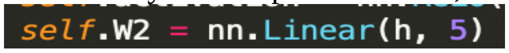
CS5740 NLP Project3 Report

Boliang Yang(by283), Jinwei Shen(js3559), Shenghua Li(sl3293)

1. Fix of 4 errors of FFNN model

This part covers the four errors (one was fixed by instructors) that we found and fixed.

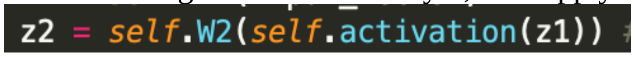
Error 1: at line23, self.W2 = nn.Linear(h, h)
The original output dimension is wrong. Since we are doing a 5-class classification task, the output linear classifier should have dimensionality 5 as output dimension, so we fix it as:



```
self.W2 = nn.Linear(h, 5)
```

Figure 1 code snippet for error 1

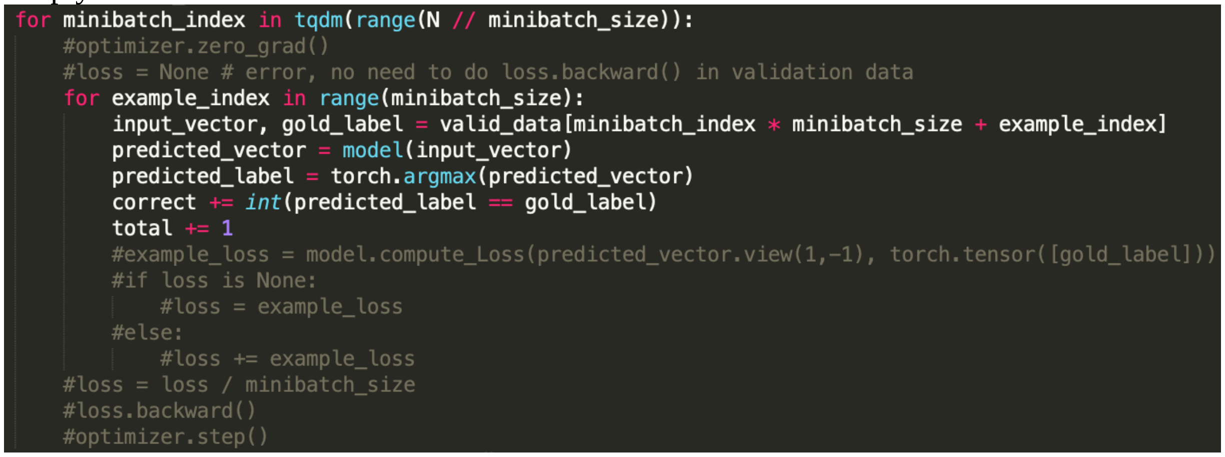
Error 2: at line 34, z2 = self.W2(z1)
Since the activation layer should be right after the first layer, then apply to self.W2, we fix it as:



```
z2 = self.W2(self.activation(z1))
```

Figure 2 code snippet for error 2

Error 3: at line 131 and line 140-146, calculating the loss of validation data and update the weights by backward propagation.
Since validation data is used to test the model, then it is unnecessary to update the weights; so we simply delete the related lines.



```
for minibatch_index in tqdm(range(N // minibatch_size)):
    #optimizer.zero_grad()
    #loss = None # error, no need to do loss.backward() in validation data
    for example_index in range(minibatch_size):
        input_vector, gold_label = valid_data[minibatch_index * minibatch_size + example_index]
        predicted_vector = model(input_vector)
        predicted_label = torch.argmax(predicted_vector)
        correct += int(predicted_label == gold_label)
        total += 1
        #example_loss = model.compute_Loss(predicted_vector.view(1,-1), torch.tensor([gold_label]))
        #if loss is None:
            #loss = example_loss
        #else:
            #loss += example_loss
    #loss = loss / minibatch_size
    #loss.backward()
    #optimizer.step()
```

Figure 3 code snippet for weights' update

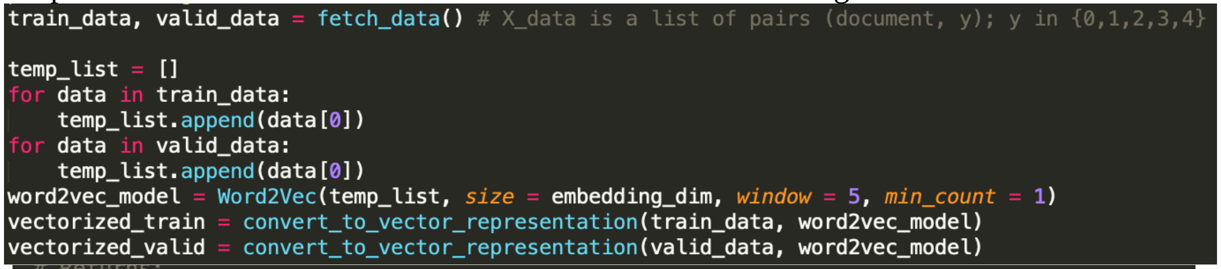
Error 4 (fixed by instructors): Line 105 (of original codes): the computation of the loss should be accumulated for every training data. In the original codes, the loss did not get accumulated, and it was just simply the loss of the final single input data.
So in the fixed version, loss is computed by '+=' expression, thus the computation of loss is accumulated by every sample. Then we can average the loss and do the backpropagation.

