**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 it 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.

* **input\_dim:** size of the input vectors (word embedding dimensions).
* **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.output\_dim:** number of possible output classes.
* **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 LogSoftMax. It is also implemented with the PyTorch’s nn.LogSoftmax() module. This function takes an input vector x and computes the log of 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 to 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. For each epoch in range(args.epochs): The model is set to training mode and the optimizer’s gradients are zeroed. The data is then shuffled randomly and split into minibatches of 16 samples. Next, the model makes predictions and gets the training loss. After this, the model predicts on the validation dataset to get the validation accuracy.

If the do\_infer flag is present, the model chooses 10 samples from the test dataset and for each sample, it makes a prediction and prints it, as well as the actual target variable and the loss for that sample. At the end, it gives an average loss.

## **RNN (25 pt)**

A Recurrent Neural Network (RNN) is a type of neural network designed to process sequential data. It does so by including loops in its architecture, allowing information to be retained across time steps. Unlike FFNNs, RNNs pass data not only from the input to the output, but also back to the previous layers. Each hidden layer at time step t calculates the hidden state using the formula , where is the input at time step t,  is the hidden state from the previous time step, and are the weight matrices, is the bias term, and is an activation function – typically Tanh or Sigmoid.

The hidden state then feeds into the next layer and influences future time steps. The final output at time step t is calculated as , where is the weight matrix for the output layer, is the bias term for the output and is an output specific activation function – SoftMax for classification in our case. My task was to implement the forward() function for this model, which initially had the following structure.

def forward(self, inputs):

*# [to fill] obtain hidden layer representation*

*# [to fill] obtain output layer representations*

*# [to fill] sum over output*

*# [to fill] obtain probability dist.*

return predicted\_vector

Figure 2.2.1 – Initial forward() function for the RNN model

To implement this function, I again referred to the model’s \_\_init\_\_() function. Here, a two-layer RNN model is defined, which has a single RNN layer with h neurons that uses the Tanh activation function. This layer is followed by a linear output layer which uses the LogSoftMax for classification and the negative log-likelihood loss function for training.

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

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

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

self.numOfLayer = 1 *# number of RNN layers stacked together*

*#======== RNN Layer Start ========*

*# input\_dim: size of input vectors*

*# h: number of hidden units*

*# nonlinearity: activation function for hidden layer*

self.rnn = nn.RNN(input\_dim, h, self.numOfLayer, nonlinearity='tanh')

*#======== RNN Layer End ========*

self.output\_dim = 5 *# number of classes for star ratings 1-5*

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

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

self.softmax = nn.LogSoftmax(dim=1) *# log softmax for output probabilities*

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

self.loss = nn.NLLLoss() *# negative log likelihood loss function for training*

Figure 2.2.2 – RNN model’s \_\_init\_\_() function.

* **input\_dim:** size of the input vectors (word embedding dimensions).
* **self.h:** number of neurons in the hidden layer.
* **self.numIfLayer:** number of RNN layers stacked together.
* **self.RNN:** core RNN layer implemented with PyTorch’s nn.RNN module. It takes in the input vector size, the number of hidden units, the number of stacked RNN layers and the nonlinearity function Tanh – which maps the values to the range [-1, 1].
* **self.output\_dim:** number of possible output classes (5).
* **self.W:** linear transformation output layer that maps from hidden state dimensions (h) to output dimension (output\_dim), implemented using PyTorch’s nn.Linear module.
* **self.softmax:** output layer’s activation function LogSoftMax. It is also implemented with the PyTorch’s nn.LogSoftmax() module. This function takes an input vector x and computes the log of 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 RNN transformation from **self.RNN** (including the Tanh activation function) to the input input\_vector (size n) and saves the result in hidden (size h),
3. extracts the final hidden state at hidden[-1] (last element) and saves it in the same variable hidden,
4. passes hidden (size h) through the linear transformation layer **self.W** and saves the result in the variable 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, inputs):

*# [to fill] obtain hidden layer representation*

\_, hidden = self.rnn(inputs)

*# [to fill] obtain output layer representations*

hidden = hidden[-1]

*# [to fill] sum over output*

output = self.W(hidden)

*# [to fill] obtain probability dist.*

predicted\_vector = self.softmax(output)

return predicted\_vector

Figure 2.2.3 – Final forward() function of the RNN model

Besides the \_\_init\_\_() and forward() functions, the RNN model also contains an additional method compute\_Loss(self, predicted\_vector, gold\_label), which is the same as the FFNN model – computes the loss of the predicted vector using the negative log likelihood loss function.

The \_\_main\_\_() function of this model’s file has a similar structure to the FFNN file. It parses the arguments from the command line (same as the FFNN), loads the data with the load\_data() function and fixes the random seeds.

