Felipe Pazos (fap26)
Michael D'Alessandro (msd248)
Oscar So (ons4)
Project 3
11/11/2019

Neural Networks

Part 1: Feedforward Neural Network

Here are four major errors we found:

1) The neural network output dimension was incorrect. The original code has the second layer of the network being a linear combination from *h* to *h.* However, the classification task only has 5 labels. As a result, it should be the case that the final layer is linear combination from *h* to *5*. This is a simple one line fix:

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

2) In the forward pass, the network does not apply the activation function between layers. This matters because missing the activation function makes having two layers redundant. Applying the forward pass without activation becomes equivalent to two matrix multiplications, which is simply equivalent to a single matrix multiply. Not having the activation essentially removes the hidden layer entirely from our network. To fix this, simply add in the activation function:

```
z1 = self.W1(input_vector) //original
z1 = self.activation( self.W1(input_vector)) //fix
```

3) The neural network code was kept in training mode the entire time. This is problematic because it means that the network continued to train on the validation set. This is not the desired behavior - we are using the validation set to measure the validity of our trained network. Training the network on the validation examples ruins the entire point of validation. To fix it this bug, simply add the following line after training, but before validation:

```
model.eval() //fix
```

   Also, removing loss.backward() and optimizer.step() from the bottom of the code would further prevent the model from learning in the evaluation stage and prevent redundancy of code.

4) The network wasn't properly training. In the original file, *ffnn.py*, the backwards pass and optimizer step were only occurring after the entire training set had been processed. This is fixed in *ffnn1fix.py*, by having the backwards pass and optimizer step occur after every minibatch. The fix here was indenting the following two lines into the training code block:

```
loss.backward()
optimizer.step()
```

There were other minor errors, apart from error 4, that was also fixed in *ffnn1fix.py*. For example, in make_indices(), 'unk' was added to the vocab when the global variable was defined to be '<unk>' in variable unk instead. Thus, it should be vocab.add(unk) instead of vocab.add('unk'). This is because it would add one extra unknown word to the vocab, giving unk a "weight"/count of 2 instead of 1. Secondly, in the first edition of *ffnn1fix.py*, the code for validation still used the training dataset, thus, we just changed all "train_data" below our added model.eval() into "valid_data" instead. The final results of FFNN is pictured below in *Image 1*.

