**CS4375 Assignment 2**

<https://github.com/carlopizzuto/4375-HW2>

Carlo Pizzuto

cxp200021

# Introduction and Data (5 pt)

In this project, there are two neural networks that need their forward pass functions complete. The first to implement is a Feedforward Neural Network (FFNN), and the second is a Recurrent Neural Network. Both models will be trained for the task of performing a 5-class Sentiment Analysis.

The data being used for this project comes from a set of Yelp reviews. There are three distinct datasets included – Training, Testing, and Validation. Each of the datasets comes in a .json file, with each object (data point) in all datasets having a ‘*text’* variable and ‘*star’* variable. The ‘*text’* (independent) variable is the Yelp review, which will be passed to each Neural Network (NN) to predict the ‘*star’* (target) variable, which ranges from 1 to 5, inclusive.

The **Train** dataset, which will be used to train both Neural Networks, consists of 16,000 data samples, each with its ‘*text’* and ‘*star’* variables. In this dataset, all 5 possible target variables are represented equally. This means that each unique ‘*star’* value has an equal representation in the dataset.

A graph of blue bars

Description automatically generated with medium confidence

Figure 1.1 – distribution of target variable **‘*stars’*** on the **Train** dataset.

The **Test** dataset consists of 800 data samples, each with its ‘*text’* and ‘*star’* variables. In this dataset, only there are only ‘*star’* values of 3, 4, and 5. The ‘*star’* value 3 is the least present, with 20% of values having a 3 ‘*star’* value. 4 and 5 on the other hand each have 40% representation in this dataset.

A graph with blue squares

Description automatically generated with medium confidence

Figure 1.2 – distribution of target variable **‘*stars’*** on the **Test** dataset.

The **Validation** dataset consists of 800 data samples, each with its ‘*text’* and ‘*star’* variables. In this dataset, only there are only ‘*star’* values of 1, 2, and 3. The ‘*star’* value 3 is the least present, with 20% of values having a 3 ‘*star’* value. 1 and 2 on the other hand each have 40%.

A graph of a bar

Description automatically generated with medium confidence

Figure 1.3 – distribution of target variable **‘stars’** on the **Validation** dataset.

# Implementations (45 pt)

## FFNN (20 pt)

A Feed-Forward Neural Network (FFNN) is a type of Neural Network in which the data flows in one direction - from the input to the output – without any feedback loops. It takes the data and moves it form the input layer  through one or more hidden layers to produce an output. Each neuron in the hidden layer applies the linear transformation function  to the input and passes the result through an activation function like Rectified Linear Unit *(ReLU)*. The output of the activation function is then passed to the next hidden layer’s neurons as the input to the linear transformation function. This process continues for all hidden layers, and the last one (output layer) giving the output of the network.

The FFNN model I implemented only had one hidden layer, which uses the ReLU activation function, and an output layer which uses the SoftMax activation function. The SoftMax function is critical to the output, since the output is a category (a star from 1 to 5) and not a continuous number (e.g the price of a house). My task was to implement the forward() function for the model, which initially had the following structure.

def forward(self, input\_vector):

*# [to fill] obtain first hidden layer representation*

*# [to fill] obtain output layer representation*

*# [to fill] obtain probability dist.*

return predicted\_vector

Figure 2.1.1 – Initial forward() function structure of the FFNN model.

To fill in this function, I referred to the model’s \_\_init\_\_() function, which initializes a two-layer Feed-Forward Neural Network. It has h hidden units provided as an argument to the file, a hidden layer with the ReLU activation function, an output layer with the softmax activation function, and uses the negative log-likelihood loss function for training.

def \_\_init\_\_(self, input\_dim, h):

super(FFNN, self).\_\_init\_\_()

self.h = h *# number of hidden units (neurons)*

*#**======== Hidden Layer Start ========*

self.W1 = nn.Linear(input\_dim, h) *# input layer to hidden layer*

self.activation = nn.ReLU() *# relu - hidden layer activation function*

*#======== Hidden Layer End ========*

self.output\_dim = 5 *# number of classes*

*#======== Output Layer Start ========*

self.W2 = nn.Linear(h, self.output\_dim) *# hidden layer to output layer*

self.softmax = nn.LogSoftmax() *# softmax - output layer activation function*

*#======== Output Layer End ========*

self.loss = nn.NLLLoss() *# cross-entropy/**negative log likelihood loss function*

Figure 2.1.2 – FFNN model’s \_\_init\_\_() function.

