Inference & Evaluation Engine for Feed-Forward Neural Networks in Haskell

CS-IS-2010-1: Functional Programming Report

Supervisor: Partha Pratim Das

Santripta Sharma
(1020211136)

May 12, 2024

Contents

1 Introduction

Through the course of this project, we have developed a command-line tool to run inference on (or evaluate) an arbitrary feed-forward neural network (for classification tasks). The primary goal for the project was to gain familiarity with haskell (and the functional programming paradigm) by implementing a medium-sized project using it.

In this regard, the project has been quite succesful, forcing me to cover a lot of the bases in a relatively short time. The entire process took me from the basics, performing command-line I/O, and using the haskell tool stack, all the way to writing parser combinators for binary data, using monads, and implementing test suites with HSpec.

Besides the learning outcomes from the project, the tool that has been developed is also useful in practical settings. It has been designed & developed keeping in mind optimality, generalizability, and usability. An example use case of this tool is one where the user has been handed a trained classifier network (by, say, a tech-consultancy firm), and they wish to verify its performance on their data, without writing an evaluation/inference script on their own (due to time constraints or inexperience with the technology).

2 Background & Motivation

2.1 Motivation

Today, Machine Learning has left no domain untouched. What initially started as a niche subfield of statistics has now found its application in nearly every sector & discipline. This poses a simple problem: A large chunk of the users of these models are not familiar with how they work or how to use them. This has led to widespread black-boxing, where the development/training of the model is often outsourced to someone with this technical knowledge, with the 'client' being given instructions on how to use the model.

Besides the inherent security risk, where the external party could place a backdoor into the model, this approach also forces the client to take the external party at their word when it comes to the effectiveness of the model, at least initially, which opens the door up to being defrauded. Additionally, in some cases, the instructions provided may be obtuse, rendering the client unable to use the model.

Therefore, there is a need to be served for providing a solution that can perform inference and evaluation on these models, while also being friendly to users who aren't very tech-savvy. In this project, we particularly focus on a popular subclass of ML models, the feed-forward classifier, which is a generalisation of linear/logistic regression, and a foundational piece in most deep neural networks, with this unique position leading it to be one of the most common architectures for these models.

2.2 Background: Feed-Forward Networks

The Feed-Forward Network (FFN) is a versatile network architecture for deep neural networks, able to be used as both classifiers or regressors (though this project restricts its scope to the former). It is comprised of a set of layers ($L = \{L_1, L_2, \ldots, L_{n_{hidden}+2}\}$), each containing nodes, called 'neurons', each of which are connected to every neuron in the next layer by weighted edges. Each neuron also has an associated 'activation' value, though this is only

