CS6375 Assignment 1

https://github.com/NellaCu/CS6375-Assignment-1

Allen Cui
Axc180094

1 Introduction and Data

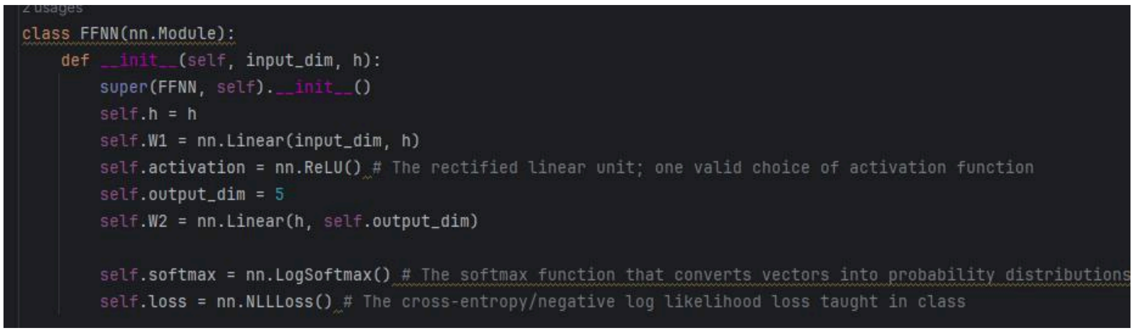
The project is to learn about two models that have been learned throughout the course and to compare them through parameters and a pre-defined dataset using a 5-class Sentiment Analysis task. The two models that are being focused on in this assignment are FFNN and RNN. The dataset that is being used and validated involves multiple reviews through the form of Yelp reviews where after training, is able to train the model into predicting the rating based on their comment. Afterwards, we would then compare the two and analyze the performance and differences.

The data comprise JSON files that have multiple reviews ranging to around 8000 to both train and validate the results. These JSON files will end up being taken in differently to test out the impact of the parameters (primarily the Epochs and Hidden dim).

