CS4395 Assignment 2

https://github.com/RidaZameer07/cs4395_assignment2

Rida Zameer

RXZ200007

1 Introduction and Data (5pt)

Project Description:

In this project, I implemented and tested two NN (Neural Network) Models: an FFNN (Feed-Forward Neural Network) and a RNN (Recurrent Neural Network). The goal was to train these models using the provided data sourced from Yelp reviews (JSON format) and test the performance of a sentient analysis to accurately predict an outcome (classifying from 1-5 stars based on data analysis). The models were trained with multiple hyperparameters, such as hidden dimension size and epoch count (number of repeated runs per tested hidden dimension size). My results will focus on going over the effects of changing these parameters, particularly when increasing the hidden dimension size.

FFNN: I tested the FFNN with an epoch size of 5 and values of 20 and 50 for the hidden dimension size (HDS). In my experiments, I compare the average accuracy values of the three tests I conducted (baseline/default, HDS = 20, HDS = 50), specifically comparing the training and validation averages for each test. The results indicated that increasing the hidden dimension size from 20 to 50 did increase accuracy, but only by a small portion (0.015). Compared to the baseline, both tests do perform better by about .09. However, increasing the size of the HDS from 20 to 50 significantly increases the run time by 4 to 5 times.

RNN: I tested the RNN with an epoch size of 10 and a hidden dimension size of 64. I compared this test with the baseline/default to better understand the effect the growing dimension sizes had on the accuracy results, as well as the time and epoch limitation statement given while running the program. I give specific measurements for each epoch successfully run and calculate average and best scores for accuracy. Compared to the baseline (HDS = 32, epoch = 10), the test I conducted (HDS = 64, epoch = 10), the test with increased HDS did help boost accuracy measurements faster in the earlier epochs, but it also seemed to overfit faster as it stopped at just 2 epochs compared to 8. To summarize, comparing the results of the baseline and test showed that increasing the hidden dimension size gave roughly the same but higher accuracy and took about the same amount of time. The noteworthy difference would be noted in how many epochs were able to run before terminating for overfitting (8 versus 2 for baseline versus test).

Task and Data:

The task is to classify the data, which consists of extracted Yelp restaurant reviews, into the predicted rating (1 to 5 stars). The dataset is split into the training set and validation set (the test set is not used). The training set for FFNN includes 500 examples, while the validation set includes 50. For RNN, the training set is size 1000 and validation is 800. All data consists of a sentence and its true value/rating, all formatted in JSON.

FFNN:

Train size	Val Size
500	50

RNN:

Train size	Val Size
1000	800

2 Implementations (45pt)

2.1 FFNN (20pt)

I was able to successfully complete the unfinished code in ffnn.py by adding a few lines of code to the forward() function, which implements the forward pass portion of the feed-forward neural network algorithm/model:

```
def forward(self, input_vector):  Xinya Du *
    # [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
```

To explain the process of how the file successfully models the FFNN, I will briefly explain my understandings of the functions, code and tools used.

First, this function converts the documents into vectors with word frequencies of the vocabulary found in the document using the vocabulary's indices.

```
# vectorized_data = A list of pairs (vector representation of input, y)
def convert_to_vector_representation(data, word2index):  2 usages  Xinya Du
    vectorized_data = []
    for document, y in data:
        vector = torch.zeros(len(word2index))
        for word in document:
            index = word2index.get(word, word2index[unk])
            vector[index] += 1
        vectorized_data.append((vector, y))
    return vectorized_data
```

Next, the model is composed of several components that help calculate the values that result from the FFNN, as specified in the initialization statements:

```
class FFNN(nn.Module):  2 usages  Xinya Du *
    def __init__(self, input_dim, h):  Xinya Du
        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
        self.output_dim = 5
        self.W2 = nn.Linear(h, self.output_dim)

        self.softmax = nn.LogSoftmax() # The softmax function that converts vecto
        self.loss = nn.NLLLoss() # The cross-entropy/negative log likelihood loss
```

- **Self.W1:** Is the first linear layer (input -> 1st hidden layer). The linear layer is used to calculate the output with a linear equation: output = W * x + b, which is the weighted sum of inputs plus a bias. W1 maps the input to an output from the input size to the output size.
- **Self-activation**: uses ReLU (a tool included in pytorch) as a non-linearity activation function. This keeps positive values and rounds up negative values to 0, helping the model learn complex patterns better and avoiding linearity.
- **Self.W2**: is the second linear layer which converts (hidden values -> output), and is used similarly to the first linear layer.
- Finally **self.softmax:** converts raw output scores to a probabilistic value (specifically using a log-probability distribution over the output classes)

- Tools that are used include :
    - **Pytorch:** for using helpful predefined functions to help build and test the neural network (nn. Functions such as nn.Linear, nn.activation or .torch functions such as torch.argmax)
    - **TQDM:** for progress bars (aesthetic purposes)
    - **JSON:** for representing the data type files of vectors used in the NN