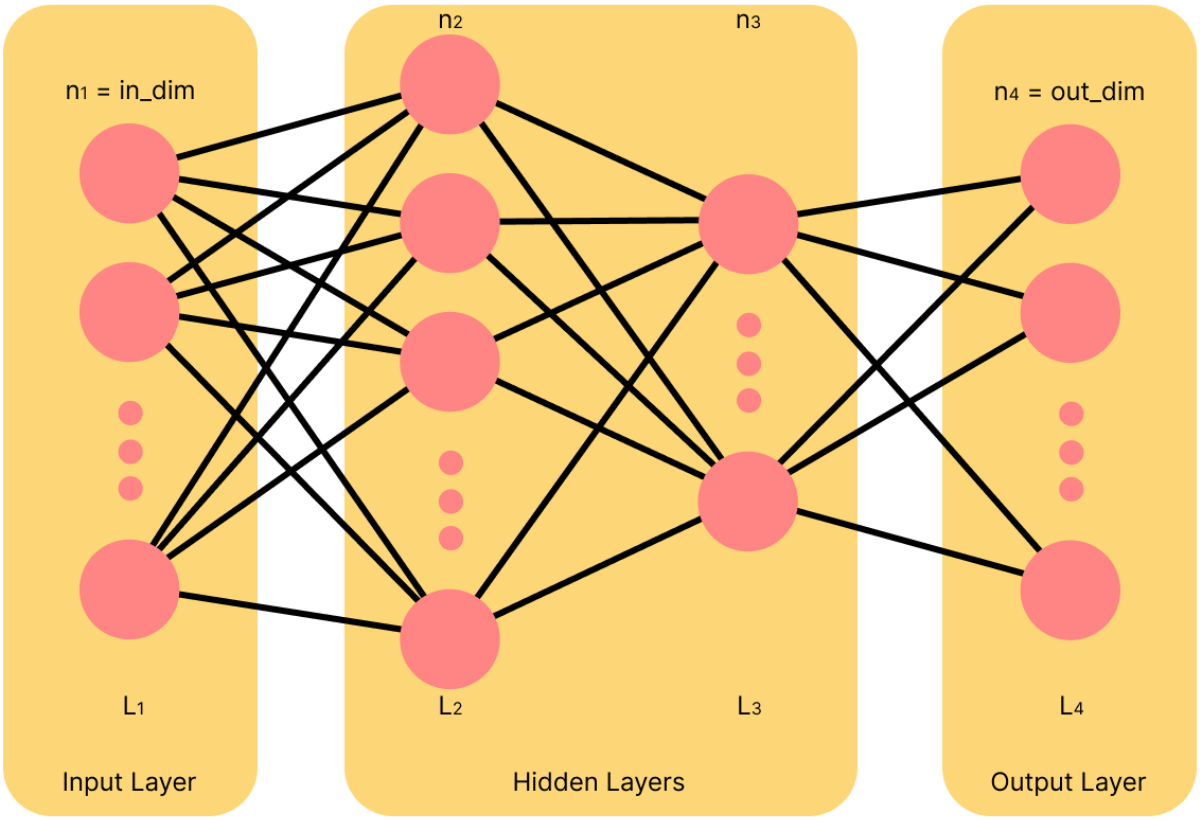

Figure 1: a generic FFN with $n_{hidden} = 2$

meaningful while the model is being inferenced. Figure 1 provides a pictorial representation of this architecture. There are three types of layers in the networks:

- **Input Layer** (L_1): The first layer of the network is the input layer. Here, each neuron's activation value corresponds directly to the value of a distinct feature in the datapoint currently being inferenced. From this, we derive the relation:

$$|L_1| = in\_dim \tag{1}$$

  Where in_dim is the dimensionality (number of features) of a datapoint in our dataset.

- **Hidden Layers** ($L_2, \ldots, L_{n_{hidden}+1}$): The layers between the first and last (exclusive) are the hidden layers. **The model's architecture is parametrised by the number of hidden layers (n_{hidden}), and the size of each hidden layer (the number of neurons in it)**, which can both be any integer (though these are usually selected very carefully depending on the problem). On each of these layers, a non-linear activation function is applied after activations are calculated, allowing the network to capture non-linear relations (eg. ReLU).

- **Output Layer** ($L_{n_{hidden}+2}$): The final layer of the network is the output layer. Here, each neuron's activation values correspond (through some function) to the likelihood that the datapoint currently being inferenced belongs to a distinct class. This unknown function is called a non-linear activation function, and **is another parameter in the model's architecture**, with the softmax function being most commonly applied to classification tasks. We also have the relation:

$$|L_{n_{hidden}+2}| = out\_dim \tag{2}$$

Where *out_dim* is the number of distinct classes our datapoint can belong to.

While the model architecture is parametrised by the number of hidden layers, size of each hidden layer, and the choice of activation function to be applied at the output layer, being a machine learning architecture, it also contains a large number of trainable parameters. We omit discussion of the training procedure, due to it being out of scope for this project.

The output of the training are the trainable parameters, the weights & biases. As mentioned earlier, $\forall i > 1$, each neuron in L_i is connected to each neuron in the layer L_{i-1} by a weighted edge. Then, the weights matrix for this layer is simply a submatrix of the adjacency matrix representation of the subgraph induced by these adjoining edges (if we discard the matrix elements corresponding to nodes not joined by an edge, i.e., nodes in the same layer). Then, if $|L_i| = n_i, |L_{i-1}| = n_{i-1}$, the weights matrix for this layer $W_i : n_i \times n_{i-1}$. The propagation formula for the FFN at this layer is given by:

$$X_i = \text{Activation}(W_i X_{i-1} + B_i) \tag{3}$$

Where X_i, X_{i-1} are the neuron activations at layers $i, (i-1)$ respectively (these are column vectors, so $X_i : n_i \times 1, X_{i-1} : n_{i-1} \times 1$), $B_i : n_i \times 1$ is the bias column vector for this layer (trained parameter), and Activation$(\cdot)$ is the non-linear activation function (preserves matrix dimension). The knowledge that activation vectors (X_is) are represented as column vectors, along with an understanding of matrix multiplication can be used to derive the following lemmas:

Lemma 1 *Given the weights matrix $W_i : a \times b$ at layer L_i, we have that $n_{i-1} = b$.*

Lemma 2 *Given the weights matrix $W_i : a \times b$ at layer L_i, we have that $n_i = a$.*

Similarly, using our knowledge of matrix-matrix addition, we have:

Lemma 3 *Given the weights matrix $W_i : a \times b$ at layer L_i, the biases vector B_i has the shape $B_i : a \times 1$.*

3 Literature Survey

3.1 Existing Neural Network Libraries

Before we implement custom logic for reconstructing and running inference on a FFN, we first check the Haskell ecosystem for any existing neural network/machine learning libraries.

We immediately found neural, a pure haskell framework for training and inferencing on various kinds of neural networks. Additionally, we find grenade, neural-network-hmatrix, and hnn. Unfortunately, all of them have the same problems.

For our chosen problem, they are all overkill, attempting to provide a generic framework for training and using neural networks. The training part adds the majority of the overhead to the interfaces provided by these libraries, but we do not require any training support for this project. Adding to this, our project only cares about a very specific architecture, the FFN, whereas these projects pursue support for many architectures, leading to a less overall flexible (programmatic) interface for our case, compared to one we design on our own.

For these reasons, we choose to implement the inference without the support of a deep learning library.

Library	$n = 10$	$n = 50$	$n = 100$
DLA	2.65us	289.0us	2.24ms
Hmatrix	1.32us	55.8us	292.0us
NumHask	714.0us	63.5ms	593.0ms
Massiv	12.0us	205.0us	1.52ms
Massiv (Parallel)	76.1us	220.0us	866.0us
Matrix	12.6us	1.1ms	8.44ms
Naive C Implementation	51us	323us	4.78ms

Table 1: Linear Algebra Library Benchmark (matrix multiplication of size $n \times n$)

3.2 Fast Linear Algebra in Haskell

As seen in (3), the computation we perform during inference is essentially a series of matrix multiplications (and additions). In order to optimise this, our first task is to determine how haskell supports these operations.

At a cursory glance, it can be seen that there are various linear algebra libraries available for use with haskell. In order to hone down on one library for usage, this benchmark from 2015 is the best evidence we have to go off (short of running our own benchmark). Table 1 displays their results for the matrix multiplication task.

According to this benchmark, the Hmatrix library achieves SOTA performance for both the matrix multiplication and repeated matrix multiplication task (in a single-threaded setup). Then, we choose to utilise this library for maximising performance.

4 Problem Statement

4.1 Problem Statement

Develop a classification model over the Iris Dataset and store the model. Then write a Haskell code to restore the model, input new data (based on the four features of the iris) and generate a prediction in real time.

1. Input: The Iris Dataset.
2. Output: Real-time prediction for the class of the flower of a new data row containing the sepal width, petal width, sepal length and petal length.
3. Method: You can use Python to train and save a classification model (SVM or NN). However, restoring the model and the real-time prediction of a new data row has to be written only in Haskell.

4.2 Objectives

Based on this problem statement & initial motivation, we aim to develop a system which:

- Should be able to load a trained classification model.
- Given any new data point of the same format as the iris dataset by the user, should be able to use the loaded model to make a prediction on this data and report it to the user.

- Should be as performant as possible, since the problem description suggests that the application domain is a real-time system.
- Should be simple to use for a layperson.
- Should generalise beyond the iris dataset.

5 Scope

In order to fulfill our objectives, we have developed a system with the following featureset:

1. Parsing capabilities to restore the shape, weights, and biases of **an arbitrary** FFN classifier from a file.
2. Interactive & batched modes of performing inference.
    - **Interactive**: The user enters a single new datapoint and gets predictions for it in real-time. Structured as a Read-Eval-Print-Loop (REPL).
    - **Batched**: The user provides many datapoints in a csv file, and receives predictions for each point.
3. An evaluation mode: an extension of batched inference mode which also takes in expected outputs, and determines model accuracy.