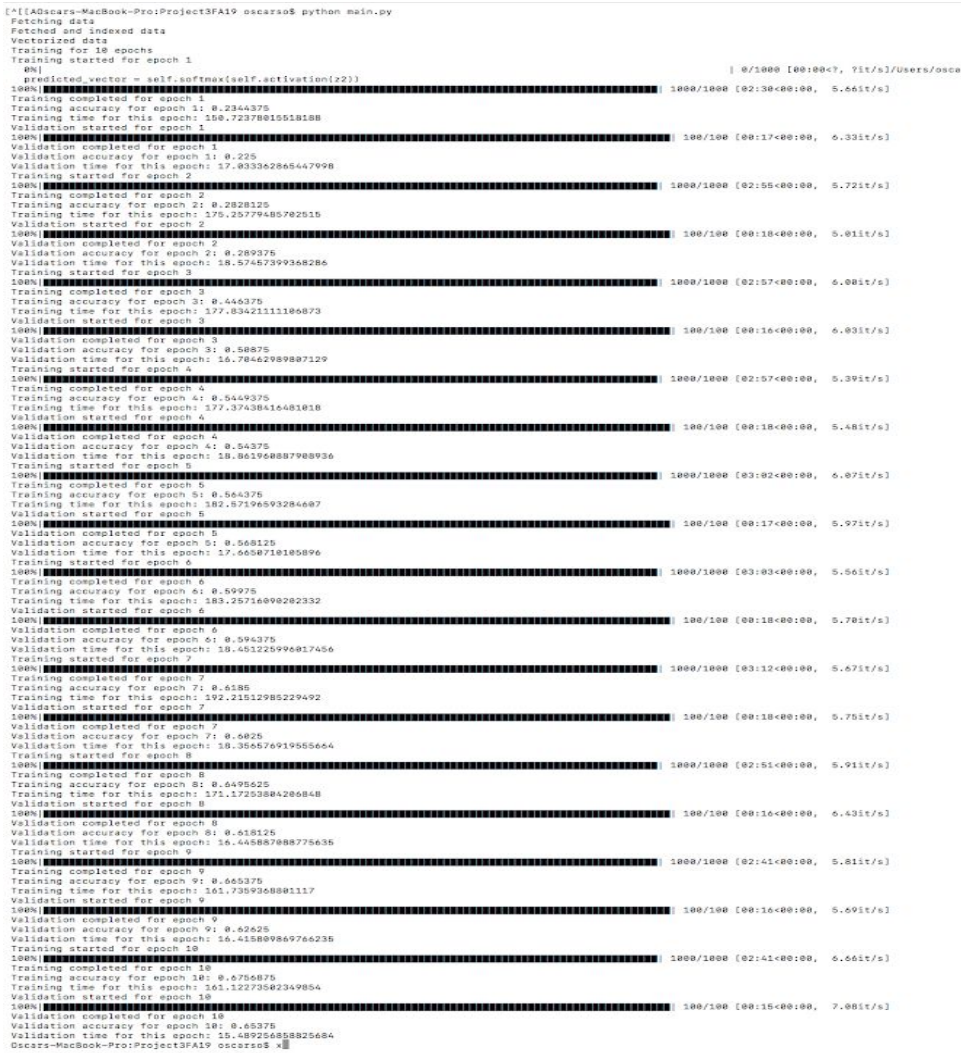

Image 1: Screenshot of FFNN After Bug Fixes

Part 2: Recurrent Neural Network

I. Representation:

Instead of the bag of words vector representation that was given to us in *ffnn1fix.py*, we decided to represent each x_i as a word embedding from spaCy's "en_core_web_lg" dataset.[4] This means that each word will be a 1x300 dimension, and each document with n words will be represented as a list of n (1x300 dimension) embeddings. We call the *convert_to_vector_representation()* function on the training data and the validation data to translate it into the vectors, which we then pickle/cache to prevent having to do it again.

Code Snippet:

```
def convert_to_vector_representation(data):
    spacy.prefer_gpu()
    nlp = spacy.load("en_core_web_lg")
    vectors = []
    for document, star in tqdm(data):
        #document = nlp(document)
        document_vectors = []
        for i in document:
            i = nlp(i)
            document_vectors.append(i.vector)
        vectors.append((torch.tensor(document_vectors), star))
    return vectors
```

```
if CACHED:
    train_data = pickle.load(open("train_data.pkl", "rb"))
    valid_data = pickle.load(open("valid_data.pkl", "rb"))
else:
    train_data = convert_to_vector_representation(train_data)
    valid_data = convert_to_vector_representation(valid_data)
    pickle.dump(train_data, open("train_data.pkl", "wb"))
    pickle.dump(valid_data, open("valid_data.pkl", "wb"))
```

II. Initialization:

Since we are using pytorch.nn's built-in RNN model[5], it immediately initializes a hidden weight vector without us having to code it. Initially, we had an *init_hidden()* function that created a tensor of 0s in 3-dimensions, but leveraging information from OH, we were advised to remove that function given PyTorch's RNN's functionalities. This is also because that the hidden dimension will be passed through the model regardless, and if the hidden dimension doesn't initially exist, the model will create it itself. The initialized weights from the model itself will be drawn from a uniform distribution between $(-\sqrt{k}, \sqrt{k})$ where $k = 1/h$.[5]

Code Snippet:

```
self.rnn = nn.RNN(input_dim, h, n_layers,
                  nonlinearity='relu')
```

```
def forward(self, inputs):
    out, hidden = self.rnn(inputs)
    out = self.fc(out[-1].view(1, -1))
    output = self.softmax(out)
    return output, hidden
```

III. Training:

We will be using 10 epochs for training, and every epoch will be further broken down into mini-batches of 16 data points. After every minibatch, the loss will be averaged and we will then use loss.backward() and optimizer.step() to update our model to learn. This applies gradient descent to our RNN. This method first computes the gradient, and steps the weights in the direction of the negative gradient. This works because the gradient points in the direction that most increases the loss; by stepping in the negative direction, we effectively take the step that best decreases the loss.