Then, the RNN model is initialized with input\_dim as 50 and h as the augment from the command line (args.hidden\_dim). This, under the hood, calls the model’s \_\_init\_\_(input\_dim, h) function from Figure 2.2.2. Then a PyTorch Adam optimizer is initialized with:

* the model parameters - which tell the optimizer which parameter to update in training,
* learning rate of 0.01 - which controls the step size during gradient descent.

Adam, like SGD, is used to update the network weights during training. However, Adam uses momentum while tracking moving averages of past gradients, while SGD updates parameters based only on the current gradient.

After some other variable initialization and logging, the function proceeds to start the training loop, taking the data in minibatches of 16 random samples. It (like FFNN) also does a validation step after each training iteration. Unlike FFNN, this loop also includes a stopping condition: if the validation accuracy of this iteration is less than the last one AND the training accuracy for this iteration is higher than the last one. For experimentation, I later changes this to stop based on the difference between this epoch’s validation accuracy and the last one.

# **Experiments and Results (45 pt)**

## **Evaluations (15 pt)**

The Feed-Forward Neural Network (FFNN) is evaluated using both the accuracy and loss metrics. For each training epoch, the model processes data in minibatches of 16 samples and tracks the number of correct predictions and total samples. These tracked values are then used to compute the accuracy – correct / total. The loss is computed using the Negative Log Likelihood (NLL) loss function after the LogSoftMax activation, implementing cross-entropy loss. After each epoch, the model performs validation using the same metrics (but not updating the weights). This validation step helps monitor for overfitting and gives a sense of the model’s generalization performance.

The Recurrent Neural Network (RNN) uses a similar evaluation method but also implements an early stop feature. Like the FFNN, this model processes the training data in minibatches of 16 samples tracks accuracy (correct / total) as well as the NLL loss. However, the RNN adds an overfitting prevention mechanism, which compares the current validation accuracy with the previous epoch’s validation accuracy. If the validation accuracy decreases but training increases, the training stops to prevent overfitting. This way, the best model is the one from the previous epoch, which has the highest validation accuracy.

The Negative Log Likelihood (NLL) loss function used in both models is commonly used for classification tasks. It quantifies the difference between the predicted probability distribution and the true labels, and penalizes the model based on the negative log of the probability assigned to the correct class. It is defined as

Where C is the number of classes (5 in our case),  is the binary indicator (0 or 1) if class i is the correct classification and is the predicted probability for class i.

## **Results (30 pt)**

**3.2.1 FFNN**

**1 epoch with 10 hidden dimensions:**

Total training time (with validation): **6.8 seconds**

epoch train\_acc val\_acc train\_loss val\_loss

1 0.3946 0.3150 1.4309 1.3024

With 10 hidden dimensions and 1 epoch, the model reaches a training accuracy of 0.3946 and validation accuracy of 0.3150. Loss remains high, meaning limited learning capacity in a single pass. Compared to later models, this configuration lacks the complexity needed for better accuracy.

**1 epoch with 20 hidden dimensions:**

Total training time (with validation): **10 seconds**

epoch train\_acc val\_acc train\_loss val\_loss

1 0.4122 0.5212 1.3040 1.3161

Doubling the hidden dimensions to 20 improves accuracy – training accuracy reaches 0.4122 and validation accuracy jumps to 0.5212. This means that a larger hidden layer better captures patters, although it takes more time (around 3.2 seconds more per epoch).

**5 epochs with 5 hidden dimensions:**

Total training time (with validation): **36.5 seconds**

epoch train\_acc val\_acc train\_loss val\_loss

1 0.3096 0.4537 1.2933 1.5037

2 0.4405 0.4313 1.7229 1.3693

3 0.4893 0.5637 1.2074 0.8814

4 0.5383 0.4838 1.0397 1.4532

5 0.5636 0.5475 1.0074 1.4572

Training for 5 epochs with 5 hidden dimensions improves accuracy – training peaks at 0.5635 (epoch 5) and validation peaks at 0.5637 (epoch 3). However, fluctuations in validation accuracy indicate potential overfitting. Compared to single-epoch models, multiple epochs improve learning, although low dimensions limit generalization.

**5 epochs with 20 hidden dimensions:**

Total training time (with validation): **59.1 seconds**

epoch train\_acc val\_acc train\_loss val\_loss

1 0.4122 0.5212 1.3040 1.3161

2 0.5091 0.4487 1.4770 1.1986

3 0.5454 0.5775 0.9599 1.0839

4 0.5734 0.5062 0.8694 1.1584

5 0.6024 0.6025 0.7730 1.0902

With 5 epochs and 20 hidden dimensions, the model performs best – 0.6024 training and 0.6025 validation accuracy at the last epoch. Validation loss stabilizes, but slight increases could be a hint of overfitting. Compared to prior models, this setup shows that both more epochs and higher hidden dimensions improve model learning.