2.2 RNN (25pt)

I was able to successfully complete the unfinished code in rnn.py by adding a few lines of code to the forward() function, which implements the forward pass portion of the recurrent neural network algorithm/model:

```
def forward(self, inputs):  Xinya Du *
    # [to fill] obtain hidden layer representation (https://pytorc
    output, hidden = self.rnn(inputs)
    # [to fill] obtain output layer representations
    z = self.W(output)
    # [to fill] sum over output
    z_sum = torch.sum(z, dim=0)
    # [to fill] obtain probability dist.
    predicted_vector = self.softmax(z_sum)

    return predicted_vector
```

This function first runs the input vectors through the RNN layer and outputs from the RNN at every time step. A hidden layer is not used here. Next, a linear layer is applied to each time step from the RNNs output. The size is converted to 5 to represent the 5 output classes per word. Next, the class scores are averaged using torch.sum. Finally, a log softmax is used to get log probabilities over the 5 classes.

To start, I will briefly discuss the initializations that serve as the backbone to implementing the model's algorithm:

```
class RNN(nn.Module):  2 usages  Xinya Du *
    def __init__(self, input_dim, h):  # Add relevant parameters  Xinya Du
        super(RNN, self).__init__()
        self.h = h
        self.numOfLayer = 1
        self.rnn = nn.RNN(input_dim, h, self.numOfLayer, nonlinearity='tanh')
        self.W = nn.Linear(h, out_features: 5)
        self.softmax = nn.LogSoftmax(dim=1)
        self.loss = nn.NLLLoss()
```

- Input_dim is the input vector size, and h is the hidden vector size.
- Similarly to the FFNN, the output from the RNN is passed through a linear layer, and a softmax probability is applied to convert raw values to a distribution over the output classes.
- The number of layers shows that there is 1 RNN layer present in this model.
- "Tanh" is a non-linear activation function that keeps input values to range from -1 to 1, which is used more often for the cyclic nature found in RNNs, as compared to reLU (used in FFNNS).
- The final layer is converted into 5 output classes to represent the ratings from 1-5 stars.

Tools:

Similar to the FFNN, PyTorch, TQDM, and JSON are used for the neural network functions, data handling, and progress bar visualizations when testing the model performance.

Difference between FFNN and RNN:

Unlike an FFNN, which averages the vocabulary all at once, RNN processes input vectors sequentially and one-at-a-time, while maintaining the memory of the previous input through a hidden layer. RNNS also output a step at a time, unlike FFNNs, which outputs once at the end of an experiment (per epoch). Lastly, the RNN takes each time step and applies the output layer, and sums up the scores. RNNs' biggest difference with FFNNs is the implementation of a hidden layer to relay information to the next timed output.

3 Experiments and Results (45pt)

Evaluations (15pt)

To evaluate the performance of both models, I use accuracy as the primary metric, which measures the proportion of correctly classified examples out of the total number of given examples. Accuracy counts are given for both the training and validation sets, per epoch if analyzed separately, and averaged per tested combination of hidden dimension numbers. Additionally, training time is also briefly considered to compare shifts in accuracy with increasing time costs. This helps us understand computational efficiency, model performance, and efficiency trade-offs regarding changing hidden dimension sizes for the different layers.

Results (30pt)

FFNN:

For FFNN, apart from the default parameters, I experimented with an epoch size of 5 and hidden dimension sizes of 20 and 50.

Overall/Detailed Results per Epoch run:

FFNN:	Hidden Dims	Epochs	EPOCH NUM	Training Accuracy	Validation Accuracy	Train Time (s)	Val Time (s)
Baseline	10	1		**0.513**	**0.55875**	**3.7819**	**0.15949**
Test 1	20	5					
			1	0.53625	0.52875	5.99	0.18
			2	0.5845	0.58	5.61	0.17
			3	0.6185	0.565	5.66	0.17
			4	0.644875	0.57875	5.49	0.19
			5	0.657875	0.59125	5.53	0.17
			AVG	**0.6084**	**0.56875**	**5.656**	**0.176**
Test 2	50	5					
			1	0.527	0.535	35.18	0.45
			2	0.585625	0.59375	21.93	0.44
			3	0.6225	0.59625	23.78	0.4
			4	0.64525	0.59625	24.05	0.38
			5	0.651	0.5975	24.17	0.63
			AVG	**0.606275**	**0.58375**	**25.822**	**0.46**

Average/Summarized Results:

FFNN AVG/SUMMARY	Hidden Dims	Epochs	Training Accuracy	Validation Accuracy	Train Time	Val Time
Baseline	10	1	**0.513**	**0.55875**	**3.7819**	**0.15949**
Test 1	20	5	**0.6084**	**0.56875**	**5.656**	**0.176**
Test 2	50	5	**0.606275**	**0.58375**	**25.822**	**0.46**