Code Snippet:

```
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]
        input_vector = input_vector.to(device)
        predicted_vector, hidden = model(input_vector.unsqueeze(1))
        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
    if (loss.item() > 100):
        print ("Stopping: Model might be Overfitting")
        os._exit(1)
    loss.backward()
    print(loss.item())
    optimizer.step()
```

IV. Model:

Zooming in on the previous code snippet, we have the core model of the RNN. This is where we feed the model our input vector of train_data, which is a list of vectors of each word in a document as a word embedding, and allow the model to learn from it. We had to use *input_vector.unsqueeze(1)* because our batch_size for the RNN model itself was just 1. Therefore, we had to add a dummy dimension such that if our initial input_vector was a k x d matrix, we would not have a k x 1 x d matrix.[3]

Code Snippet:

```
for example_index in range(minibatch_size):
    input_vector, gold_label = train_data[minibatch_index *
                                          minibatch_size + example_index]
    input_vector = input_vector.to(device)
    predicted_vector, hidden = model(input_vector.unsqueeze(1))
```

V. Linear Classifier:

Once the RNN is finished with the forward pass, we still need to pass it through an additional linear layer so that the output can have a dimension of 5 x 1. This is because the intermediate outputs have a dimension of h x 1, where h is the size of the hidden dimension. Doing this preserves expressibility in the RNN, but is incompatible with our required output. To fix this, we pass the final output (given by *out[-1]*) through a final linear layer with dimension h x 5. Then, we softmax the output to normalize the output vector.[10]

Code Snippet:

```
def forward(self, inputs):
    out, hidden = self.rnn(inputs)
    out = self.fc(out[-1].view(1, -1))
    output = self.softmax(out)
    return output, hidden
```

VI. Stopping:

In the stopping case, we realized that when the training loss was increasing, the model seemed to start overfitting. The loss values were often around 1-2, but overfitting occurs when the loss significantly increases to a larger number, in our case, set to 100. We have tried using models with a high learning rate as a trial and error and gotten 18 digit losses before the model turned unstable and loss became nan. Thus, we have decided to set a cap that when a training loss rises above our cap of 100, we will stop training the data and exit, giving the user a print statement that informs the user that the model could be overfitting.

Code Snippet:

```
    if loss is None:
        loss = example_loss
    else:
        loss += example_loss
loss = loss / minibatch_size
if (loss.item() > 100):
    print ("Stopping: Model might be Overfitting")
    os._exit(1)
loss.backward()
print(loss.item())
```

VII. Hyperparameters:

The hyperparameters we set in the RNN included the hidden dimensions, number of epochs, the number of layers, and the learning rate. We set the hidden dimensions to be 32 based on information from Piazza[1] as well as our own cost-benefit analysis. Increasing the hidden dimensions would greatly increase the computational and time costs of running the RNN, which we did not feel was justified by increases in accuracy.

Additionally, we set the number of epochs to 10 because we noticed that our losses were not changing in the last few epochs, and we did not think increasing the number of epochs would have any further impact on the loss.[2] Thus, we did not think it was worth the extra computing power and time to increase the number of epochs.

Another hyperparameter choice was the number of layers of the RNN. We ended up setting this value to one after extensive resource. We came to this decision having leveraged Piazza[1] as well as the official PyTorch docs which set the number of recurrent layers to one by default.[7]

The learning rate was the hyperparameter where we focused most of our attention. When the learning rate was set to 0.010, the model was learning too quickly and things became unstable as our loss became “NaN” after a few epochs.[7, 8] Our solution to this was to decrease our learning rate to 0.005. While our model did not learn as quickly, it was a more stable approach.