2. Implementation of RNN model

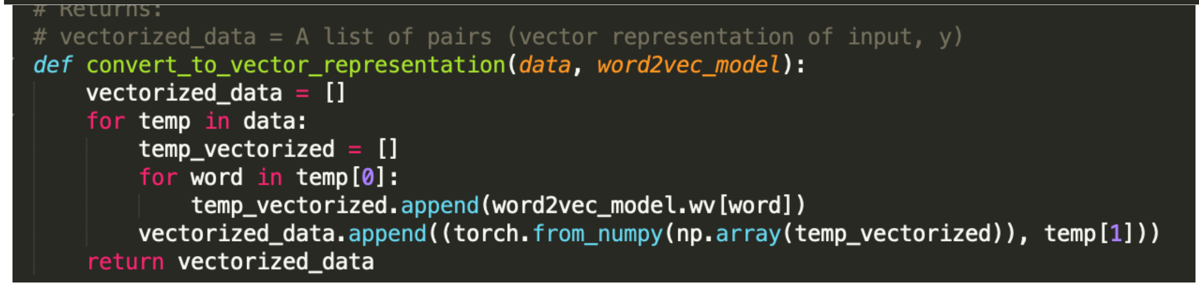
In this section, we will go through how to design and implement an RNN model for our sentiment analysis task.

(1) Representation:

The FFNN model used the bag of words representation, which is a relatively simple representation. In our RNN model, instead of using the bag of word representation for fair comparison with FFNN, we introduced the more famous and efficient Word2Vec representation [1], which is to assign the input to vectors using pretrained word embeddings. Using word-embedding can also take advantage of the expressiveness of first matrix in the simple RNN. So it is reasonable to use a famous word-embedding like Word2Vec.



```
train_data, valid_data = fetch_data() # X_data is a list of pairs (document, y); y in {0,1,2,3,4}

temp_list = []
for data in train_data:
    temp_list.append(data[0])
for data in valid_data:
    temp_list.append(data[0])
word2vec_model = Word2Vec(temp_list, size = embedding_dim, window = 5, min_count = 1)
vectorized_train = convert_to_vector_representation(train_data, word2vec_model)
vectorized_valid = convert_to_vector_representation(valid_data, word2vec_model)
```

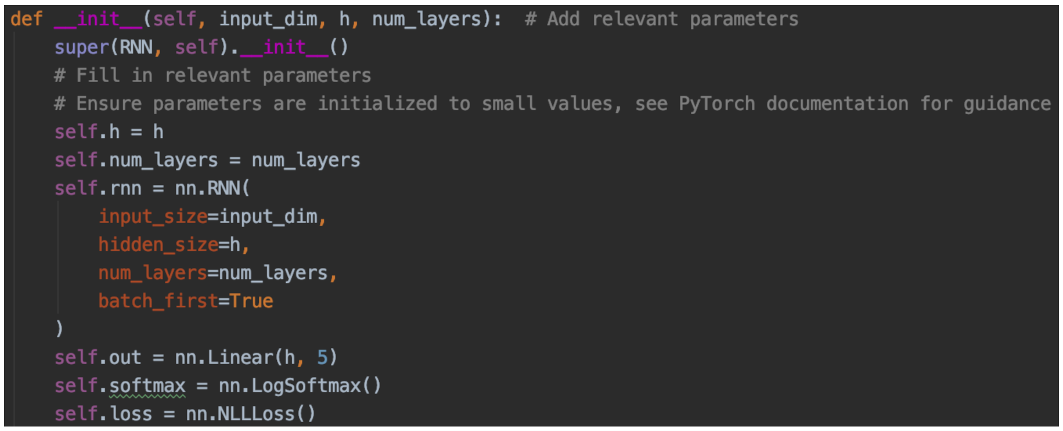


```
# Returns:
# vectorized_data = A list of pairs (vector representation of input, y)
def convert_to_vector_representation(data, word2vec_model):
    vectorized_data = []
    for temp in data:
        temp_vectorized = []
        for word in temp[0]:
            temp_vectorized.append(word2vec_model.wv[word])
        vectorized_data.append((torch.from_numpy(np.array(temp_vectorized)), temp[1]))
    return vectorized_data
```

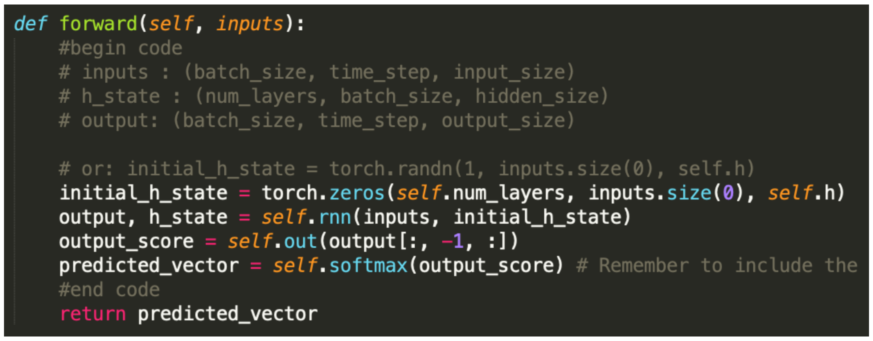
Figure 4 codes for obtaining data and give the representation

(2) Initialization:

For the parameters, they are mainly declared and initialized inside the RNN class __init__ and forward methods. In order to train the model mainly from our dataset, we initialized the weights to zeros at the beginning of training part (see (3) below: optimizer.zero_grad()). We also initialized the first hidden state as zeros in the forward method. Initializing weights to 0 is a reasonable choice as mentioned in class.



