FP Project Presentation: Iris NN Inference

Santripta Sharma

April 4, 2024

Problem Definition & Solution Specification

Problem Statement & Requirements

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.

Problem Statement & Requirements

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.

Problem Statement & Requirements

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.

Problem Statement & Requirements

From this statement, we derive our requirements:

Requirements

Problem Statement & Requirements

From this statement, we derive our requirements:

Requirements

- The system should be able to load a trained classification model.

Problem Statement & Requirements

From this statement, we derive our requirements:

Requirements

- The system 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, the system should be able to use the loaded model to make a prediction on this data and report it to the user.

Problem Statement & Requirements

From this statement, we derive our requirements:

Requirements

- The system 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, the system should be able to use the loaded model to make a prediction on this data and report it to the user.
- The inference part of the system should be as performant as possible.

Specifications

In order to meet these requirements, we build a system with the following feature set:

Specifications

In order to meet these requirements, we build a system with the following feature set:

1. Parsing capabilities to restore the shape, weights, and biases of **an arbitrary** FFN-based classifier from a file.

Specifications

In order to meet these requirements, we build a system with the following feature set:

1. Parsing capabilities to restore the shape, weights, and biases of **an arbitrary** FFN-based 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).

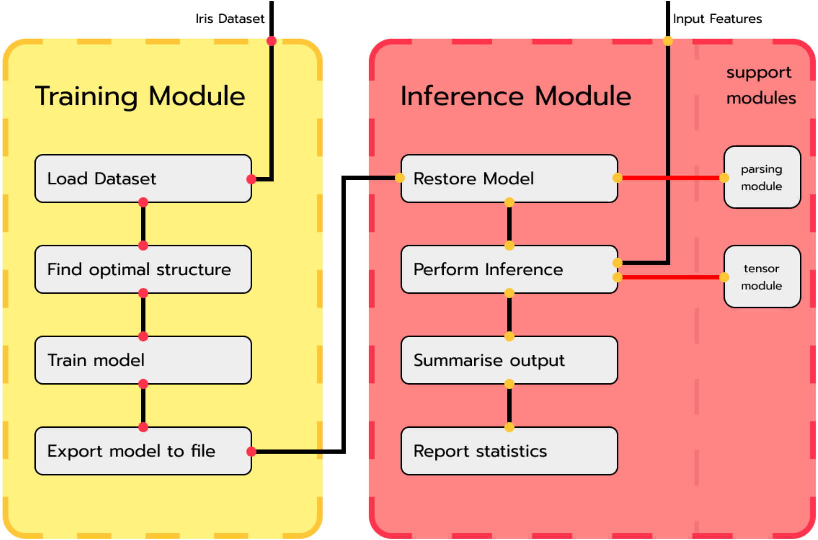
Specifications

In order to meet these requirements, we build a system with the following feature set:

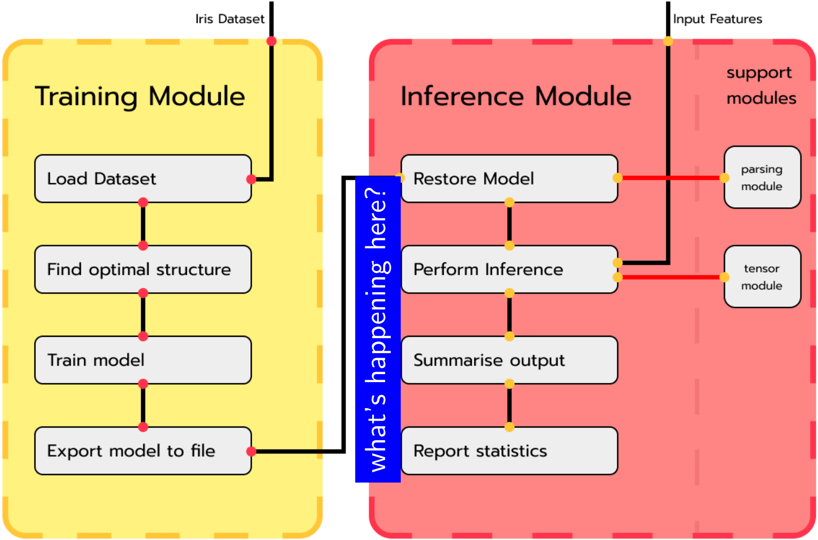
1. Parsing capabilities to restore the shape, weights, and biases of **an arbitrary** FFN-based classifier from a file.
2. Interactive & batched modes of performing inference.
    - **Interactive**: The user enters a single new datapoint and gets predictions for it in real-time. Structured as a Read-Eval-Print-Loop (REPL).
    - **Batched**: The user provides many datapoints in a csv file, and receives predictions for each point. Additionally, the system reports aggregate statistics.

System Design & Architecture

High-level view



High-level view



Model File Format

- For the prototyping stage, we opt to use a plaintext format for the trained model parameters, since this allows easier debugging and editing.

Model File Format

- For the prototyping stage, we opt to use a plaintext format for the trained model parameters, since this allows easier debugging and editing.
- Using the propagation formula $Wx + b$, we can infer the incoming and outgoing dimension of each layer from its weights and biases matrices alone.