2 Implementations

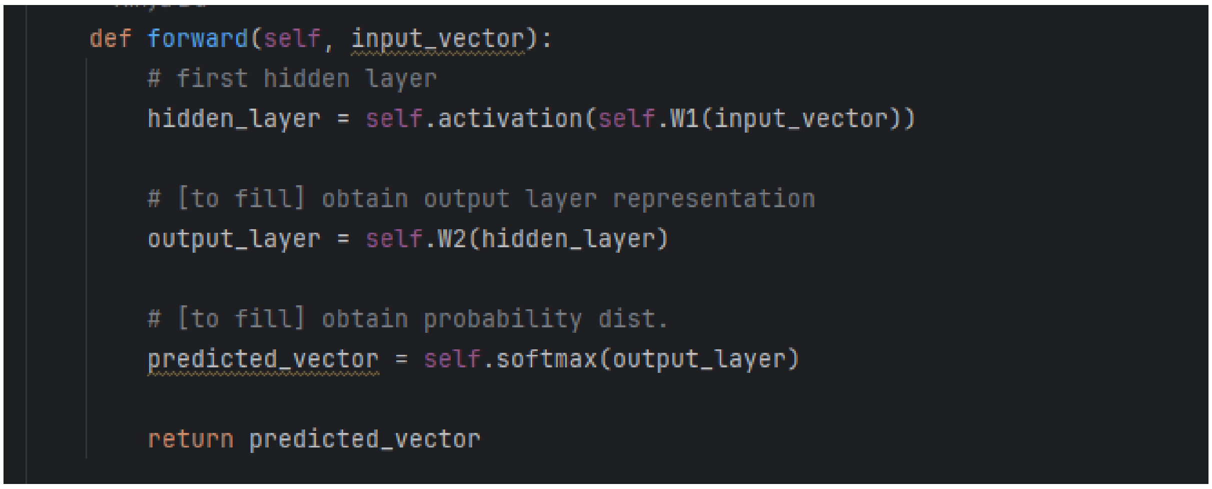
2.1 FFNN

The FFNN.py implemented all the functionality except the forward function so information can't travel from one layer to another. In class, we learned that the Neural Network comprises mostly three main layers, the input layer, the hidden layer, and the output layer. This formatting can be assisted with using library functions from the PyTorch library, allowing us to translate the inputs into layers as shown through figure 1. The implementation of most of the FFNN is already done so the only thing that had to be implemented was the forward function. By applying what we know about Neural networks we can see that in figure 2 that it takes an input vector that we would need to turn into a layer (in this case a hidden layer). This can be done with the already created functions in figure 1 where we can convert the input vector into a linear transformation and apply an activation function over it. This will get us our hidden layer which we can then apply the transition from the hidden layer to the output layer. Lastly, we have to softmax it which will give us a probability distribution.



```
class FFNN(nn.Module):
    def __init__(self, input_dim, h):
        super(FFNN, self).__init__()
        self.h = h
        self.W1 = nn.Linear(input_dim, h)
        self.activation = nn.ReLU() # The rectified linear unit; one valid choice of activation function
        self.output_dim = 5
        self.W2 = nn.Linear(h, self.output_dim)

        self.softmax = nn.LogSoftmax() # The softmax function that converts vectors into probability distributions
        self.loss = nn.NLLLoss() # The cross-entropy/negative log likelihood loss taught in class
```

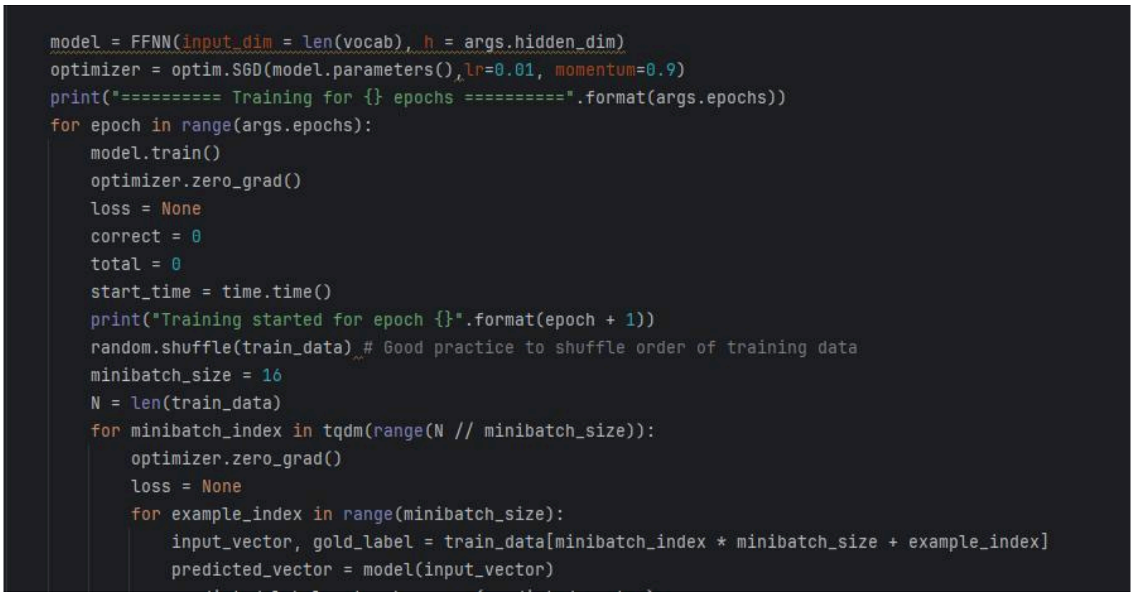
Figure 1: Initialization



```python
    def forward(self, input_vector):
        # first hidden layer
        hidden_layer = self.activation(self.W1(input_vector))

        # [to fill] obtain output layer representation
        output_layer = self.W2(hidden_layer)

        # [to fill] obtain probability dist.
        predicted_vector = self.softmax(output_layer)

        return predicted_vector
```

Figure 2: Forward Function for FFNN

Additionally the PyTorch library allows us to train our data (which we convert to a vector representation) as seen in figure 3. By using the optimizer (a function that does stochastic gradient descent), we can train our neural network and back propagate it. This structure is repeated in the validation function except we have removed the additional bulk of randomizing, changing weights, etc.