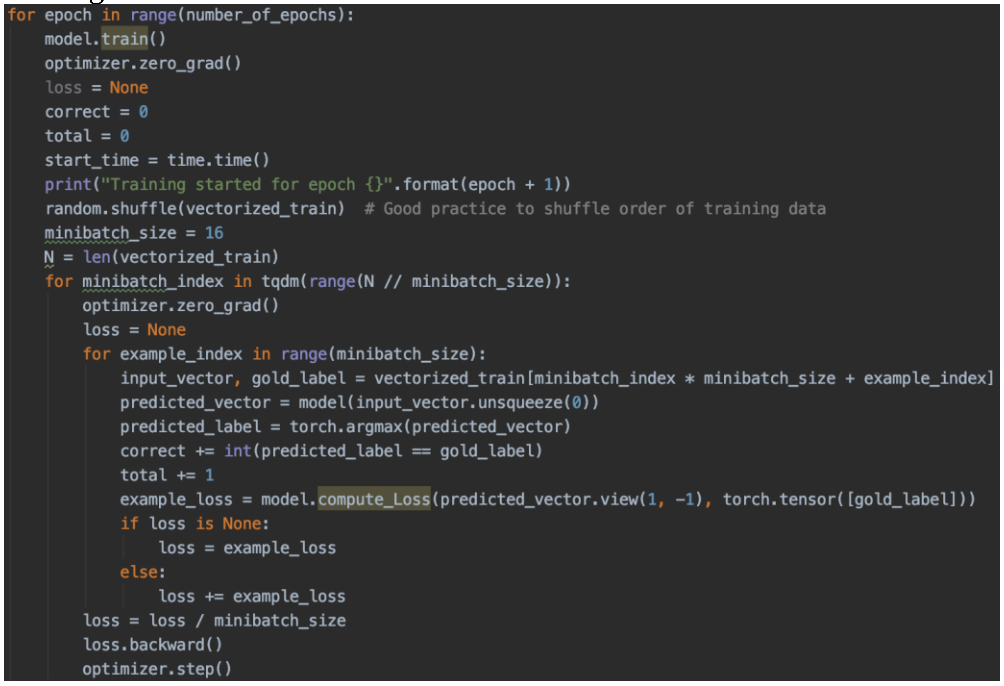
```
def __init__(self, input_dim, h, num_layers):  # Add relevant parameters
    super(RNN, self).__init__()
    # Fill in relevant parameters
    # Ensure parameters are initialized to small values, see PyTorch documentation for guidance
    self.h = h
    self.num_layers = num_layers
    self.rnn = nn.RNN(
        input_size=input_dim,
        hidden_size=h,
        num_layers=num_layers,
        batch_first=True
    )
    self.out = nn.Linear(h, 5)
    self.softmax = nn.LogSoftmax()
    self.loss = nn.NLLLoss()
```



```python
def forward(self, inputs):
    #begin code
    # inputs : (batch_size, time_step, input_size)
    # h_state : (num_layers, batch_size, hidden_size)
    # output: (batch_size, time_step, output_size)

    # or: initial_h_state = torch.randn(1, inputs.size(0), self.h)
    initial_h_state = torch.zeros(self.num_layers, inputs.size(0), self.h)
    output, h_state = self.rnn(inputs, initial_h_state)
    output_score = self.out(output[:, -1, :])
    predicted_vector = self.softmax(output_score) # Remember to include the
    #end code
    return predicted_vector
```

Figure 5 codes for initialization and declaration

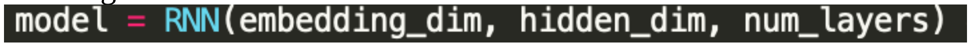
(3) Training:
We shuffled the training data and then divided the processed training sets (word-embedding based representation) into mini-batches. Then we used the training part as in ffnn.py: we still use NLLLoss function as loss computation method here, so every predicted result will be computed a loss related to the label, then we took an average of the loss over the batch size. After that we used the backward function to calculate the gradients and used optimizer to update the weights.



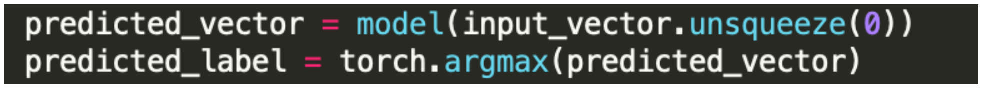
```python
for epoch in range(number_of_epochs):
    model.train()
    optimizer.zero_grad()
    loss = None
    correct = 0
    total = 0
    start_time = time.time()
    print("Training started for epoch {}".format(epoch + 1))
    random.shuffle(vectorized_train)  # Good practice to shuffle order of training data
    minibatch_size = 16
    N = len(vectorized_train)
    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 = vectorized_train[minibatch_index * minibatch_size + example_index]
            predicted_vector = model(input_vector.unsqueeze(0))
            predicted_label = torch.argmax(predicted_vector)
            correct += int(predicted_label == gold_label)
            total += 1
            example_loss = model.compute_Loss(predicted_vector.view(1, -1), torch.tensor([gold_label]))
            if loss is None:
                loss = example_loss
            else:
                loss += example_loss
        loss = loss / minibatch_size
        loss.backward()
        optimizer.step()
```

Figure 6 codes for training

(4) Model:
We have defined the __init__ method to initialize the RNN model in (2). And for each input vector, we performed the unsqueeze(0) to insert a dummy dimension batch_size in the first dimension (because we declare “batch_first = True”, we need batch_size as our first dimension). Then we calculated a predicted output vector from the RNN model. The codes are shown in figure 6.