Model File Format

- For the prototyping stage, we opt to use a plaintext format for the trained model parameters, since this allows easier debugging and editing.
- Using the propagation formula $Wx + b$, we can infer the incoming and outgoing dimension of each layer from its weights and biases matrices alone.

Then, our model file format can be described using this output spec:

```
for each layer in the network:
weights matrix
biases vector
```

If we make the assumption that the network uses a ReLU activation at every layer except the last, where softmax is used, this information is sufficient to restore the network.

Training Module

- The training module provides us with a set of models we can use to test our inference module.

Training Module

- The training module provides us with a set of models we can use to test our inference module.
- Using a 5-fold cross-validation, we perform a grid search to find the best 5 network shapes for the architecture mentioned above, as well as the worst shape.

Training Module

- The training module provides us with a set of models we can use to test our inference module.
- Using a 5-fold cross-validation, we perform a grid search to find the best 5 network shapes for the architecture mentioned above, as well as the worst shape.
- We then train these 6 models on a split of the dataset, and save them to disk.

Inference Module

- The inference module produces an executable with the following signature:
  `inference-exe <path_to_model_file> [<path_to_batch_csv>]`
  If the second, optional argument is omitted, it starts in interactive mode. Otherwise, it starts in batched mode.

Inference Module

- The inference module produces an executable with the following signature:
  `inference-exe <path_to_model_file> [<path_to_batch_csv>]`
  If the second, optional argument is omitted, it starts in interactive mode. Otherwise, it starts in batched mode.
- It first restores the model from the given model file. Here, we utilise the parsing support module.

Inference Module

- The inference module produces an executable with the following signature:
  `inference-exe <path_to_model_file> [<path_to_batch_csv>]`
  If the second, optional argument is omitted, it starts in interactive mode. Otherwise, it starts in batched mode.
- It first restores the model from the given model file. Here, we utilise the parsing support module.
- 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.

Inference Module

- The inference module produces an executable with the following signature:
  `inference-exe <path_to_model_file> [<path_to_batch_csv>]`
  If the second, optional argument is omitted, it starts in interactive mode. Otherwise, it starts in batched mode.
- It first restores the model from the given model file. Here, we utilise the parsing support module.
- 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.
- Then, based on which mode it was started in, it collects user input accordingly, and uses the tensor support module to perform the inference, and reports the prediction & activation probabilities to the user.

Example

```
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: [3.144080284838945e-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):
```

Figure 1: examples of inference in interactive mode

Tooling

Languages

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

Inference Module
The inference module is written in **Haskell**. We use the **Haskell Tool Stack** (or just Stack) as our build tool.

Misc/Testing
A mix of bash & python scripts is used to implement the end-to-end tests and certain convenience utilities.

Inference Module Dependencies

- **matrix & vector**

Inference Module Dependencies

- **matrix & vector**
  As the name suggests, these two packages provide a Matrix & Vector type respectively, also providing the definitions for common mathematical operations on them.

Inference Module Dependencies

- **matrix & vector**
  As the name suggests, these two packages provide a Matrix & Vector type respectively, also providing the definitions for common mathematical operations on them.

  Together, they form the tensor support module, which is used to finally perform the inference once the inputs have been parsed.

Inference Module Dependencies

- **matrix & vector**
  As the name suggests, these two packages provide a Matrix & Vector type respectively, also providing the definitions for common mathematical operations on them.

  Together, they form the tensor support module, which is used to finally perform the inference once the inputs have been parsed.
- **parsec**
  Parsec is Haskell's standard parser combinator library, allowing us to write atomic parsers and compose them together. Parsec forms the foundations of the parsing support module, which is used for parsing both the model files & csv files for batched input mode.

Inference Module Dependencies

- **matrix & vector**
  As the name suggests, these two packages provide a Matrix & Vector type respectively, also providing the definitions for common mathematical operations on them.

  Together, they form the tensor support module, which is used to finally perform the inference once the inputs have been parsed.
- **parsec**
  Parsec is Haskell's standard parser combinator library, allowing us to write atomic parsers and compose them together. Parsec forms the foundations of the parsing support module, which is used for parsing both the model files & csv files for batched input mode.

  Discussion of the specifics of the parser implementations is deferred to the Prototype Details section.

Test Plan

Strategy

The plan is to implement a test suite consisting of both unit & end-to-end tests. Additionally, we will be profiling our code to ensure it meets the real-time requirements.

Unit tests will be implemented using **HSpec**, the testing framework provided by **Stack**.

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.

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.

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.

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.

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.

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 on generated output activation matrices.

E2E 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.

Prototype Details

Feature Set

The prototype implements the following features:

- Complete restoration of any feed-forward classifier, given its parameters in the model file format, assuming it uses ReLU + Softmax activations.
- Interactive mode REPL, where the user can enter new datapoints, and get prediction outputs (activations + class label) for them.
- Batch mode, where the user can provide a csv file of datapoints to run inference on, and get prediction outputs for each point.