To expand on point 1, since the type of model (FFN) is known, we can generalise the system to work with any classifier following that architecture, since this is independent of the particular dataset used, as long as that dataset only contains numeric feature. This allows the system to be reused for similar tasks beyond the Iris dataset, providing us the generality we desire.

However, we do limit the class of models we allow to using ReLU (at the hidden layers) & Softmax (at the output layer) activations, which covers most, but not all, FFN classifiers.

6 Methodology & Design

6.1 Architecture

Since we are only concerned with running the inference for a new row for this dataset, we train our classification models separately (for testing purposes) before runtime. At runtime, we restore the structure, weights, and biases of the model, and push the given feature vector through the model in order to arrive at the inferred output.

Then, the system can broadly be divided into two modules, the training module & the inference module. These operate independently from each other, with the only interface between them being the inference module taking the training module's output (model structure, weights, biases) as its input, which is then used to reconstruct the model. Figure 2 illustrates the general flow of this architecture.

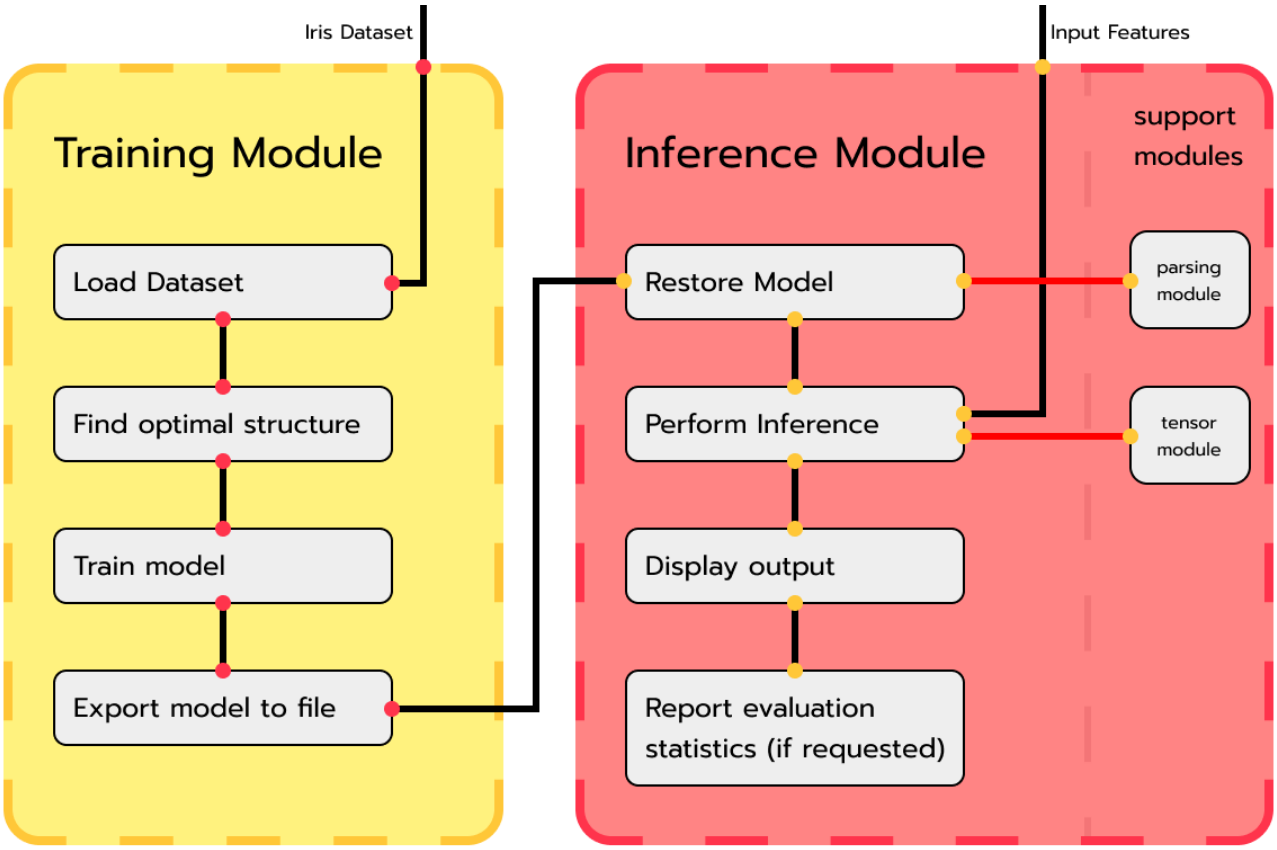

Figure 2: the high-level design of the system

6.2 Trained Model Format

From Section 2.2, we know that a feedforward neural network is comprised of several layers of neurons, with each layer (besides the input layer) having a matrix of weights & a vector of biases associated with itself.

