ASSIGNMENT 3
FEED-FORWARD NEURAL NETWORK (FFNN)

CS 478 NATURAL LANGUAGE PROCESSING (FALL 2024)
https://nlp.cs.gmu.edu/course/cs478-fall24/

OUT: Sept 18, 2024

Checkpoint 1: Due on Sept 30, 2024
Checkpoint 2: Due on Oct 9, 2024
Checkpoint 3: Due on Oct 16, 2024

TOTAL CREDITS: 100 Points

Your name: Arron Birhanu

Your GID: G01315277

Shareable link to your notebook:
https://colab.research.google.com/drive/1jI1sYS1h-hSD3g1Xr88T7uzUrTjacktP?usp=sharing

Academic Honesty: Please note that students should complete the assignment independently. While you may discuss the assignment with other students, **all work you submit must be your own!** In addition, if you use any external resources for the assignment, you must include references to these resources at the end of this PDF. However, **you are NOT allowed to use AI assistants such as ChatGPT and Claude to complete your assignment, although using them for generic concept understanding is okay.**

Overview and Submission Guideline: In this project, you will implement an FFNN model for binary sentiment classification. The assignment also comes with a Jupyter notebook (https://jupyter.org/), which can be completed on your laptop/computer locally or on the cloud by using Google Colab (https://colab.research.google.com/), although using Colab requires extra effort in setting up the data access. **Complete the notebook and save your edits *(make sure to save the cell outputs as well)*. Create a shareable link to your notebook and copy it to the line above.**
How to fill out this PDF? In most answer blanks, you only need to provide the execution output of your code implementation. There are only a few blanks that ask you to provide your code implementation for ease of grading. When you complete the PDF, submit it to Gradescope.
What to submit for each checkpoint? For each checkpoint, the grader will check your answers for the corresponding parts.

- For Checkpoint 1, Parts 1&2 will be graded. You will submit a PDF to Gradescope.
- For Checkpoint 2, Parts 3&4 will be graded. For any code you couldn't get right in Parts 1&2, although we cannot regrade your Checkpoint 1, you are encouraged to correct them based on the grader feedback – in fact, you may find it helpful when you can move on with all mistakes in the prior parts cleared. You will submit a new PDF to Gradescope, and **the output file of your model on the blind test set to Blackboard.**

- For Checkpoint 3, Parts 5&6 will be graded. Similarly, for any code you couldn't get right in the prior checkpoints, you are encouraged to fix them in your checkpoint 3 submission. You will submit a new PDF to Gradescope.

Please do NOT submit any model checkpoints to Blackboard as they are too large to download. If the grader cannot download and open your file due to you uploading an excessively large file, we may not grade your submission.

Important Notes: (1) Please do NOT modify existing code/markdown cells in the notebook unless instructed explicitly. (2) While there may have been well-packed Python tools for producing the required FFNN model, you should not use them. Instead, you are expected to implement the model from scratch, following the instructions in the notebook. The only exceptions are commonly used general-purpose Python libraries such as `numpy` and `collections`, as they do not directly produce the FFNN model. Please consult the grader or the instructor if you have questions.

Part 1: Get to Know PyTorch (10 points)

Q1: Given the two tensors a and b, can you implement the forward pass of the neural network function and its loss, and show their values? (4 points)

Your Code Output

```
f: tensor([-12.,  65.], grad_fn=<SubBackward0>)
l: tensor(118., grad_fn=<AddBackward0>)
```

Q2: Q2: Given the values of ‘a‘ and ‘b‘ and the loss ‘l‘, what will be the gradients of them? You can simply copy your answer from the notebook here. (2 points)

Your Answer

$\frac{\partial l}{\partial a_0} = 9a_0^2 = 36$, $\frac{\partial l}{\partial a_1} = 9a_1^2 = 81$, $\frac{\partial l}{\partial b_0} = -2b_0 = -12$, $\frac{\partial l}{\partial b_1} = -2b_1 = -8$

Q3: Now, try to use PyTorch’s ‘autograd‘ to calculate the gradients automatically. (4 points)

Your Code Output

```
gradient of ‘a‘: tensor([ 36., 162.])
gradient of ‘b‘: tensor([-12., -16.])
```

Part 2: Data Structure and Loading (15 Points)

Step 1: Data structured definition and dataset loading. (5 points) Now, let’s create a vocabulary based on the training set ‘train_exs‘. Similarly as our Assignment 2, we will keep only words with more than 2 occurrences in our vocabulary, while converting others into a special UNK token.

