ECE469 – Project 2

Jonathan Lam

December 2, 2020

1 Problem description

1. Implement a fully-connected (FC) neural network architecture containing one hidden layer (with an arbitrary number of inputs, hidden nodes, and output nodes) for multiclass binary classification, where each node uses a sigmoid activation. Each output should be a binary classification using a mean squared-error (MSE) loss.

2. Allow the user to input the initial weights for the network, input a training dataset, and train a neural network initialized with those weights on the given dataset for a given number of epochs and a given learning rate. Allow the user to output the trained network weights to a file.

3. Allow the user to evaluate the dataset on a test dataset, and output the results (statistics) to a file.

4. Create a custom dataset and run steps 2. and 3. on it. I.e., find or create a binary multiclass classification dataset, generate the train and test dataset files for this dataset, decide on a network architecture (i.e., number of hidden nodes), generate the weights to initiate this architecture, train and evaluate the model, and output the statistics of the model on the test dataset.

2 File formats

These are delineated in the assignment file, but included here for completeness. For sake of reproducibility, all weights and feature values should be rounded to three decimal places (zero-padded if necessary), all other numbers (numbers of features, numbers of nodes, binary labels, etc.) should be integral values, all lines should not contain leading or trailing spaces, all values on a line should be delimited by a single space, and newlines should follow the Unix format. There are example files in the GitHub repository.

2.1 Train and test datasets

The first line should contain the following three values:

1. Number of samples
2. Feature count
3. Output count

The rest of the file will include sample information, with one sample per line. Each line should contain a list of all the features followed by a list of all the output labels (0 or 1), all delimited by spaces.

2.2 Network weights

This file assumes a fully-connected architecture. The first line should be a list of the widths (number of nodes) of each layer, including the "input layer" (whose width is the number of features). Thus, in a neural network with one hidden layer, this would be the number of input features, the number of hidden nodes, and the number of output nodes.

The rest of the file are the weights of the network. Each line contains the weights of one node, and the number of weights on each node's line should be one more than the number of inputs to that node's layer (the extra weight is the bias node). The first weight should be the node's threshold weight (which gets multiplied by -1, i.e., the negative bias), and the rest of the weights should be ordered corresponding to the order of the inputs. The nodes should be grouped together by layer, and the layers should be ordered with the more shallow layer (i.e., closer to the input layer) on top of the deeper layer.

2.3 Evaluation statistics

	Expected = 1	Expected = 0
Predicted = 1	A	B
Predicted = 0	C	D

Table 1: Contingency table

Each line except for the last two lines are the statistics for a single binary class, with the classes ordered in the same way as they are in the dataset and weights files. Each line contains:

1. A (from contingency table)
2. B (from contingency table)
3. C (from contingency table)
4. D (from contingency table)
5. overall accuracy: $\frac{A+D}{A+B+C+D}$
6. precision: $\frac{A}{A+B}$
7. recall: $\frac{A}{A+C}$
8. F1 metric: $\frac{2*\text{Precision}*\text{Recall}}{\text{Precision}+\text{Recall}}$

The final two lines contain the micro- and macro-averaged overall accuracy, precision, recall, and F1 metrics, respectively.

3 Description of implementation

3.1 Tech stack and performance considerations

The language of choice is Scheme, because Lisp is fun. Chez Scheme is chosen for the implementation. Performance was clearly not the goal here – everything is built using linked lists and intended to be as much declarative-style as possible (all elements are immutable, records are rebuilt on updates), there is no batch training, SGD is not utilized, sigmoid is used rather than ReLU, nothing is vectorized nor parallelized. This is fine for the toy datasets in use here and for understanding neural network architectures, and it also makes for a fun exercise in applying a lot of list transformations on a 3-D "tensor" (e.g., using `(apply map list M)` can be thought of as transposing a matrix `M`).

(Using Chez Scheme's `--optimize-level 3` flag decreases runtime by about 10% by skipping runtime checks, for a tiny performance gain.)

3.2 Data structures

There are two major data structures: `layer` and `model`. These are defined as record-types (i.e., like C's `struct`).