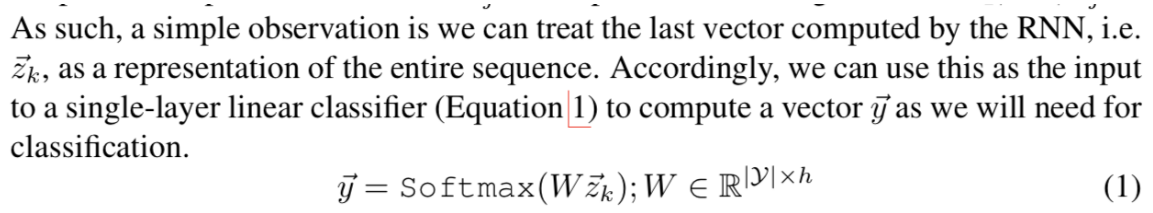
```python
model = RNN(embedding_dim, hidden_dim, num_layers)
```



```
predicted_vector = model(input_vector.unsqueeze(0))
predicted_label = torch.argmax(predicted_vector)
```

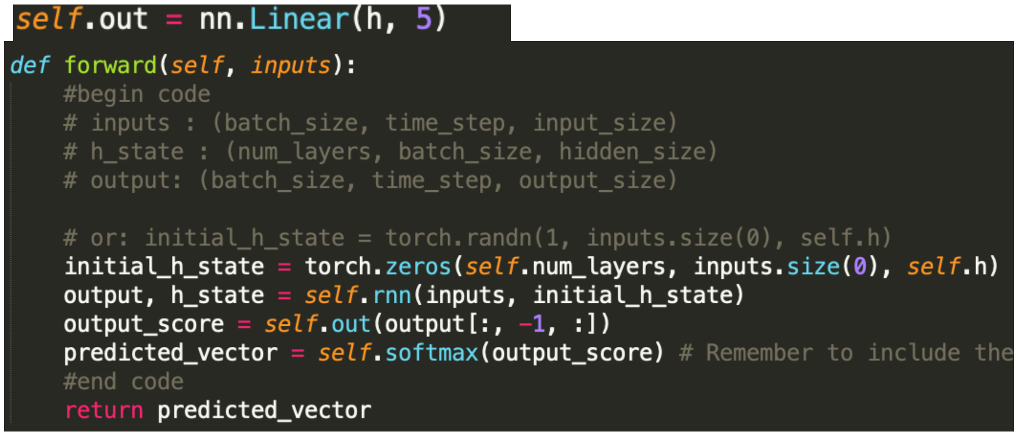
Figure 7 codes for model

(5) Linear Classifier:
We defined the output linear classifier as self.out in RNN declaration. Then inside the forward method, we used the output linear classifier to classify the last outputs of the RNN. As we can see in the figure 7, the second dimension of the output is time_step. We then can use output[:, -1, :] to get the last output of this RNN. The way to use the last vector computed by the RNN as the input to output linear classifier is mentioned by page 6 of this assignment as the screenshot below.



As such, a simple observation is we can treat the last vector computed by the RNN, i.e. $\vec{z}_k$, as a representation of the entire sequence. Accordingly, we can use this as the input to a single-layer linear classifier (Equation 1) to compute a vector $\vec{y}$ as we will need for classification.

$$\vec{y} = \texttt{Softmax}(W\vec{z}_k); W \in \mathbb{R}^{|\mathcal{Y}| \times h} \tag{1}$$