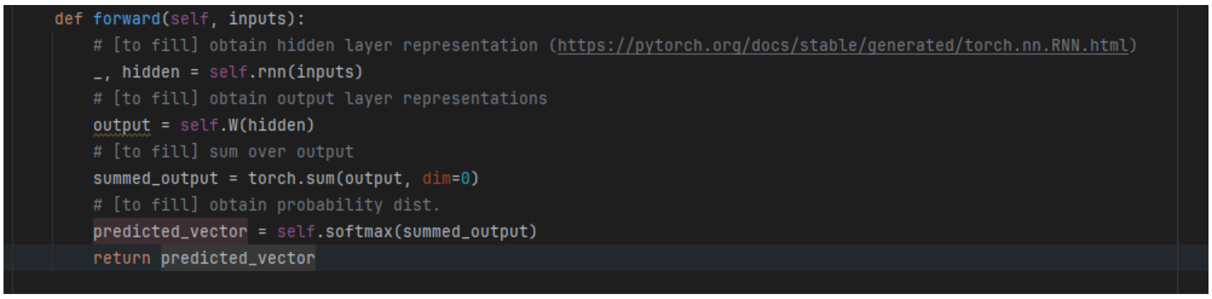


```python
model = FFNN(input_dim = len(vocab), h = args.hidden_dim)
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
print("========== Training for {} epochs ==========".format(args.epochs))
for epoch in range(args.epochs):
    model.train()
    optimizer.zero_grad()
    loss = None
    correct = 0
    total = 0
    start_time = time.time()
    print("Training started for epoch {}".format(epoch + 1))
    random.shuffle(train_data) # Good practice to shuffle order of training data
    minibatch_size = 16
    N = len(train_data)
    for minibatch_index in tqdm(range(N // minibatch_size)):
        optimizer.zero_grad()
        loss = None
        for example_index in range(minibatch_size):
            input_vector, gold_label = train_data[minibatch_index * minibatch_size + example_index]
            predicted_vector = model(input_vector)
```