Each `layer` has three fields: `train`, `infer`, and `weights`. `train` and `infer` are procedures to train the network and to use the trained layer with its current weights (if any), respectively. `weights` stores the layer's weights, if any. For this project, three different layer types are defined: `sigmoid-layer`, `dense-layer`, and `loss-layer`. Only the `dense-layer` has weights associated with it; the other two layers are essentially hardcoded functions (the sigmoid function and the MSE loss) that pass along gradients during backpropagation.

Each `model` has two fields: `layers` and `shape`. `layers` is a list of layers, in which the final layer is always a `loss-layer`. `shape` is a list of the widths of each layer, including the "input layer" – i.e., if the first line of the network weights file were "30 20 10" then `shape` would be `(30 20 10)`. Note that this generalizes past networks with a single hidden layer; an arbitrary number of layers (with arbitrary layer types) can be supported.

(Also note that there is no "input layer," which is only a useful abstraction for describing the architecture shape and redundant once we have already built the network.)

3.3 Training procedure

The training and backpropagation are handled recursively. Each layer's `train` method should take as arguments the layer inputs (equal to the features for the first layer), the label for the current sample (only to be used in the loss layer), the learning rate (for updating weights), and the following layers. For all of the layers except the final layer (`loss-layer`) the following steps are performed:

1. (Forward pass) Call the `infer` method on this layer's inputs to get this layer's outputs.

2. Recursively call the next layer's `train` method using this layer's outputs as the next layer's inputs, and passing along the rest of the arguments. This will return the gradients of this layer's nodes, as well as the updated deeper layers.

3. (Backpropagation) Using the gradients of this layer's nodes (w.r.t. the loss) and this layer's weights, calculate the gradients of each of the input nodes w.r.t. the loss using chain rule.

4. (Backpropagation) Using the gradients of this layer's nodes (w.r.t. the loss) and the inputs to this layer, update the weights for this layer using the derivative of each weight w.r.t. this node and the chain rule in order to minimize the loss. (To practice immutability, a new layer is created with the new weights and prepended to the list of updated deeper layers.)

5. (Backpropagation) Return the gradients of the input layer's nodes and the updated network (this layer and all deeper layers).

The procedure for the final layer is a different. Using the estimated outputs and true output labels, simply computes the loss and the gradient of the loss w.r.t. the outputs of the network. (This can be thought of as the base case of the recursive procedure.)

To train a model on an example, all you need to do is call the `train` procedure of the first layer, which will initiate the recursive training of the entire network. This will return a network with all of the weights updated. The `model-train` procedure performs this on every sample of the training dataset, reporting the loss, and repeats this for the number of epochs.

3.4 Inference procedure

Evaluation of a model simply involves a folding operation over the `infer` methods of the layers of the model, where the initial value are the sample features. This is implemented with the `model-predict` procedure.

Variants are written to map this procedure over a set (i.e., a test dataset rather than a single test sample) (`model-predict-set`), and binary variants are written to output the rounded model estimates for single and sets of samples (`binary-model-predict`, `binary-model-predict-set`).

3.5 Evaluation procedure

Evaluation of a trained model occurs by performing inference on a test dataset and calculating the statistics as described in §2.3. The `model-evaluate` procedure takes as input a trained model, test feature set, and test labels, and returns a list of:

1. per-class contingency tables (i.e., A, B, C, D)

2. per-class statistics (i.e., overall accuracy, precision, recall, F1 metric)

3. micro-averaged statistics
4. macro-averaged statistics

3.6 File structure

`data/` Sample data files

`data/gen_spam_ds.js` The file used to preprocess/generate the files for the spam dataset

`stats/` Sample statistics files; filenames are in the form `dataset_lr_epochs.stats`

`weights/` Sample weights files; filenames are in the form `dataset_lr_epochs.init` or `dataset_lr_epochs.trained`

`arch-defs.ss` Neural network record type definitions

`autorun.ss` Script to easily initiate the training and test procedures (see §4.1)

`dense-layer.ss` Dense layer implementation

`loss-layer.ss` Loss layer implementation

`main.ss` Main starting point; imports dependencies and defines prompted train/test procedures

`model-io.ss` Defines utilities for loading/exporting model weights/datasets/stats

`model.ss` Defines model train, test (predict), and evaluation procedures

`sigmoid-layer.ss` Sigmoid layer implementation

3.7 Source code

The source code, datasets, weight files, and statistics files are located on GitHub at @jlam55555/nn-scheme.