Limitations of the Prototype

- Lack of runtime checks for user input, leading to a restrictive input format & several non-graceful exits, without any useful feedback to the user.
- Class of models that works limited to ReLU + Softmax activation based classifiers (large but not comprehensive).
- The csv input in batch mode can't end on a blank line (parser bug).
- Batch mode doesn't provide summary statistics.

Selected Functionality

Overview

We will discuss the implementation details of two key components of the system, the parsing support module & inference module, by looking at examples of haskell code, how they fit together, and how they evolve.

Overview

We will discuss the implementation details of two key components of the system, the parsing support module & inference module, by looking at examples of haskell code, how they fit together, and how they evolve.

For this, we will look at two examples - parsing of model files using parser combinators, and the pitfall of initially specialising the inference functions to single vectors.

Parsing Model Files - Numbers

```haskell
number :: Parser Double
number = do
  sign <- option "" $ string "-"
  int <- many1 digit <?> "integer part"
  dec <- option "" $ do
    void $ char '.'
    frac <- many1 digit
    return $ '.' : frac
  pow <- option "" $ do
    void $ char 'e'
    esign <- option "" $ string "-" <|> string "+"
    num <- many1 digit
    let epart = 'e':(esign ++ num)
    return epart
  let num = read (sign ++ int ++ dec ++ pow) :: Double
  return num
```

Parsing Model Files - Vectors

```haskell
vector :: Parser [Double]
vector = do
  void $ char '['
  eatWhitespace
  numbers <- sepEndBy number (many1 whitespace)
  void (char ']' <?> "vector closing bracket")
  return numbers
```

Parsing Model Files - Matrices

```
matrix :: Parser (Matrix Double, Int, Int)
matrix = do
  void $ char '['
  eatWhitespace
  rows <- sepEndBy vector (many1 whitespace)
  void (char ']' <?> "matrix closing bracket")
  let nRows = length rows
  let nCols = length $ head rows
  return (Mat.fromList nRows nCols (concat rows), nRows,
      nCols)
```

And so on...

Batch Inference - Network Structure

```haskell
type Network = [Layer]

data Layer = Layer {
    inDim :: Int,
    outDim :: Int,
    weights :: Matrix Double,
    biases :: Matrix Double
  }
```

Batch Inference - Infer Single Vector

```
inferSingle :: Network -> Vector Double -> Vector Double
inferSingle net input = getCol 1 $ softLayer
      (foldl reluLayer inpMat nonLinear) final
  where
    inpMat = colVector input
    size = length net
    nonLinear = take (size - 1) net
    final = last net

    inferLayer :: Matrix Double -> Layer -> Matrix Double
    inferLayer inp (Layer _ _ w b) = w * inp + b

    reluLayer :: Matrix Double -> Layer -> Matrix Double
    reluLayer inp lay = relu (inferLayer inp lay)

    softLayer :: Matrix Double -> Layer -> Matrix Double
    softLayer inp lay = softmax (inferLayer inp lay)
```

Batch Inference - Generalising to Batch Inference?

```
inferBatch :: Network -> Matrix Double -> Matrix Double
inferBatch net input = softLayer (foldl reluLayer input
    nonLinear) final
  where
    size = length net
    nonLinear = take (size - 1) net
    final = last net

    inferLayer :: Matrix Double -> Layer -> Matrix Double
    inferLayer inp (Layer _ _ w b) = w * inp + b
    ...
```

Batch Inference - Bias Expansion

```
inferBatch :: Network -> Matrix Double -> Matrix Double
inferBatch net input = softLayer (foldl reluLayer input
    nonLinear) final
  where
    size = length net
    nonLinear = take (size - 1) net
    final = last net

    inputCols = ncols input

        expandCols :: Matrix Double -> Matrix Double
        expandCols mat = foldr (\_ acc -> acc <|> mat) mat
            [1..inputCols]

    inferLayer :: Matrix Double -> Layer -> Matrix Double
    inferLayer inp (Layer _ _ w b) = w * inp + (expandCols b)
    ...
```

Batch Inference - The Culprit

```
softmax :: Matrix Double -> Matrix Double
softmax v = (/ denom) . exp <$> v
     where denom = sum (exp <$> v)
```

Batch Inference - Softmax Generalisation

```
batchSoftmax :: Matrix Double -> Matrix Double
batchSoftmax mat = submatrix 1 rows 1 inputCols (foldr
    folder (fromList rows 1 [1..]) [1..inputCols])
  where
    rows = nrows mat
    folder :: Int -> Matrix Double -> Matrix Double
    folder colIdx acc = colVector ((/ denom col) . exp <$>
        col) <|> acc
      where col = getCol colIdx mat

    denom :: Vector Double -> Double
    denom col = sum (exp <$> col)
```

Plan for Completion

A high-level roadmap for the project's completion is as follows:

- Perform runtime checks on input dimension during inference.
- Runtime checks for interactive mode (for graceful fails).
- Implement the test suite discussed above
- In batched mode, allow output to csv (instead of stdout)
- Allow alternative activation functions for non-output layers, as opposed to assuming ReLU.
- Add testing mode, given batch and expected outputs, report model accuracy.
- Add aggregation mode, use multiple trained models with voting (ensemble method).

Upon delivery of all these tasks, the project will be in its completed state.