Figure 3: FFNN Main Function

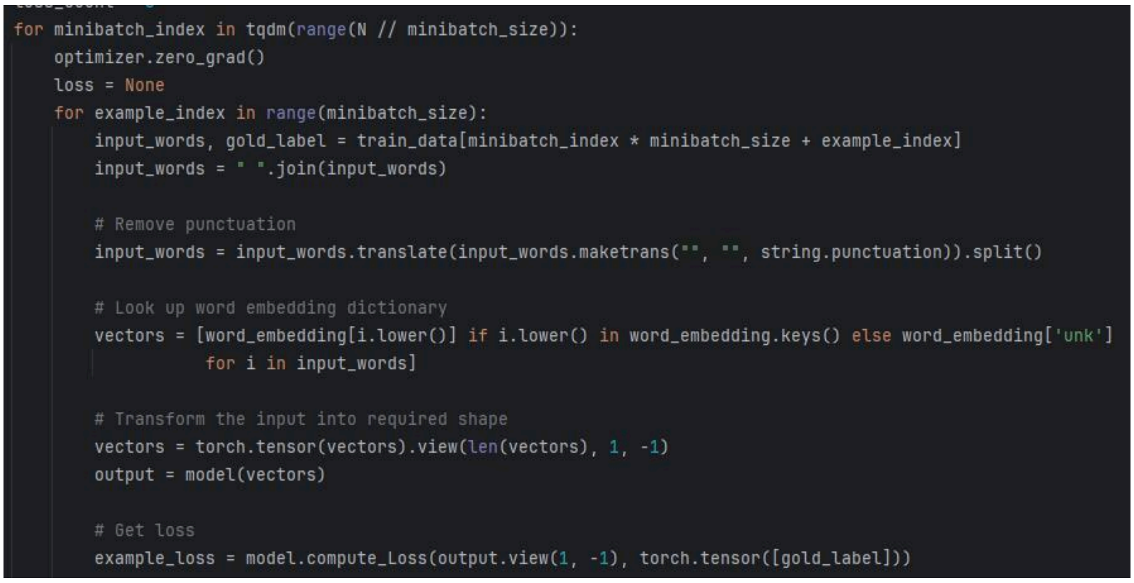
2.2 RNN

With the inclusion of RNN, we just have to make the data influence future neurons and propagate. Some differences of the FFNN is that we have to utilize a new function from the PyTorch library that allows us to take a vector input and turn it to a hidden layer. As shown through the set up in figure 4, we are able to add another input that is able to influence future neurons. Additionally, we are also utilizing a word embedding dictionary to apply meaning to the input values in figure 5. Lastly, we do the same methods for FFNN as the RNN coding is already implemented (figure 6). In order to record more data, the stopping implementation when over-fitting is ”assumed” to happen has been removed.



```
def forward(self, inputs):
    # [to fill] obtain hidden layer representation (https://pytorch.org/docs/stable/generated/torch.nn.RNN.html)
    _, hidden = self.rnn(inputs)
    # [to fill] obtain output layer representations
    output = self.W(hidden)
    # [to fill] sum over output
    summed_output = torch.sum(output, dim=0)
    # [to fill] obtain probability dist.
    predicted_vector = self.softmax(summed_output)
    return predicted_vector
```

Figure 4: Initialization for RNN



```
for minibatch_index in tqdm(range(N // minibatch_size)):
    optimizer.zero_grad()
    loss = None
    for example_index in range(minibatch_size):
        input_words, gold_label = train_data[minibatch_index * minibatch_size + example_index]
        input_words = " ".join(input_words)

        # Remove punctuation
        input_words = input_words.translate(input_words.maketrans("", "", string.punctuation)).split()

        # Look up word embedding dictionary
        vectors = [word_embedding[i.lower()] if i.lower() in word_embedding.keys() else word_embedding['unk']
                   for i in input_words]

        # Transform the input into required shape
        vectors = torch.tensor(vectors).view(len(vectors), 1, -1)
        output = model(vectors)

        # Get loss
        example_loss = model.compute_Loss(output.view(1, -1), torch.tensor([gold_label]))
```

Figure 5: Word Embedding Implementation

3 Experiments and Results

We evaluated the models by changing the hidden-dim and the epochs to see if it affects the training and validation accuracy. Also, there was some implementation to find out training loss for better evaluation of the model. This is then going to be compared to see if a model (FFNN or RNN) performed better. The RNN code has been adjusted to avoid stopping in order to record more epochs at the cost of overfitting for training loss specifically.

The results in testing the parameters of increasing epochs and hidden dimensions, there are some patterns that are exhibited. By modifying the values, FFNN did not show a statistical difference of hidden dimensions, but the epochs showed better training accuracy at the end. However, the validation did not increase by a margin that is notable.

RNN did not show any major change of accuracy of both training and validation despite changing the variations, even having more epochs did not massively change the training accuracy overall.