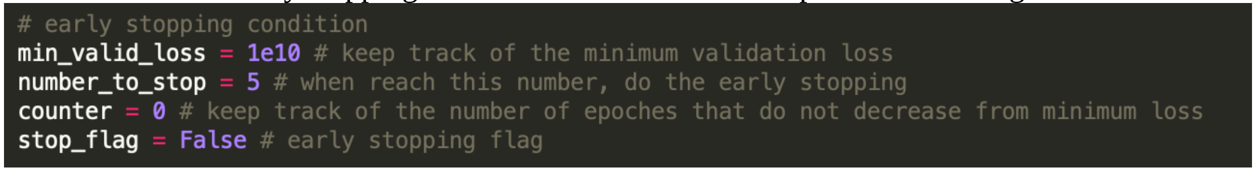


```
self.out = nn.Linear(h, 5)
```

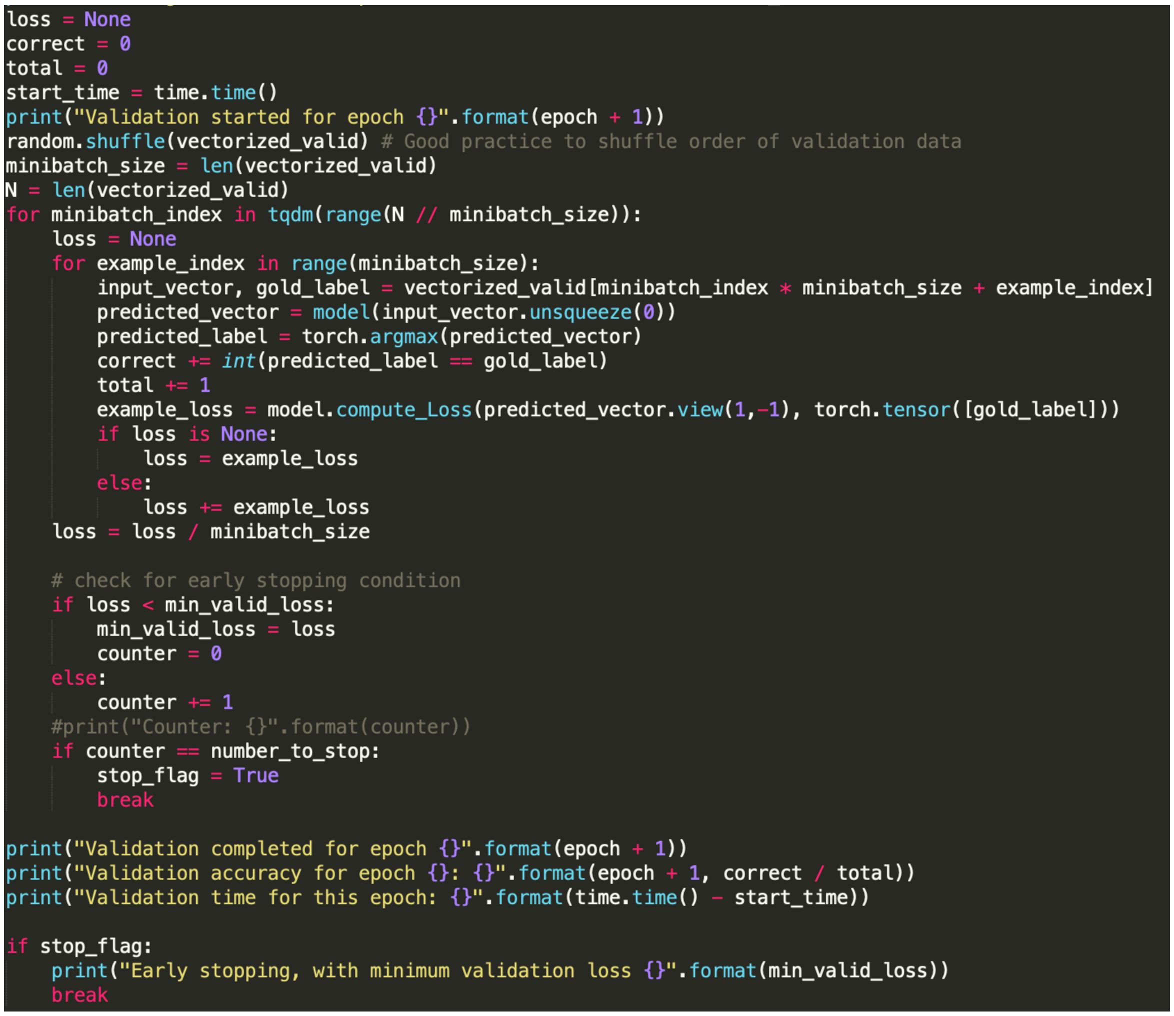
```
def forward(self, inputs):
    #begin code
    # inputs : (batch_size, time_step, input_size)
    # h_state : (num_layers, batch_size, hidden_size)
    # output: (batch_size, time_step, output_size)

    # or: initial_h_state = torch.randn(1, inputs.size(0), self.h)
    initial_h_state = torch.zeros(self.num_layers, inputs.size(0), self.h)
    output, h_state = self.rnn(inputs, initial_h_state)
    output_score = self.out(output[:, -1, :])
    predicted_vector = self.softmax(output_score) # Remember to include the
    #end code
    return predicted_vector
```

Figure 8 codes for linear classifier

(6) Stopping:
One reasonable early stopping choice is to stop training if no improvement is observed in validation data after a given number of times, which is mentioned in class. So we designed an early stopping condition that, when the validation loss does not decrease from the minimum loss for a certain number of times, we stop the training. Since initial few epochs should not be part of early stopping, we chose the number_to_stop to be 5 for 10-epochs-training. Then we added this early stopping condition into our validation part as seen in figure 8.