Brief Analysis of FFNN Results:

For training accuracy, both tests did slightly better than the default, but not by much (0.6 versus the default's 0.51). For validation accuracy, however, the measured accuracy increased gradually as the number of hidden layer dimensions increased from the default's 10 to the testing HD vals (20 and 50). Overall, despite the increase in validation accuracy, the difference between the baseline and the tests were very minimal (maximum difference of about 0.03 in accuracy). The test with a higher HD value (Test2) did have the best accuracy, measured at 0.5975. However, compared to the baseline and Test 1 training times, which were about 3 and 5 seconds, Test 2 took about 5 times longer to train at about 26 seconds. This is a significant increase in training time compared to using smaller dimension sizes. The same can be said with validation times, with the default and Test1 having validation times of 0.16 and .18 seconds while Test2 took about .46 seconds, which is almost 4-5 times longer than the formerly mentioned tests.

To conclude my analysis, I would say that the training size of 20 HDS holds a good balance between aiming for a higher accuracy while not increasing training time significantly. Choosing higher values for HDS does give better accuracy but at the cost of having significantly higher training times, which may not be worth it when choosing an optimal model for testing.

RNN:

For RNN, apart from the default parameters, I experimented with a hidden dimension number of 64 (versus 32) and the same epoch count as the default (10). This is to note when the program terminates the epochs for signaling overfitting, as seen below:

```
Training done to avoid overfitting!
```

Overall/Detailed Results per Epoch run:

RNN:	Hidden Dims	Epochs	EPOCH NUM	Training Accuracy	Validation Accuracy	Train Time (s)	Val Time (s)
Baseline	32	10					
			1	0.3545	0.3675	55	1
			2	0.3299	0.2075	60	1
			3	0.3315	0.46375	53	1
			4	0.317875	0.2375	58	1
			5	0.296375	0.0725	58	1
			6	0.28275	0.27	53	1
			7	0.2934375	0.32125	53	1
			8	0.3020625	0.3025	56	1
			9	-	-	-	-
			10	-	-	-	-
			AVG	**0.31355**	**0.2803125**	**55.75**	**1**
					Best val accuracy is: 0.32125		
Test 1	64	10					
			1	0.30281	0.34625	62	1
			2	0.33025	0.2925	73	1
			3	-	-	-	-
			...	-	-	-	-
			10	-	-	-	-
			AVG	**0.31653**	**0.319375**	**67.5**	**1**
					Best val accuracy is: 0.34625		

Average/Summarized Results:

RNN AVG/SUMMARY:	Hidden Dims	Epochs	Training Accuracy	Validation Accuracy	Train Time	Val Time
Baseline	32	10	**0.31355**	**0.28031**	**55.75**	**1**
				Best: 0.32125		
Test 1	64	10	**0.31653**	**0.31938**	**67.5**	**1**
				Best: 0.34625		

Brief Analysis of RNN Results:

Between the two tests, Test 1 (HDS = 64) achieved a higher training accuracy and validation accuracy score than the baseline (HDS = 32), with a jump in validation accuracy by about 0.04. This jump in validation accuracy can be considered slightly insignificant. To compare the training and validation times, training Test1 took slightly longer (added an extra 10 seconds), while the validation times took the same time of 1 second for both baseline and Test1. Overall, there is a tradeoff with increasing the hidden dimension size for the slight increase in accuracy. However, one thing to note/ observe is this RNN model's response to possible overfitting. As mentioned earlier, the training stops at a certain epoch number to avoid overfitting. Noting this, we can see from the results table that the baseline stopped at 8 epochs while Test1 stopped at just 2 epochs, which can signal faster pattern recognition but faster overfitting. Finally in the baseline, I noticed the accuracy results for epoch 5 were unusually low, before increasing again around 7 and 8. This can show how the model is varying in training and learning patterns, or in learning to generalize.

4 Analysis (bonus: 1pt)

N/a

5 Conclusion and Others (5pt)

Group members: I worked on this assignment individually

Feedback: The difficulty was easy to moderate, the implementation was simple, and I was able to spend more time learning through analysis and testing rather than debugging code or coming up with a solution/ algorithm from scratch. This is helpful because I am learning

how PyTorch functions are used in a well-developed implementation to help build efficient neural networks. I did have a few issues at the start of the assignment when setting up my PyTorch virtual environment and configuring it to the recommended Python 3.8.x I was also experiencing issues with my Pycharm app not verifying my login in order to utilize GitHub and push my code properly. I think improvement can be from giving some guidance on which test values to choose for the hidden layer dimensions and epoch values (some useful links or insight can be briefly explained in the assignment documentation. Besides that, everything else is self-explanatory or easy to figure out on my own. The total time is about 10 hours.