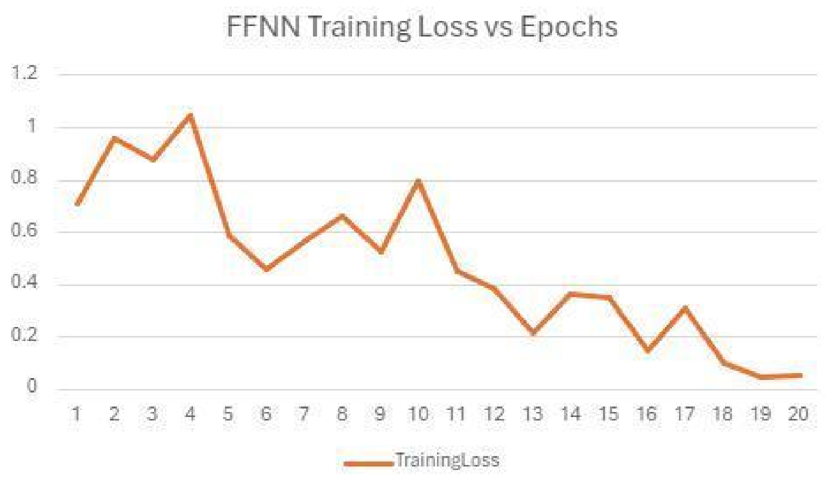


Figure 7: FFNN Training Loss vs Epochs

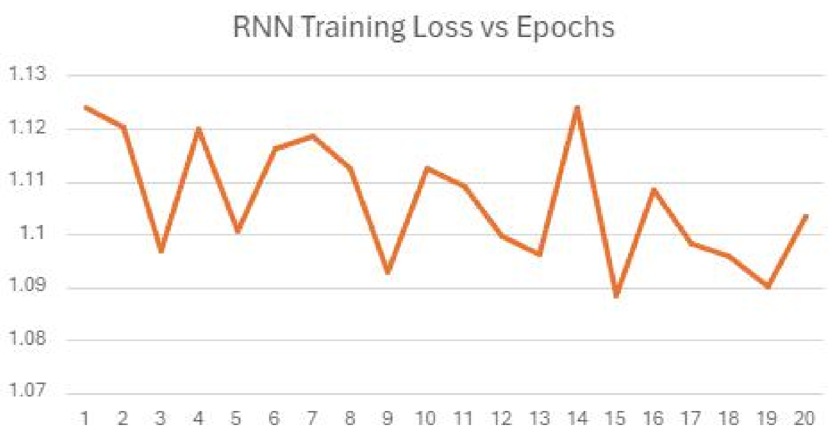


Figure 8: RNN Training Loss vs Epochs

As displayed in figure 7 and figure 8, FFNN showed way more drop of training loss than compared to RNN.

Additionally we have also added for illustration purposes of changing Hidden and Epochs in correlation with Validation Accuracy for both FFNN and RNN models. We do see that RNN did have a significant drop on 30 Epochs, but this is simply due to overfitting and modifications would realign itself with the other data points.

Hidden Layer Size (Epoch 10)	(FFNN)Validation Accuracy	(RNN)Validation Accuracy
8	0.5875	0.41875(Overfit, Epoch 2)
16	0.6025	0.44625(Overfit, Epoch 2)
24	0.59625	0.4225(Overfit, Epoch 2)

Epochs (Hidden dim 16)	(FFNN)Validation Accuracy	(RNN)Validation Accuracy
10	0.6025	0.44625(Overfit, Epoch 2)
20	0.585	0.45375(Overfit, Epoch 6)
30	0.59	0.35125(Overfit, Epoch 2)

4 Analysis

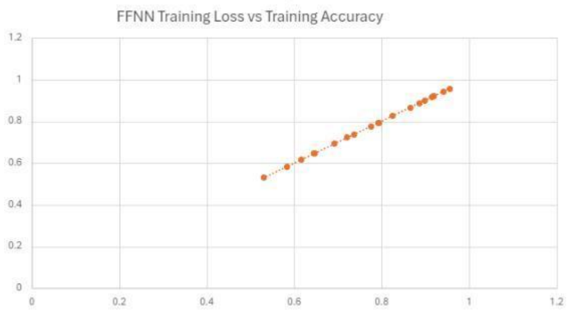


Figure 9: FFNN Training Loss vs Training Accuracy

When evaluating both figure 7 and 8 we can notice that the RNN is more stable than the FFNN. While FFNN experiences an approximately constant growth in figure 9. However despite having a reduction of training loss, there was not a noticeable increase of precision in the validation. Thus, this increase of training accuracy shows that we are most likely experiencing over fitting. Not only was the accuracy for validation bad for FFNN, the RNN also experienced low validation accuracy.

So that means that our code or how we made our neuron network was initially not strong for this particular dataset. Some updates we could make is by changing our activation function or even adding more hidden layers to make it more complex.

5 Conclusion and Others

In short, RNN is more stable than FFNN in regards to the dataset, showing very slow drops to training loss than the latter even under modifications of both the hidden dimensions and epochs. However, both of these did not show significant difference or strong validation accuracy.

Feedback: This was a good learning experience and a good assignment. It would be nice to have clarification of what we should ”expect” on the dataset to see if our data is correct.