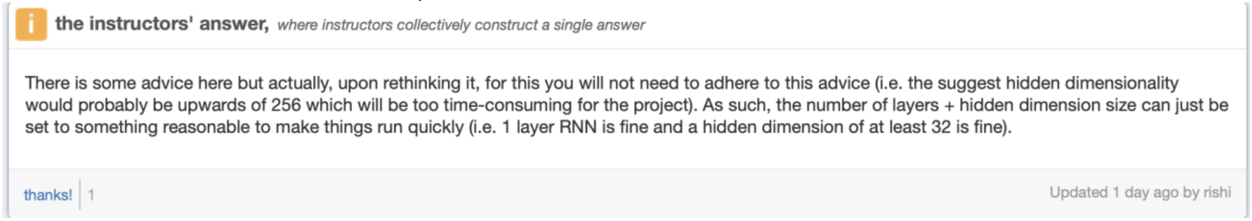
```
# early stopping condition
min_valid_loss = 1e10 # keep track of the minimum validation loss
number_to_stop = 5 # when reach this number, do the early stopping
counter = 0 # keep track of the number of epoches that do not decrease from minimum loss
stop_flag = False # early stopping flag
```



```
loss = None
correct = 0
total = 0
start_time = time.time()
print("Validation started for epoch {}".format(epoch + 1))
random.shuffle(vectorized_valid) # Good practice to shuffle order of validation data
minibatch_size = len(vectorized_valid)
N = len(vectorized_valid)
for minibatch_index in tqdm(range(N // minibatch_size)):
    loss = None
    for example_index in range(minibatch_size):
        input_vector, gold_label = vectorized_valid[minibatch_index * minibatch_size + example_index]
        predicted_vector = model(input_vector.unsqueeze(0))
        predicted_label = torch.argmax(predicted_vector)
        correct += int(predicted_label == gold_label)
        total += 1
        example_loss = model.compute_Loss(predicted_vector.view(1,-1), torch.tensor([gold_label]))
        if loss is None:
            loss = example_loss
        else:
            loss += example_loss
    loss = loss / minibatch_size

    # check for early stopping condition
    if loss < min_valid_loss:
        min_valid_loss = loss
        counter = 0
    else:
        counter += 1
    #print("Counter: {}".format(counter))
    if counter == number_to_stop:
        stop_flag = True
        break

print("Validation completed for epoch {}".format(epoch + 1))
print("Validation accuracy for epoch {}: {}".format(epoch + 1, correct / total))
print("Validation time for this epoch: {}".format(time.time() - start_time))

if stop_flag:
    print("Early stopping, with minimum validation loss {}".format(min_valid_loss))
    break
```

Figure 9 Early stopping method inside validation part

(7) Hyper-parameters:
For hidden dimensionality, we cannot find anything explicitly talking about it. So referring to the instructor's advice on Piazza, we chose hidden dimension to be 32 or 64.



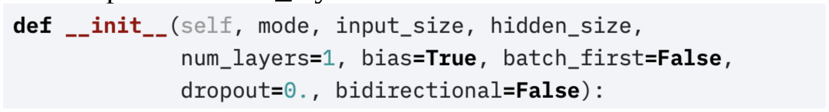
the instructors' answer, *where instructors collectively construct a single answer*

There is some advice here but actually, upon rethinking it, for this you will not need to adhere to this advice (i.e. the suggest hidden dimensionality would probably be upwards of 256 which will be too time-consuming for the project). As such, the number of layers + hidden dimension size can just be set to something reasonable to make things run quickly (i.e. 1 layer RNN is fine and a hidden dimension of at least 32 is fine).

thanks! | 1 Updated 1 day ago by rishi

Figure 10 why choose 32 or 64 for hidden dimension on Piazza

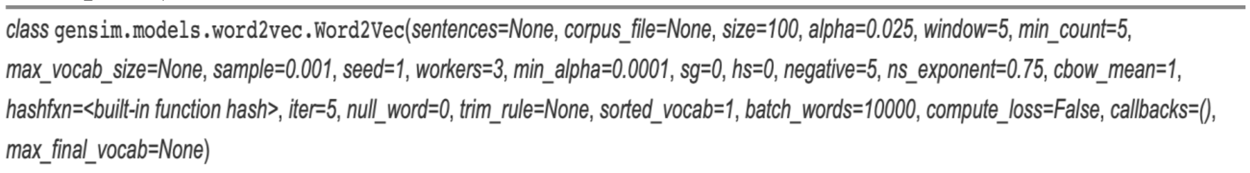
For number of layers, according to Pytorch official document[2] (see screenshot below), we set num_layers to its default value 1. Then in part 3, we set num_layers = 2, which is also a reasonable choice compared to num_layers=1.



```
def __init__(self, mode, input_size, hidden_size,
             num_layers=1, bias=True, batch_first=False,
             dropout=0., bidirectional=False):
```

Figure 11 how to set number of layers

For embedding dimension (input_dimension), according to Word2Vec documentation (see screenshot below), the default size is 100, so we set the size to be 128, which is very close to 100 and is also power of 2 (choosing power of 2 is common in model training to speed up the training time).



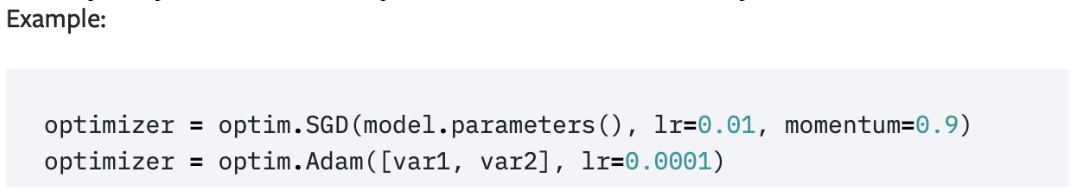
class gensim.models.word2vec.Word2Vec(*sentences=None*, *corpus_file=None*, *size=100*, *alpha=0.025*, *window=5*, *min_count=5*, *max_vocab_size=None*, *sample=0.001*, *seed=1*, *workers=3*, *min_alpha=0.0001*, *sg=0*, *hs=0*, *negative=5*, *ns_exponent=0.75*, *cbow_mean=1*, *hashfxn=<built-in function hash>*, *iter=5*, *null_word=0*, *trim_rule=None*, *sorted_vocab=1*, *batch_words=10000*, *compute_loss=False*, *callbacks=()*, *max_final_vocab=None*)

Figure 12 the settings of Word2Vec function

For the hyperparameters of pytorch.nn.RNN, “batch_first = True” sets the first dimension of input and output to be batch_size, which is convenient for us to do the implementation later (as we talked about in (4)).