Show your vocabulary size below:

Your Code Output

```
Number of words in the vocabulary: 7143
```

Show the word indices of the first example in the training set (i.e., ‘train_exmaple_0‘):

Your Code Output

```
The first example in the training set is indexed as:
 [2, 3, 4, 5, 6, 7, 2, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 10, 18,
    6, 19, 20, 21, 22, 23, 24, 25, 26, 27, 1, 28, 1, 29, 30, 1, 31]
```

Step 2: Implementing 'SentimentExampleBatchIterator' for batch loading. (10 points) Show your code below. Only the "TODO" part is needed. No need to copy the entire 'SentimentExampleBatchIterator' class implementation.

Your Code

```
# TODO: implement the batching process, which returns batch_inputs, batch_lengths, and batch_labels
batch_inputs = []
batch_labels = []
batch_lengths = []

# Get the maximum length in this batch for padding
max_length = max(len(ex.word_indices) for ex in batch_exs)

for ex in batch_exs:
# Collect word indices, pad to max_length
    padded_input = ex.word_indices + [self.PAD_idx] * (max_length - len(ex.word_indices))
    batch_inputs.append(padded_input)
    batch_labels.append(ex.label)
    batch_lengths.append(len(ex.word_indices))  # Original length before padding

return (torch.tensor(batch_inputs), torch.tensor(batch_lengths), torch.tensor(batch_labels))
```

Show the loaded two batches by your iterator for the sanity check:

Your Code

```
Batch 0:
tensor([[ 2,  3,  4,  5,  6,  7,  2,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17, 10,
         18,  6, 19, 20, 21, 22, 23, 24, 25, 26, 27,  1, 28,  1, 29, 30,  1, 31,
          0],
        [ 2, 32, 33, 34, 35, 12,  2, 36, 35,  2, 37, 14, 38,  4, 39, 40, 16, 20,
         41, 35, 42, 43, 44, 45, 46, 47, 48, 49, 10,  1, 50, 35,  1, 51, 10,  1,
         31]])
tensor([36, 37])
tensor([1, 1])
----------
Batch 1:
tensor([[ 1, 52, 53,  1, 20,  1, 35, 54, 55, 20, 56, 57, 58, 27, 20, 56, 59, 60,
          1,  6,  2, 61, 55, 62,  2, 63, 64, 65, 66,  2, 67, 27, 68, 27, 69, 35,
          2, 70, 31],
        [71,  2, 72,  4, 73, 74, 75, 31,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
          0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
          0,  0,  0]])
tensor([39,  8])
tensor([1, 1])
----------
```

Part 3: FFNN Model Construction (25 Points)

Show your completed code for the '__init__' function. Only the "TODO" parts are needed. No need to copy the entire class implementation (5 points).

Your Code

```
# TODO: implement a randomly initialized word embedding matrix using nn.Embedding
# It should have a size of `(vocab_size x emb_dim)`
self.word_embeddings = nn.Embedding(self.vocab_size, self.emb_dim)
# TODO: implement the FFNN architecture using nn functions
self.classifier = nn.Sequential(
            nn.Linear(self.emb_dim, self.n_hidden_units),  # Linear layer from embedding to hidden units
            nn.ReLU(),  # Activation function
            nn.Linear(self.n_hidden_units, 1)  # Final output layer for binary classification
        )
# TODO: the loss function
self.loss = nn.BCEWithLogitsLoss()  # Binary Cross Entropy loss with logits
```

Show your completed code for the ‘forward‘ function. Only the “TODO” parts are needed. No need to copy the entire class implementation (10 points).

Your Code

```
# TODO: implement
 def forward(self, batch_inputs: torch.Tensor, batch_lengths: torch.Tensor) ->
    torch.Tensor:
        """
        The forward function, which defines how FFNN should work when given a
    batch of inputs and their actual sent lengths (i.e., before PAD)
        :param batch_inputs: a torch.Tensor object of size (n_examples,
    max_sent_length_in_this_batch), which is the *indexed* inputs
        :param batch_lengths: a torch.Tensor object of size (n_examples), which
    describes the actual sentence length of each example (i.e., before PAD)
        :return: the logits of FFNN (i.e., the unnormalized hidden units before
    sigmoid) of shape (n_examples)
        """
        # TODO: implement
        embedded = self.word_embeddings(batch_inputs)  # Get embeddings
        # Average pooling over the embeddings based on the lengths
        # We will take the mean of the embeddings for each input sequence
        embedded = embedded.sum(dim=1) / batch_lengths.unsqueeze(1)  # Average
    over the lengths
        logits = self.classifier(embedded)  # Get logits from the classifier
        return logits.squeeze()  # Squeeze to get shape (n_examples,)
```

Show your completed code for the ‘batch_predict‘ function. Only the “TODO” parts are needed. No need to copy the entire class implementation (10 points).

