional, we implemented non-linear layers and back-propagation on very small networks to ensure that our algorithm works properly. Testing this entailed printing parts of the training process and ensuring that gradient updates were logical as training progressed.

Overall, the backend was fairly difficult with regards to testing since neural networks consist of highly rigid mathematical properties that are abstracted by Lacaml, meaning we know they are correct. Also, with random weight initialization of networks, they are entirely non-deterministic and thus it is impossible to come up with weight test cases for a full model that is meaningful. Thus, our testing plan was to test out the simple math operations we defined (loss and activation functions), as well as loading and saving matrix weights, and basic matrix operations. With regards to testing on the entire model, we tested on very simple datasets (randomly generated) that showed us if the network was converging properly. We also implemented a useful print matrix function that allowed us to debug matrix operations.

We tested the GUI interactively. The first phase of implementing the GUI entailed building the interface components and adding the drawing functionality. After drawing was implemented, we implemented functionality for the reset and classify buttons, which were again tested interactively, since reset simply clears the drawing canvas and classify displays its output on the GUI itself. Part of testing classify also included testing that drawings were being saved and loaded correctly. Testing the saving component was done by checking that the image was saved properly and in the right location after hitting the classify button. Testing the loading component was done interactively on utop. Loading an image converts it into a matrix to be used by the neural network; our saved images were each 28x28, which converts to a 784x1 matrix where each entry is a value between 0 and 1 based on its brightness. Because this matrix is very large, we thought it would be more efficient to look at the output of loading various images on utop, rather than writing test suites that would check against very large and hard-to-predict expected outputs. The visualization graphic was tested in a similar way.

b. Known bugs

The images saved by the GUI can be distorted on some Macs. When this occurs, the image cannot be properly classified. The images appear as normal on Linux machines, including the CS 3110 VM.

There is a known bug in loading images. The loaded images are supposed to be dumped as matrices into a text file, which is then fed into the classifier. Loading an image saved from the GUI required a Camlimages function that worked in the utop but did not work when running the compiled code. Our rather dubious workaround was to call utop in the GUI's classify function, convert the image to a matrix and save it to a file, quit, and resume the classification.

7. Division of Labor

Luca and Agrippa implemented the backend neural network interface. Agrippa completed the lower-level system design and baseline modules such as Matrix, while Luca focused on implementing the higher-level Model module. Luca also trained the neural network on the MNIST dataset and verified the outputs. Agrippa estimates he spent around 14 hours on the project. Luca estimates he spent around 50 hours on the project.

Stephanie implemented the frontend GUI, including the canvas and interaction with the backend classifier. Once completing the graphical components of the frontend, she also assisted with completing and debugging the backend in its later implementation phase. She estimates she spent around 50 hours on the project.

Kyra spent the majority of her time working on the original conception of the visualization module, which was intended to connect to the frontend GUI and visualize the network resulting from the users' input. However, this proved to be much more difficult than anticipated to implement, so she completed a more simplified visualization network. She estimates she spent about 35 hours on the project.