We do test how the accuracy differs when changing some of these hyperparameters in part 3.

Code Snippet:

```
def main():
    if FLAG == 'RNN':
        hidden_dim = 32
        number_of_epochs = 10
        rnn_main(hidden_dim=hidden_dim, number_of_epochs=number_of_epochs)
```

```
def __init__(self, input_dim, h, output_size, n_layers, device):
```

```
model = RNN(input_dim=300, h=hidden_dim,
            output_size=5, n_layers=1, device=device)
```

```
optimizer = optim.SGD(model.parameters(), lr=0.005, momentum=0.9)
```

Part 3a: Across-Model Comparison

To take a deeper dive at the data, we had the models output to a .csv file information about the predicted labels and gold standard label for each sequence in the validation set. Thus, we can compare results of the models on a more granular scale and look at how they performed on each sequence.

Question 1: What if the learning rate is 0.005 for both models?

Argument: Altering the learning rate to be the same for both models is a fair comparison because the learning rate controls the rate at which the model learns. Setting them to be the same should give results that are comparable.

Results:

FFNN with lr=0.005

Epoch 1:	Validation Accuracy: 0.555
Epoch 2:	Validation Accuracy: 0.523125
Epoch 3:	Validation Accuracy: 0.341875
Epoch 4:	Validation Accuracy: 0.49625
Epoch 5:	Validation Accuracy: 0.350625
Epoch 6:	Validation Accuracy: 0.470625
Epoch 7:	Validation Accuracy: 0.444375
Epoch 8:	Validation Accuracy: 0.435
Epoch 9:	Validation Accuracy: 0.385625
Epoch 10:	Validation Accuracy: 0.38625

RNN with lr=0.005

Epoch 1:	Validation Accuracy: 0.279375
Epoch 2:	Validation Accuracy: 0.294375
Epoch 3:	Validation Accuracy: 0.3325
Epoch 4:	Validation Accuracy: 0.366875
Epoch 5:	Validation Accuracy: 0.339375

Epoch 6:	Validation Accuracy: 0.330625
Epoch 7:	Validation Accuracy: 0.31625
Epoch 8:	Validation Accuracy: 0.296875
Epoch 9:	Validation Accuracy: 0.353125
Epoch 10:	Validation Accuracy: 0.35875

Question 2: What if hidden dimensions are doubled, 64, for both models?

Argument: Setting the learning rate to 0.005 for both models and having 10 epochs will create a fair comparison for us to study how the two models differ when the hidden dimensions are changed to 64 for both models

Results:

FFNN with hidden_dim = 64

Epoch 1:	Validation Accuracy: 0.678125
Epoch 2:	Validation Accuracy: 0.76875
Epoch 3:	Validation Accuracy: 0.64
Epoch 4:	Validation Accuracy: 0.669375
Epoch 5:	Validation Accuracy: 0.748125
Epoch 6:	Validation Accuracy: 0.543125
Epoch 7:	Validation Accuracy: 0.7425
Epoch 8:	Validation Accuracy: 0.5975
Epoch 9:	Validation Accuracy: 0.663125
Epoch 10:	Validation Accuracy: 0.763125

RNN with hidden_dim = 64

Epoch 1:	Validation Accuracy: 0.30625
Epoch 2:	Validation Accuracy: 0.31125
Epoch 3:	Validation Accuracy: 0.355625
Epoch 4:	Validation Accuracy: 0.29625

Epoch 5:	Validation Accuracy: 0.30875
Epoch 6:	Validation Accuracy: 0.34875
Epoch 7:	Validation Accuracy: 0.335625
Epoch 8:	Validation Accuracy: 0.365
Epoch 9:	Validation Accuracy: (Training Stopped: Overfitting)
Epoch 10:	Validation Accuracy: (Training Stopped: Overfitting)

Question 3: What if epoch was 5 instead of 10 for both models?