Your Code

```
# TODO: implement
def batch_predict(self, batch_inputs: torch.Tensor, batch_lengths: torch.Tensor)
    -> List[int]:
        """
        Make predictions for a batch of inputs. This function may directly invoke
     ‘forward‘ (which passes the input through FFNN and returns the output logits
    )

        :param batch_inputs: a torch.Tensor object of size (n_examples,
    max_sent_length_in_this_batch), which is the *indexed* inputs
        :param batch_lengths: a torch.Tensor object of size (n_examples), which
    describes the actual sentence length of each example (i.e., before PAD)
        :return: a list of predicted classes for this batch of data, either 0 for
     negative class or 1 for positive class
        """
        # TODO: implement
        logits = self.forward(batch_inputs, batch_lengths)  # Get logits
        preds = (torch.sigmoid(logits) > 0.5).long().tolist()  # Apply sigmoid
    and threshold at 0.5
        return preds  # Convert predictions to list
```

Part 4: Sentiment classifier training and evaluation (15 Points)

In this part, we will start training the FFNN classifier and then evaluate it on the dev set.

Show your code implementations below:

Classifier Initialization (2 points)

```
# TODO: create the FFNN classifier
model = FeedForwardNeuralNetClassifier(
    len(vocab),
    emb_dim=300,            # Embedding size of 300 (common for GloVe embeddings)
    n_hidden_units=300     # Hidden layer size of 300
)
```

Optimizer Creation (2 points)

```
# TODO: create the optimizer
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```

Training Implementation (only the "TODO" parts; 8 points)

```
# TODO: clean up the gradients for this batch
optimizer.zero_grad()  # Reset gradients using the optimizer
# TODO: call the model and get the loss

logits = model(batch_inputs, batch_lengths)  # Forward pass to get logits
loss = model.loss(logits, batch_labels.float())  # Calculate loss

# TODO: backpropagation
loss.backward()  # Backpropagate the loss
optimizer.step()  # Update the model parameters
```

Show the best accuracy you obtained along with the epoch number (3 points):

Best Acc	Epoch
.784	9

Don't forget to run your model on the blind test set and upload the 'test-blind.output.txt' file to Blackboard!

Part 5: Exploration of FFNN hyper-parameters (20 Points)

Q5.1: In the following table, show your hyper-parameter tuning results (10 points).

Q5.2: Conduct experiments with different learning rates (but still use the Adam optimizer). (8 points)

Q5.3: What do you observe from these two experiments and why do you think that the observation would happen? (2 points)

Embedding Size	Hidden Size	Dev Acc (%)	Best Epoch
300	300	79	4
100	300	79.2	9
50	300	76.3	8
300	100	78.6	9
300	50	79.6	6

Learning Rate	Dev Acc (%)	Best Epoch
0.0001	78.6	0
0.001	78.7	4
0.01	78.7	1
0.1	75.9	0

Your Answer

From the hyper-parameter tuning, we saw observe that an increase in the embedding size and hidden layer size generally yield better results, which is likely due to the model's ability to capture more complex patterns. However, excessively large sizes due lowered this number due to overfitting. Regarding the learning rate, 0.001 provided the best performance, while too high or too low learning rates either led to unstable training or slow convergence.

Part 6: Exploration of the Pre-trained GloVe Word Embedding (15 Points)

Show your code in 'FeedForwardNeuralNetClassifierwGlove' for initializing the word embedding with GloVe (5 points):

Your Code

```
# TODO: implement the use of pre-trained word embeddings
self.word_embeddings = nn.Embedding.from_pretrained(glove_word_embeddings)
```

Show the best accuracy you obtained along with the epoch number (5 points):

Best Acc	Epoch
78.6	3

What do you observe and why could it happen? (5 points)

Your Answer

The tedious manner taken to find the accuracies and epochs of the provided .txt file were self-explanatory when considering the size of the downloaded data file (approx. 2 GB). The accuracy stayed consistent with smaller batches (approx. just over 75 percent). It's clear that this model works perfectly fine with small batches as it does with large batches.

Post-Assignment Questions (Required)

Q-P1: Did you use any resources (e.g., online tutorials, blog posts, etc.) while completing this assignment? If you did, please list them below:

Your Answer

I used pytorch documentation for part 6 to find the required parameters for nn.Embedding as well as the suggestions list provided by the built-in module in google colab. Ironically, the coding suggestions and auto-correct the Google applications provide are grounded by NLP models. Furthermore, I used an insightful video on word embeddings and the scale at which they can be controlled in a similar environment to the model we're working with.

Q-P2: Did you use AI assistants (e.g., ChatGPT) while completing this assignment? If you did, please describe how it was used. Note that you are NOT allowed to use the AI assistant's code output as yours, but it is okay to use it to assist you in generic concept understanding.

Your Answer

I intended to use AI for the purpose of learning about the nn.Embedding module. It led me to the pytorch documentation I mentioned earlier and that got me started with part 6.