3.8 Note on floating-point precision

Once when running on a peer's dataset that there was a difference of 0.001 in one number of the macro-averaged statistics, when all other outputs (statistics and trained weights) matched exactly for all other datasets, including both the provided datasets and both of our custom datasets.

Chez Scheme uses arbitrary-precision floating-point by default (analogous to Java's `BigDecimal`), which differs from the more standard IEEE 32-bit or 64-bit floating point. I believe this may be due to the rounding difference rather than some algorithmic error.

4 Instructions

Make sure Chez Scheme is installed. The following are tested with Chez Scheme 9.5 on Debian 10 (kernel 4.19.0).

4.1 Without entering the REPL

```
$ scheme --script autorun.ss
```

For sake of example, this is what I used to train and time the spam example (`--optimize-level 3` reduces execution time by roughly 10%):

```
$ time scheme --optimize-level 3 --script autorun.ss 1>/dev/null <<EOF
        weights/spam.init
        data/spam.train
        0.1
        25
        weights/spam_0.1_25.trained
        data/spam.test
        stats/spam_0.1_25.stats
        EOF
```

4.2 Using the REPL

```
$ scheme --optimize-level 3
```

```
> (load "main.ss")
> (prompt-train-test-model)
```

These two commands run the same commands as `autorun.ss`. The procedure `prompt-train-test-model` is a convenience function to facilitate the training and testing of a FC model from start-to-finish (as described for this assignment), prompting the user for the relevant model parameters and input/output files. This method calls the procedures `prompt-train-model` and `prompt-test-model`, also defined in `main.ss`, which can also be used separately. It should be easy to see from the source code that this calls the underlying model loading, training, testing, and exporting procedures described in more detail in the previous section.

5 Custom dataset

The dataset I chose was the Spambase UCI dataset[1]. This attempts to classify emails as spam or not spam based on a number of custom features. The dataset comes from a set of work emails.

The dataset comprises 4601 samples. There are 57 continuous inputs (email features), and one boolean output (whether the email is classified as spam). A more in-depth description of the features can be found at the UCI website.

- Features 1-48 indicate the percentage frequencies of 48 common words found in spam emails.
- Features 49-54 indicate the percentage frequencies of 6 common character sequences (mostly emoticons) that are commonly found in spam emails.
- Feature 55 indicates the average length of capital letter runs (consecutive capital letters).
- Feature 56 indicates the maximum length of capital letter runs.
- Feature 57 indicates the sum of the lengths of capital letter runs.

The dataset files (`data/spam.train`, `data/spam.test`, and `weights/spam.init`) are generated using the script `data/gen_spam_ds.js`. The number of outputs (arbitrarily chosen to be 64 for this problem) and hidden nodes (1 output in this case) must be explicitly specified. The preprocessing steps are:

1. Download the dataset file from the UCI repository.
2. Read in the data as a 2-D matrix, where each row is one sample.
3. Shuffle the dataset (Fisher-Yates).
4. Separate features from labels.
5. Scale each feature to the range $[0, 1]$ (min-max scaling).
6. Create an 80/20 train/test split.
7. Generate weight matrices using number of input nodes, number of hidden nodes, and number output nodes. (Weights are randomly generated from a standard normal distribution using the Box-Muller transform).
8. Export the train/test datasets and weights to their respective files.

I arbitrarily chose to use a network with 64 hidden nodes, and trained with a 0.1 learning rate for 25 epochs. This gives decent overall accuracy ($> 90\%$, which is fair for the non-critical task of spam detection). Given that this dataset is much larger than the provided ones, and the network is larger, the training takes much longer; 25 epochs takes roughly 26 seconds on my system (i7-2600 CPU) with `--optimize-level 3`, or 30 seconds without it.

[1] https://archive.ics.uci.edu/ml/datasets/Spambase