Argument: Setting epochs to 5 for both models with all other parameters set to standard will create a fair comparison for us given the fact that both models will have the same amount of data, and epochs, to train itself with. This way, not one model will have more training than the other and provide higher results.

Results:

FFNN with 5 Epochs:

Epoch 1:	Validation Accuracy: 0.643125
Epoch 2:	Validation Accuracy: 0.71625
Epoch 3:	Validation Accuracy: 0.71625
Epoch 4:	Validation Accuracy: 0.670625
Epoch 5:	Validation Accuracy: 0.6025

RNN with 5 Epochs:

Epoch 1:	Validation Accuracy: 0.33625
Epoch 2:	Validation Accuracy: 0.3225
Epoch 3:	Validation Accuracy: 0.319375
Epoch 4:	Validation Accuracy: 0.308125
Epoch 5:	Validation Accuracy: 0.349375

Overall Analysis: *Nuanced Qualitative Analysis* (Utilizing stars: [0,4])

For our nuanced qualitative analysis, we plan on taking a look at a few sequences and how the models handled the task based on their last calculated epochs. We will look at the following sentences below to get a grasp onto some qualitative analysis of our models:

Sentence 1190:
`This is what my family gets for our bagel brunch tradition after we fly in to visit relatives.\n\nThis Los Angeles native and Jewish kid LOVES what a bagel!`

Gold Label: 3 stars

Q1) Learning Rate of 0.005:
FFNN: 3 stars
RNN: 4 stars

Q2) Hidden Dimensions of 64:
FFNN: 4 stars
RNN: 3 stars

Q3) 5 Epochs:
FFNN: 1 stars
RNN: 3 stars

The only neural network model that was incorrect by a large margin was our FFNN for the third model, which only used 5 epochs. While the RNN got this sentence right, the FFNN did not. We believe that this is because 5 epochs is not enough time for the FFNN to train and the weights were not fully updated. Besides this model, we are impressed with the other models because they got the rating right within a one star deviance. This sentence, if we observe it, would not even be labelled by most humans as a 3 star review. It is quite impressive that the models were able to label it in such a precise manner.

Sentence 9:
`I wish I could give less than 1 star for this place. First, we pulled in and there were only two other Cars in the parking lot. The patio was full of rusty overturned chairs so we couldn't eat outside. My girlfriend and I ordered 2 waters which had a metallic taste to it. Then I ordered a green tea martini, which they said that they were unable to make due to the fact that \"nobody liked it\" so they got rid of the green tea liquor. We then ordered 3 sushi rolls for half off sushi night. Once they came out, we each tried a piece of the shrimp tempura roll. The shrimp tasted like it was fried last week`

due to the fact that it was hard as a rock. We then tried the eel which was entirely skin and no meat. We also both noticed that nothing was very cold. It was all at room temperature, the rice tasted old, and the seafood had a bad aroma to it. After 20 minutes our server finally came over and we told her that everything was inedible. She said \"Okay\" and took our plates away. After waiting 10 more minutes, we got 2 fortune cookies. The manager never stopped over and apologized, and neither did the server. I will never be returning.

Gold Label: 0 stars

Q1) Learning Rate of 0.005:
FFNN: 0 stars
RNN: 0 stars

Q2) Hidden Dimensions of 64:
FFNN: 0 stars
RNN: 0 stars

Q3) 5 Epochs:
FFNN: 0 stars
RNN: 0 stars

It makes sense that this review was correctly classified by every single neural network. This makes sense, as this review is clearly negative in just about every way. An FFNN would have an easy time classifying this, as there are enough words with negative connotation to make a bag of words representation very obviously correspond to a negative rating. Words like “less”, “rusty”, “metallic”, “old”, “bad”, “inedible”, “and “apologized” are all words that would be very likely to be found in a negative review. Similarly, the word embeddings used by the RNN should be able to pick up on the negative connotation of these words.