Then, our naive model format is a binary file structured as follows:

```
number of layers
for each non-input layer in the network:
    weights matrix rows
    weights matrix columns
    weights matrix
    biases vector length
    biases vector
```

Immediately, from lemma 3, we can reduce this to a compacted format:

```
number of layers
for each non-input layer in the network:
    weights matrix rows
    weights matrix columns
    weights matrix
    biases vector
```

We can make another space optimisation by noticing that lemma 1 allows us to infer the number of columns in the weights matrix of one layer from the number of neurons in the previous layer. Since the format is defined iteratively over layers, we can exploit this relation while parsing using memoisation. All we need to begin is the number of neurons in the input layer.

```
number of layers
number of neurons in input layer
for each non-input layer in the network:
    weights matrix rows
    weights matrix
    biases vector
```

Since each stored integer here is 4 bytes wide, we have brought down the size of our format from $4 \times (3l+1) + D = 12l + D + 4$ bytes, where l is the number of non-input layers and D is the constant size of the actual data (weights & biases) to $4 \times (l+2) + D = 4l + D + 8$ bytes, which scales much better with number of layers. Then, we have managed to enhance the information density of our format using our observations about the network architecture.

6.3 Training Module

The purpose of the training module is to provide us with a set of models we can use to test our inference module. As such, we don't strictly require a high performance model. However, we still perform some basic hyperparameter selection after loading the dataset in using a 5-fold cross validation approach to determine the shape of the network (number & size of hidden layers).

The model architecture used is the one mentioned above, a simple feedforward network, a sequence of linear layers using the ReLU activation function between the hidden layers,

and the softmax activation at the end. Using a grid search, we find the top 5 performing shapes, and also one poorly performing shape, out of a generated set of possible shapes for the network.

Before fitting 6 models using these shapes on the data, the data is transformed by converting the output classes column from textual labels to numerical labels, to avoid any ordering ambiguities arising during the training process. This transformed data is then split into training & test sets, and we fit the models on the training set.

Finally, we write the networks' weights and biases into model files, which can then be used by the inference module.

6.4 Inference Module

The inference module constitutes the bulk of the codebase. It produces an executable with the following signature:

```
inference-exe <path_to_model_file> [<path_to_batch_csv>] [<path_to_labels_csv>]
```

If the second, optional argument is omitted, it starts in interactive mode. Otherwise, it starts in batched mode. If the third argument is provided, it runs in evaluation mode.

In either case, its first task is to restore the model from the given model file. Here, we utilise the parsing support module, which is written using parser combinators. Further implementation details of the parser are discussed in the Test Plan & Prototype Details section.

Through this parsing process, the model shape, and its parameters are loaded into a data structure which takes the form of a list of layers, where each layer stores its input & output shapes, and the weights & biases matrices.

Next, based on which mode it was started in, the system either enters a REPL or parses the input batch csv file it was provided (again, with the help of the parsing support module). In either mode, once the feature vector(s) are acquired, the tensor support module is used to perform the sequence of computations that lead to the final prediction.

The final output (for each feature vector) is the predicted class and the probabilities associated with each class. If the program was started in evaluation mode, it will also report an accuracy metric, that is, how many of the predicted outputs matched their correct labels as given in the labels csv file, as a percentage.

7 Work Done

7.1 Tooling

7.1.1 Training Module

The training module is written in python, using numpy, pandas, and sklearn to perform the data processing & model training/selection.

7.1.2 Inference Module

The inference module is written in Haskell. We use the Haskell Tool Stack (or just Stack) as our build tool. The following dependencies are used:

- **hmatrix**
  As described earlier in 3.2, this library provides vector & matrix types, along with the definitions for common mathematical operations on them, including matrix multiplication, conversions, etc.

  It is the backbone of tensor support module, which is used to finally perform the inference once the inputs have been parsed.