The number_of_epochs is set to 10, the same value as in FFNN model.

For the optimizer, we set model=optim.SGD, learning rate=0.01, and momentum=0.9, which are all the same as in FFNN model. According to the documentation (screenshot below), torch.optim provides an example with the same choice of parameter values.

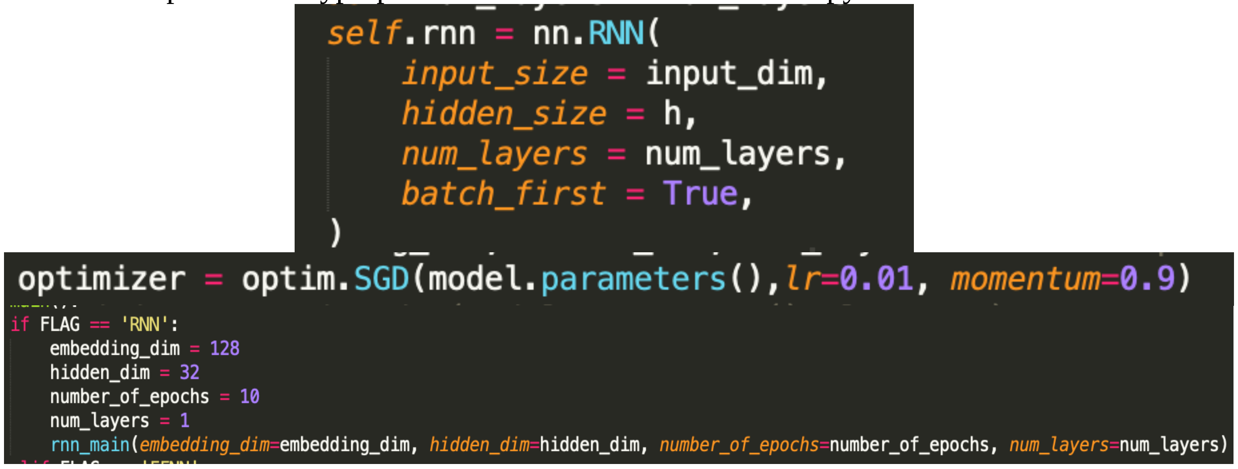


Example:

```
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
optimizer = optim.Adam([var1, var2], lr=0.0001)
```

Figure 13 the example for an optimizer

Below are screenshots for RNN hyperparameter declarations, optimizer model declaration, and how we pass these hyperparameters to RNN model in main.py.



```
self.rnn = nn.RNN(
    input_size = input_dim,
    hidden_size = h,
    num_layers = num_layers,
    batch_first = True,
)
```

```
optimizer = optim.SGD(model.parameters(),lr=0.01, momentum=0.9)
```

```
if FLAG == 'RNN':
    embedding_dim = 128
    hidden_dim = 32
    number_of_epochs = 10
    num_layers = 1
    rnn_main(embedding_dim=embedding_dim, hidden_dim=hidden_dim, number_of_epochs=number_of_epochs, num_layers=num_layers)
```

Figure 14 how to set the parameters of RNN

3. Across and within-model comparison

3.1 Across-model Comparison – Fair Comparison

The fair comparison compares three pairs of Feedforward model and RNN model. The configurations are different across pairs, but relatively similar within each pair. After that, we performed nuanced qualitative analysis on the results.

The three configurations we used are: (1) Set the hidden dimension for both models to 32. (2) Change the dimension of hidden layer for both models to 64. (3) Change the optimizer for both models from SGD (stochastic gradient descent) to Adagrad (adaptive gradient method). The results are shown in the table below.

Table 1 Fair Comparison of FFNN and RNN with different configurations

Model Configuration	FFNN accuracy	RNN accuracy
hidden dim = 32, optimizer = SGD	0.5275	0.29625
hidden dim = 64, optimizer = SGD	0.546875	0.320625
hidden dim = 64, optimizer = Adagrad	0.55625	0.378125

Both models in each pair have nearly identical configuration with 2 layers and a maximum of 10 epochs of running. For each configuration, we only changed one hyperparameter based on the previous one.

From the above result, FFNN model outperforms RNN model in all configurations. FFNN takes each word as input and process individually, while RNN takes in a sequence of words and process each sequence with the information of current and previous sequences. This should, theoretically, gives better results than FFNN. But the unexpected result we get may due to the small number of layers. Here is an example that FFNN did a good job but RNN gave an entirely wrong result (labeled as 1, predicted as 4)
"I order 2 Hazelnut Lattes. They were wrong. Was told to pull around the front for the correct d rinks. 10 minutes later and I'm still sitting here in the parking lot. What an embarrassment and a waste of $9." The reason might be that RNN gives more weights to words at the beginning of the sequence, so that the important words like "embarrassment" and "waste" are not taken into serious consideration, while FFNN captures these individual words correctly.

Across configurations, we can see that increasing the dimension of hidden layer increases the accuracy of both models by a fair amount. And switching the optimizer from SGD to Adagrad slightly improves the result of FFNN model, while greatly improves the RNN model accuracy.

Looking at the samples predicted incorrectly by the model, the number of samples that are predicted entirely wrong (labeled 1and predicted 4 or 5, or the opposite case) greatly reduced. Combining with the accuracy improvement acquired from the transition between configuration 1 and 2, we can conclude that a reasonably larger hidden dimensionality could effectively increase model accuracy, and the ideal size of hidden layer is somewhere between the input layer size and output layer size [3].