Sentence 561:
first pizza i requested well done, but got burnt and black. the nice owner gladly replaced it free, but i've had several disappointing meals there. how it get's such a stellar rating is beyond me.\n\ngreat people, great idea, average food at best.

Gold Label: 1 star

Q1) Learning Rate of 0.005:
FFNN: 3 stars
RNN: 3 stars

Q2) Hidden Dimensions of 64:
FFNN: 3 stars
RNN: 2 stars

Q3) 5 Epochs:
FFNN: 2 stars
RNN: 0 stars

For this sentence, none of the models' last epochs got the labelling correct. All the models, except for the 5 epoch RNN, gave the review a much higher score than the gold label did. There are many words that could've impacted the model's decision in labelling. For instance, words like "well done", "free", "great", "best", could all result in a higher rating learnt by the bag of words representation for the FFNN, and the vector representations in the RNN.

Part 3b: Within-Model Comparison

Model 4: Combination of Model 2 and Model 3

Description:

```
model = RNN(input_dim=300, h=hidden_dim*2,
            output_size=5, n_layers=2, device=device)
```

Results:

Epoch 1:	Validation Accuracy: 0.31625
Epoch 2:	Validation Accuracy: 0.354375
Epoch 3:	Validation Accuracy: 0.34375
Epoch 4:	Validation Accuracy: 0.28375
Epoch 5:	Validation Accuracy: 0.311875
Epoch 6:	Validation Accuracy: 0.32375
Epoch 7:	Validation Accuracy: 0.369375
Epoch 8:	Validation Accuracy: 0.363125
Epoch 9:	Validation Accuracy: (Training Stopped: Overfitting)
Epoch 10:	Validation Accuracy: (Training Stopped: Overfitting)

Model 3: Baseline Model with num_layers = 2

Description:

```
model = RNN(input_dim=300, h=hidden_dim,
            output_size=5, n_layers=2, device=device)
```

Results:

Epoch 1:	Validation Accuracy: 0.33125
Epoch 2:	Validation Accuracy: 0.3
Epoch 3:	Validation Accuracy: 0.344375
Epoch 4:	Validation Accuracy: 0.34
Epoch 5:	Validation Accuracy: 0.261875
Epoch 6:	Validation Accuracy: 0.27875
Epoch 7:	Validation Accuracy: 0.3375
Epoch 8:	Validation Accuracy: (Training Stopped: Overfitting)
Epoch 9:	Validation Accuracy: (Training Stopped: Overfitting)
Epoch 10:	Validation Accuracy: (Training Stopped: Overfitting)

Model 2: Baseline Model with Hidden Dimensions Doubled

Description:

```
model = RNN(input_dim=300, h=hidden_dim*2,
            output_size=5, n_layers=1, device=device)
```

Results:

Epoch 1:	Validation Accuracy: 0.33375
Epoch 2:	Validation Accuracy: 0.36
Epoch 3:	Validation Accuracy: 0.255625
Epoch 4:	Validation Accuracy: 0.335625
Epoch 5:	Validation Accuracy: 0.3675
Epoch 6:	Validation Accuracy: 0.32625
Epoch 7:	Validation Accuracy: 0.40625
Epoch 8:	Validation Accuracy: 0.306875

Epoch 9:	Validation Accuracy: 0.32375
Epoch 10:	Validation Accuracy: 0.34625

Model 1: Baseline Model

Description:

```
model = RNN(input_dim=300, h=hidden_dim,
            output_size=5, n_layers=1, device=device)
```

Results:

Epoch 1:	Validation Accuracy: 0.3275
Epoch 2:	Validation Accuracy: 0.2975
Epoch 3:	Validation Accuracy: 0.355625
Epoch 4:	Validation Accuracy: 0.285625
Epoch 5:	Validation Accuracy: 0.30625
Epoch 6:	Validation Accuracy: 0.338125
Epoch 7:	Validation Accuracy: 0.31625
Epoch 8:	Validation Accuracy: 0.32
Epoch 9:	Validation Accuracy: 0.323125
Epoch 10:	Validation Accuracy: 0.26375

Notes:

- Baseline Model (Model 1) is fast
- Model 2 is twice as slow as Model 1
- Model 3 is thrice as slow as Model 1
- Model 4 is twice as slow as Model 3.