- **attoparsec**
  Attoparsec is a highly performant parser combinator library, focusing on complex text & binary file formats. Attoparsec forms the foundations of the parsing support module, which is used for parsing both the model files & csv files for batched input/evaluation mode.

7.1.3 Testing/Miscellaneous

A mix of bash & python scripts is used to implement the end-to-end tests and certain convenient utilities (eg. infer.py for loading and making predictions using sklearn, used to verify inference module). We also utilise the timeit package to benchmark our performance.

7.2 Test Suite

Due to the large number of disjoint moving parts in the project, the plan is to implement a comprehensive test suite consisting of unit & end-to-end tests. Additionally, we will be profiling our system to ensure it meets the real-time requirements.

7.2.1 Unit Testing

First, we note that the training module does not require testing, due to its simplicity and it almost exclusively using sklearn library functions. For the inference module, all the parser functions, and many of the inference functions are complex enough to warrant unit testing. We plan to implement these using HSpec, which is the testing framework Stack provides.

In order to cover the large space of inputs for these functions, we plan to heavily depend on generated test suites, as opposed to hand-crafted ones.
Consider two examples:

- **For the matrix parser:** We generate random real-valued matrices, stringify them, run them through the parser, and verify that the same matrix is recovered.
- **For batch softmax:** Use single-vector inference softmax as oracle and verify batch softmax.

7.2.2 End-to-End Testing

We use an end-to-end test to verify the correctness of our inference module. Here, we treat our sklearn model as an oracle, randomly generating many batches of feature vectors (in addition to the existing test split), running them through both our oracle & the inference module, and comparing the output activations using mean square error.

7.2.3 Profiling

To profile our inference, we run batch prediction on networks with various shapes, timing how long the inference itself takes (not the parsing), and compare it with the time taken to run the same inferences in python with scikit-learn.

7.3 Implementation

7.3.1 Project Structure

```
src
|
-- inference              <--- inference module
|  |
|  -- app/Main.hs         <--- driver code
|  -- src/Lib.hs          <--- tensor support module
|  -- src/Parser.hs       <--- parsing support module
|  -- test/Spec.hs        <--- test suite
|  ...
-- models                 <--- pickled sklearn models (for testing)
-- weights                <--- model files
-- infer.py               <--- sklearn inference helper
-- bench.ipynb            <--- time comparison benchmark
-- e2e.ipynb              <--- end-to-end test implementation
-- gen.ipynb              <--- dataset generator for testing
-- iris*.csv              <--- datasets
-- train_model.ipynb      <--- training module
```

7.3.2 Functionality/User Flow

7.3.3 Parsing a model file

7.3.4 Challenges & Mitigation

7.3.5 Migrating Libraries

7.3.6 LAPack & Blas

7.3.7 Batch inference

8 Results & Discussion

8.1 Example Runs

```
santr@Legion5 ~\..\..\inference  main > stack exec inference-exe ../weights/trained_iris_model_11_14.txt
Enter sepal length, sepal width, petal length, petal width (or nothing to exit):
5.5,4.2,1.4,0.2
Class probabilities: [0.9999928701987176,7.129801280611513e-6,1.7656920753763803e-15]
Predicted class: 0
Enter sepal length, sepal width, petal length, petal width (or nothing to exit):
6.0,2.9,4.5,1.5
Class probabilities: [3.1440802848389 45e-2,0.9409172096439032,2.764198750770721e-2]
Predicted class: 1
Enter sepal length, sepal width, petal length, petal width (or nothing to exit):
6.0,2.9,4.5,1
Class probabilities: [4.6936239833101695e-3,0.9947834890411401,5.228869755496816e-4]
Predicted class: 1
Enter sepal length, sepal width, petal length, petal width (or nothing to exit):
5.5,2.8,4.5,1
Class probabilities: [1.5967634846730167e-2,0.9778894929350018,6.142872218268111e-3]
Predicted class: 1
Enter sepal length, sepal width, petal length, petal width (or nothing to exit):
3.4,2.3,4.5, 2.3
Class probabilities: [6.80055788704634e-2,0.10147692802600372,0.830517493103533]
Predicted class: 2
Enter sepal length, sepal width, petal length, petal width (or nothing to exit):
```