The transition from configuration 2 to 3 is to study how different optimizers affect model performance. The SGD uses the same learning rate at each iteration, while Adagrad uses adaptive gradient that adjusts the learning rate according to the squared gradient at each iteration. So Adagrad eliminates the need for tuning the best learning rate in SGD, and generally yields better results across models.

3.2 Within-model Comparison – Ablation Study

The ablation study performs a nuanced quantitative analysis of the RNN model.
The two features that we are concerned about are the dimension of the hidden layer and the number of layers.

We first listed the results of four models in the table 1. We trained every model for 10 epochs but some of them had an early stopping which had been mentioned in Section 2. The results in the table are just in the sequence of model 4, 3, 2, 1.

Table 2 Experimental Results for Four RNN models

The settings of model	Best Validation Accuracy	Early-Stopping
4. Hidden dim = 64, number of layers = 2	**0.326875**	No
3. Hidden dim = 32, number of layers = 2	0.2675	Yes, Epoch 9
2. Hidden dim = 64, number of layers = 1	0.313125	No
1. Hidden dim = 32, number of layers = 1	0.298125	Yes, Epoch 6

From model 4 to model 3, we only change the hidden dimension of each RNN layer from 64 to 32. However, the model accuracy dropped greatly (about 5% of accuracy) and had an early stopping since the model cannot learn information efficiently. So, we can reasonably conclude that when hidden dimension is too small, it will seriously restrict the model's learning ability.

From model 4 to model 2, we turned number of layers from 2 to 1. We can find it from the results that the accuracy does decrease, but not so much (about 1.3% of accuracy). It implies that on the given setting, using more layers can improve the efficiency of the model.

If we modify the hidden dimension and number of layers at the same time, as from model 4 to model 1, we can also find the results decrease by about 3% of accuracy. So, decreasing the hidden dimension and number of layers when they are not great will certainly have a negative effect on the model's efficiency.

The improvement that resulted from increase of hidden dimension and number of layers are true when they are within some certain range. Since dimensions like 32, or even 16, 8,4 are too small, improving them will be good for model. However, if we increase them endlessly, to say, like making them over 10,000, then the result will be not so good.

This point is same with number of layers, using 2 layers instead of 1 may be a good idea, but if we turned them into 5 or over 10, the result will not be so good, despite the huge computation.

And we can also find in the results that the number of layers and the hidden dimension are related in the overall performance. When the hidden dimension is small, increasing the number of layers may not always improve the results, just like changing from model 1 to model 3. We think it may result from the total representation ability of the model is not enough, the results can be improved by increasing number of layers only when the hidden dimension is enough.

4. Answers for the questions in Part 4:

In this section, we will answer the four questions sequentially.

1. What is a fundamental reason for RNNs to struggle with long-distance dependencies?
   Answer: The RNN's hidden layers are subject to repeated multiplications, so the gradients are eventually driven to zero when the length of layer increases. Thus, it cannot preserve long-distance information very well.

2. Why feeding a sentence backwards may improve the performance?
   Answer: Since in an RNN model, when a word that is put into the model after other words, the computation of this word in the model are more than other words, which means that word will be considered more by the model. And for a sentence, in many cases, the words that appear late will be more important than the beginning ones. For example, 'great' is more important than 'NLP' in a sentiment analysis task.

3. Beyond concerns of dataset size, why might we disfavor RNN models?
   Answer: We first can think of the long-distance dependencies. Besides that, using RNN instead of feature engineering may make us less aware of what is going on. To be more specific, we cannot know exactly what features are learned by the model, so it is harder to optimize a certain RNN model by necessary semantic knowledge.

5. Miscellaneous

5.1 Libraries

The libraries we mainly use for RNN models are numpy, torch and genism. By numpy, we did some basic numeric operations; torch can provide necessary machine learning frameworks and genism can give Word2Vec embeddings.

5.2 Reference

[1] Mikolov, Tomas, et al. "Efficient estimation of word representations in vector space." arXiv preprint arXiv:1301.3781 (2013).
[2] Paszke, Adam and Gross, Sam and Chintala, Soumith and Chanan, Gregory and Yang, Edward and DeVito, Zachary and Lin, Zeming and Desmaison, Alban and Antiga, Luca and Lerer, Adam, Automatic Differentiation in PyTorch, NIPS Autodiff Workshop, 2017.
[3] Blum, Adam, Neural Networks in C++, 1992.

5.3 Workflow & Contribution of each group member:

At first, we discussed the whole structure of the work and found FFNN and RNN to be relatively independent, so the coding work can be divided (though the implementations of Pytorch are similar). Then we kept on discussing the necessary follow-up tests that we need for this project and communicating about whether to do some tests or not. We shared the codes in a private Github repository, so we can keep up with each other's work and post the problems encountered.

Boliang Yang (by283): RNN implementation and related experiments, report revision.
Jinwei Shen(js3559): Comparison of FFNN and RNN models, test of codes.
Shenghua Li(sl3293): FFNN debugging, RNN code testing and structure of the paper.

5.4 Feedback:

The overall difficulty of the project is medium, we can handle it and learn necessary parts. And the instructions are detailed, which helped us much. But some comparison and problems are somehow vague, thus made us a bit confused (But most of them are solved in the Piazza posts).