Overall Analysis: *Nuanced Quantitative Analysis*

For our 4 models, we chose to use a combination of changing the layers of the RNN and also the hidden dimensions of the RNN. For model 4, we doubled the hidden dimensions from 32 to 64 and also doubled the RNN layers from 1 to 2. For model 3, we doubled RNN layers from 1 to 2 only. For model 2, we doubled the hidden dimensions from 32 to 64 only. For model 1, this is the baseline model, we kept everything as is with hidden dimensions as 32 and RNN model layer as 1. Above, we have shown the validation accuracy for 10 epochs during our run of the model's training and validation, and this quantitative accuracy[11] demonstrates how well the model has been trained to match ratings with the validation set. Looking at model 4, we can see

that the model starts to overfit halfway through epoch 9, and provides a final validation accuracy of ~36%. This overfitting could be because of the number of layers doubling, which would lead to the RNN's output being passed through itself one more time and having more precise weights to update instead. Looking at model 4's comparison with model 3, we see that model 3 started overfitting even earlier in the training stage, near the beginning of epoch 8. This could mean that the hidden dimensions doubling could've actually helped the model by not overfitting too early in the training stage. However, with a decrease in the hidden dimension, the validation accuracy also decreases to 33.75% instead, which is lower than 36% from model 4 itself. Looking at model 2, it seems that reducing the number of layers in the RNN definitely helped as all the epochs were run through consistently, with the validation accuracy going as high as 40.6% in epoch 7 and ending with ~35% in epoch 10, which is rather similar to model 4's accuracy. This could go to demonstrate that increasing the number of hidden dimensions would definitely be beneficial in increasing the accuracy of the model, and with a greater hidden dimension, there is a greater dimension for weights to be trained, which could increase the expressibility of the network. Lastly, the baseline model provides a final accuracy of 26.4% with averages of around 31%. Once again, fulfilling our analysis that an increase of hidden dimension would benefit the model and an increase in the RNN's layer allows the model to be a bit more expressive, but not as important as the hidden dimension itself.

Part 4: Questions

1. **Earlier in the course, we studied models that make use of *Markov* assumptions. Recurrent neural networks do not make any such assumption. That said, RNNs are known to struggle with long-distance dependencies. What is a fundamental reason for why this is the case?**

   In an RNN, each level of computation only receives input from previous dependencies based on the hidden layer which is a fixed size. As the RNN progresses and processes more inputs, inputs from further in the past decay in importance because the hidden layer cannot accommodate every single example simultaneously. The hidden layer cannot hold infinite information, so as the network learns more about more recent examples, it must be the case that some information about very long-distance dependencies is lost. As a result, RNNs will perform worse the further the distance in the dependencies are.

2. **In applying RNNs to tasks in NLP, we have discovered that (at least for tasks in English) feeding a sentence into an RNN backwards (i.e. inputting the sequence of vectors corresponding to (*course*, *great*, *a*, *is*, *NLP*) instead of (*NLP*, *is*, *a*, *great*, *course*)) tends to improve performance. Why might this be the case?**

   An RNN works because it works to learn generalizations about the input sequence while considering the context of what was processed before. This means that the generalizations and patterns that make the RNN work will only be dependent on what the RNN has seen before it - not what it has yet to see. As a result, based on the syntax of the English language, it may be the case that context from the end of a sentence is more valuable to a general understanding than the context given by the beginning of a sentence. One example of this is the fact that English tends to usually have a subject-verb ordering as opposed to verb-subject. It may be more valuable for a model to train by processing the verb first as opposed to the other way around. By providing the RNN with a reversed input, it will process the last words first, letting these words be used when analyzing earlier words.