Interactive Mode

```
santr@Legion5 ~\..\..\inference  main > stack exec inference-exe ../weights/trained_iris_model_11_14.txt ../iris_test_unlab.csv
Class probabilities: [[6.788069367870167e-2,0.10159029607449008,0.8305290102468084],[3.590190767435096 4e-3,0.9961394191781754,2.7039005438953006e-4],[0.9999928701987176,7.129801280611513e-6,
1.7656920753763803e-15],[6.788069367870167e-2,0.10159029607449008,0.8305290102468084],[0.9997931459107217,2.0685408849581693e-4,7.826406948256144e-13],[6.788069367870167e-2,0.101590296074490
08,0.8305290102468084],[0.9998903151803747,1.0068481935563328e-4,2.6963339048919205e-13],[1.0299402786280224e-2,0.986775576307 32,2.9250289063996 46e-3],[1.4242720084203042e-2,0.979760844113955
36,5.996435801843 2225e-3],[2.7740609506729528e-3,0.997059275701807,1.6666334752007638e-4],[6.788069367870167e-2,0.10159029607449008,0.8305290102468084],[7.975016182034585e-3,0.99040362544341
49,1.621358374550 5727e-3],[1.9719227979717653e-2,0.9697177836727938,1.0562988347488526e-2],[1.9552100629353832e-2,0.9694046422326 73,1.104325713797314e-2],[3.411633441793321e-2,0.932202126853
6523,3.368153872841441e-2],[0.9996244318644812,3.7556813194117885e-4,3.5775828747613247e-12],[3.144080284838945e-2,0.940917209643903 2,2.764198750770721e-2],[3.484905018062 63e-2,0.93253095417
56764,3.261999564369738 6e-2],[0.9990679978473168,9.320821417669997e-4,1.0916246294021249e-11],[0.9999645962882338,3.540371174993943 5e-5,1.628756972375877 5e-14],[6.788069367870167e-2,0.101590
29607449008,0.8305290102468084],[5.444840066956508e-2,0.8579246165017527,8.762698282868227e-2],[0.9987756550648446,1.224344925649546 6e-3,9.505888716005339e-12],[0.9990627882499301,9.37211715
6607511e-4,3.440919841652506e-11],[7.930580999432195e-2,0.32473236369954167,0.5959618263061363],[0.9999858665354969,1.413346442988985e-5,7.334615631293096e-14],[0.9992747343382843,7.25265660
3722888e-4,1.343233997165538 8e-12],[5.003800141475212e-3,0.9943827488901111,6.134509684137407e-4],[4.083488964380474e-3,0.995800297690 2944,1.162133453251 1472e-4],[0.9991702002041827,8.297997
925089694e-4,3.308331912412361e-12],[6.788069367870167e-2,0.10159029607449008,0.8305290102468084],[7.249806764286 14e-2,0.751735233847934 2,0.175766698509 20437],[0.999863344042619,1.3665595710
495498e-4,2.760523479907108e-13],[7.612686591745083e-2,0.26448901343910536,0.6593841206434439],[6.788069367870167e-2,0.10159029607449008,0.8305290102468084],[2.1622328571595368e-2,0.96736875
48024659,1.100891662593875 8e-2],[0.9998457122530138,1.5428774692624616e-4,5.998422087635437e-14],[6.79750122643889e-2,0.1220246084978843 1,0.8100037923772 67]]
Predicted classes: [2,1,0,2,0,2,0,1,1,1,2,1,1,1,1,0,1,1,0,0,2,1,0,0,2,0,0,1,1,0,2,1,0,2,2,1,0,2]
```

Batched Mode

8.2 Testing Outcomes

8.3 Profiling Outcomes

8.4 Limitations

- Lack of runtime checks for user input, leading to a restrictive input format & several non-graceful exits, without any useful feedback to the user.
- The csv input in batch mode can't end on a blank line (parser bug)

- Batch mode doesn't provide summary statistics
- Functionality is limited to feed-forward classifiers, though this is purely due to presentation of the output
- The command-line interface, while designed for usability, still may serve as a barrier for the most technically-challenged individuals
- Requires the weights to be in a well-known format, and for the user to locate the weights file manually
- Class labels aren't currently considered

9 Conclusions

10 Extensions & Future Work

- Infer the format of the weights file based on a few well-known formats (.pkl, .pth, .safetensors), instead of only working for our custom format. This requires more than just looking at the extension, as we have to further infer where the weights and biases of each layer lie based on the keys in the file.

11 References