* **self.h:** number of neurons in the hidden layer.
* **self.W1:** computes the hidden layer’s linear transformation and transforms the data from input\_dim (vocabulary size) dimensions to h dimensions. It is implemented with the PyTorch module, which has the nn.Linear() function.
* **self.activation:** hidden layer’s activation function Rectified Linear Unit (ReLU). It is also implemented with the PyTorch module, which has the nn.ReLU() function. This function returns:
  + 0 if x is less than 0.
  + x if x is greater than or equal to 0.
* **self.W2:** same as self.W1 but transforms the data from h dimensions to 5 (unique star ratings) dimensions.
* **self.softmax:** output layer’s activation function SoftMax. It is also implemented with the PyTorch module, which has the nn.LogSoftmax() function. This function takes an input vector x and computes the softmax probabilities for each element in x. It is defined as:
* **self.loss:** computes the loss, or error, between the predicted output and the actual value. It uses the negative log likelihood loss function, which I will cover in section 3.1 Evaluations.

Given the \_\_init\_\_() function, I could now fill out the forward() function. This function

1. gets an argument *input\_**vector* of size n,
2. applies the linear transformation function from **self.W1** to the input *input\_vector* (size n) and saves the result in *hidden* (size h),
3. passes *hidden* (size h) through the **self.activation** function *ReLU* and saves the result in the same variable *hidden* (size h),
4. applies the linear transformation function from **self.W2** to the input *hidden* (size h), and saves the result in *output* (size 5),
5. applies softmax from **self.softmax** to *output* (size 5) to predict the probabilities of each class and saves it in *predicted*\_*vector* (size 5),
6. returns *predicted\_vector.*

def forward(self, input\_vector):

*# [to fill] obtain first hidden layer representation*

hidden = self.W1(input\_vector)

hidden = self.activation(hidden)

*# [to fill] obtain output layer representation*

output = self.W2(hidden)

*# [to fill] obtain probability dist.*

predicted\_vector = self.softmax(output)

return predicted\_vector

Figure 2.1.3 – Final forward() function for FFNN model.

Besides the \_\_init\_\_() and forward() functions, the FFNN model also contains an additional method compute\_Loss(self, predicted\_vector, gold\_label) which takes in a prediction vector *prediction\_vector* and the actual value vector *gold\_label* and computes the loss with the activation function at self.loss (negative log likelihood loss function).

The file also includes some additional methods, each with their relevant comments.

def make\_vocab(data):

*# Returns:*

*# vocab = A set of strings corresponding to the vocabulary*

def make\_indices(vocab):

*# Returns:*

*# vocab = A set of strings corresponding to the vocabulary including <UNK>*

*# word2index = A dictionary mapping word/token to its index (a number in 0, ..., V-1)*

*# index2word = A dictionary inverting the mapping of word2index*

def convert\_to\_vector\_representation(data, word2index):

*# Returns:*

*# vectorized\_data = A list of pairs (vector representation of input, y)*

def load\_data(train\_data, val\_data, test\_data):

*# Returns:*

*# train\_data = A list of pairs (document, y) from training data*

*# test\_data = A list of pairs (document, y) from test data*

*# valid\_data = A list of pairs (document, y) from validation data*

Figure 2.1.4 – Additional functions present in the FFNN file.

The \_\_main\_\_() function present in the file is used to train the model. It takes in several command line arguments, including the number of hidden dimensions (--hidden\_dim), number of epochs (--epochs), paths to the data (--val\_data, …), and Boolean flags to specify training or inference (my addition, --do\_train, --do\_infer). After parsing the arguments into the *args* Namespace variable (like a python dictionary), the function fixes random seeds and loads and vectorizes the data using the functions in Figure 2.1.4.

After handling the data, the FFNN model is initialized with the arguments input\_dim being the length of vocab variable (*len(vocab)*, with vocab = *make\_vocab(train\_data)*), and h to the argument from the parser (*args.hidden\_dim*). This, under the hood, calls the \_\_init\_\_(input\_dim, h) function from Figure 2.1.2. Then, a PyTorch Stochastic Gradient Descent (SGD) optimizer is initialized with

* the model parameters - which tell the optimizer which parameter o update in training,
* learning rate of 0.01 - which controls the step size during gradient descent,
* momentum of 0.9 - which helps accelerate the optimizer in the relevant direction.

If the do\_train flag is present in the arguments, the function proceeds with training the model. The model is set to training and the data is shuffled randomly for better generalization and, the data is spl

## RNN (25 pt)

# Experiments and Results (45 pt)

## Evaluations (15 pt)

## Results (30 pt)

# Analysis (bonus: 10 pt)

# Conclusion and Other (bonus: 5 pt)