3. **In using RNNs and word embeddings for NLP tasks, we are no longer required to engineer specific features that are useful for the task; the model discovers them automatically. Stated differently, it seems that neural models tend to discover better features than human researchers can directly specify. This comes at the cost of systems having to consume tremendous amounts of data to learn these kinds of patterns from the data. Beyond concerns of dataset size (and the computational resources required to process and train using this data as well as the further environmental harm that results from this process), why might we disfavor RNN models?**

A hidden cost of using RNNs is that the human data engineer creating them doesn't understand how the underlying feature detection works. As a result, it can be incredibly difficult to debug a broken network. When a model isn't getting a sufficiently high validation accuracy, it can be difficult to understand why. Further, in the cases that the language model is being used to generate understanding (as in the paper regarding deceptive opinion spam), an RNN often isn't the best choice as neural networks are functionally black boxes that don't provide much insight into why a certain classification was made.

Part 5a: Miscellaneous

For all work regarding the neural networks, we used the provided PyTorch library. For our RNN that used vectorized word embeddings, we used the spaCy library. We also used the pickle library for cache-ing our vectorized data of word-embeddings and the os library for exiting the code when we feel like the model is overfitting. We also used PyTorch's online example, "NLP From Scratch: Classifying names with a character-level RNN"[6] to understand more on how RNNs worked, but their implementation is vastly different than ours given they used character-level vectors instead of ours, using word-level vectors.

Our group split up the work evenly. Writing the assignment was split by the three of us and reviewed and discussed. Each of us agreed to take lead on a different part of the assignment which involved understanding the background of what each part involved. However, we all coded with one another in order to ensure we all could explain what each part of the code involved.

Cumulatively, we spent about 20 hours of in-person coding sessions where all three of us were involved. This does not take into account the hours we spent debugging on our own and training the datasets or the time spent learning the material for this project.

In terms of time spent on each section, each of us spent about an hour on the first part of the assignment, with about 2 hours of training time each. For the second part, we each spent 5 hours working on the RNN and about 15 hours each training the models and fixing bugs. For the third part we spent about an hour each brainstorming the models and 6 hours each training models. The fourth and fifth parts involved about an hour of work combined together.

Training took approximately 4 hours each the first few times because we did not pickle our data. Initially, it took us about 3 hours to vectorize all data into word embeddings, and another 30-45 minutes total to run the training and validation for the RNN. After the pickling, the time went down significantly to less than 1 hour per training.

Overall we found this assignment to be challenging and rewarding. While no one in our group had used PyTorch before, we were able to get a working knowledge of it fairly quickly. We found that building the RNN was incredibly helpful at rounding out our knowledge of the subject. However, we thought more of a framework for PyTorch would have been helpful as many of the errors we ran into were as a result of our relative inexperience when working with PyTorch. Furthermore, we would've loved to either get less work on this particular project, or more time to work on it. This is because this project was really time-intensive and we spent a lot of our time working on the project, particularly during training.

Part 5b: Sources

[1] https://piazza.com/class/jzwr5pdhtlf134?cid=581
[2] https://discuss.pytorch.org/t/resolved-how-many-epochs-should-one-train-for/1885
[3] https://piazza.com/class/jzwr5pdhtlf134?cid=567
[4] https://spacy.io/models/en
[5] https://pytorch.org/docs/stable/nn.html
[6] https://pytorch.org/tutorials/intermediate/char_rnn_classification_tutorial.html
[7] https://pytorch.org/docs/master/nn.html#torch.nn.LSTM
[8] https://piazza.com/class/jzwr5pdhtlf134?cid=560
[9] https://machinelearningmastery.com/learning-rate-for-deep-learning-neural-networks/
[10] https://piazza.com/class/jzwr5pdhtlf134?cid=613
[11] https://piazza.com/class/jzwr5pdhtlf134?cid=590