**3.2.2 RNN**

**10 epochs with 32 hidden dimensions:**

Total training time (with validation): **292.2 seconds**

Early stop: **epoch 9** was the last epoch.

epoch train\_acc val\_acc train\_loss val\_loss

1 0.2202 0.2213 1.5505 1.5505

2 0.2275 0.2137 1.5674 1.5674

3 0.2385 0.1913 1.5590 1.5590

4 0.2384 0.2625 1.5887 1.5887

5 0.2344 0.1638 1.5449 1.5449

6 0.2406 0.2325 1.5609 1.5609

7 0.2271 0.2375 1.6722 1.6722

8 0.2343 0.3800 1.5950 1.5950

9 0.2348 0.2350 1.5936 1.5936

10 0.0000 0.0000 0.0000 0.0000

Training the RNN for 10 epochs with 32 hidden dimensions caused an early stop at epoch 9, with moderate accuracy improvement but high fluctuation. Training accuracy pealed at 0.02406 and validation accuracy peaked at 0.3800 on epoch 8, indicating some learning but poor generalization.

**10 epochs with 20 hidden dimensions:**

Total training time (with validation): **349.6 seconds**

No early stop

epoch train\_acc val\_acc train\_loss val\_loss

1 0.2447 0.2925 1.5997 1.5997

2 0.2517 0.2562 1.4653 1.4653

3 0.2517 0.2200 1.4992 1.4992

4 0.2652 0.2250 1.5286 1.5286

5 0.2662 0.2587 1.6432 1.6432

6 0.2694 0.2162 1.6273 1.6273

7 0.2610 0.3050 1.7699 1.7699

8 0.2552 0.2013 1.4674 1.4674

9 0.2663 0.2425 1.6716 1.6716

10 0.2731 0.3425 1.5201 1.5201

With 20 dimensions, training accuracy improves steadily to 0.2731, with a validation accuracy reaching 0.3425 by epoch 10. Loss values fluctuate, showing limited overfitting with modest generalization. Compared to the 32-dimension model, accuracy becomes more stable, even though training time increases. This suggests that fewer units allow a steadier leaning at the cost of lower convergence.

**10 epochs with 50 hidden dimensions:**

Total training time (with validation): **185.1 seconds**

Early stop: **epoch 6** was the last epoch.

epoch train\_acc val\_acc train\_loss val\_loss

1 0.2176 0.1425 1.6318 1.6318

2 0.2396 0.3450 1.6746 1.6746

3 0.2312 0.2188 1.4959 1.4959

4 0.2294 0.2325 1.7293 1.7293

5 0.2299 0.1375 1.6562 1.6562

6 0.0000 0.0000 0.0000 0.0000

Increasing the hidden dimensions to 50 caused the model to top after epoch 5, achieving the highest validation accuracy of 0.3450 at epoch 2. Loss values fluctuate widely, which indicates inconsistent learning. Compared to the previous models, 50 dimensions seems to increase instability, meaning that the optimal number may be lower.

**10** epochs with **5** hidden dimensions:

Total training time (with validation): **204 seconds**

Early stop: **epoch 6** was the last epoch.

epoch train\_acc val\_acc train\_loss val\_loss

1 0.2531 0.1525 1.6523 1.6523

2 0.2536 0.1537 1.5233 1.5233

3 0.2619 0.1787 1.6382 1.6382

4 0.2557 0.0887 1.6325 1.6325

5 0.2637 0.1825 1.4195 1.4195

6 0.3013 0.0013 1.6739 1.6739

7 0.0000 0.0000 0.0000 0.0000

Using 5 hidden dimensions results in a low accuracy, with training and validation peaking at 0.3013 (epoch 6) and 0.1825 (epoch 5) respectively. The big difference in training and validation accuracy on epoch 6 also suggests extreme overfitting. Compared to models with more hidden dimensions, this model’s low accuracy reflects a lack of capacity for pattern recognition.

RNN – **10** epochs with **100** hidden dimensions:

Total training time (with validation): **136.7 seconds**

Early stop: **epoch 3** was the last epoch.

epoch train\_acc val\_acc train\_loss val\_loss

1 0.2117 0.1150 1.6706 1.6706

2 0.2278 0.2900 1.9196 1.9196

3 0.2324 0.1062 1.5170 1.5170

4 0.0000 0.0000 0.0000 0.0000

With 100 hidden dimensions, training accuracy remains low with a peak of 0.2324. Validation accuracy peaks at 0.2900 on epoch 3, which drops significantly on the next epoch. Compared to previous models, very high dimensions in this setup likely cause overfitting and poor generalization.