Assignment 5: Subjective/Objective Sentence Classification Using Word Vectors and NLP

Original Author: Harris Chan
Deadline: Thursday, November 5, 2020 at 9pm
Late Penalty: There is a penalty-free grace period of one hour past the deadline. Any work that is submitted between 1 hour and 24 hours past the deadline will receive a 20% grade deduction. No other late work is accepted.

This assignment must be done individually. This assignment is out of **100 points**. You can find the mark associated with each major section. You will be marked based on the correctness of your implementation, your results, and your answers to the required questions in each section.

Learning Objectives

In this assignment you will:

1. See how text data is processed using the `torchtext` and `spacy` libraries
2. Make use pre-trained word vectors as a basis for classifying text
3. Implement a *basic*, a *convolutional* and a *recurrent* neural network architecture for text classification
4. Use a full train-validate-test data split.
5. Build a simple, interactive application using all three models.

What To Submit

You should hand in the following files:

- A PDF file `assignment5.pdf` containing answers to the written questions in this assignment. Graded questions are located the last section; please answer these questions and include them in your report under the appropriate section number headings.
- Your code can be submitted in two ways: Either as a single `assign5.ipynb` notebook, **or** as individual files based on the skeleton code files: `split_data.py`, `main.py`, `models.py`, `subjective_bot.py`, and **any other code files you wrote and used.**
- Your 3 saved models `model_baseline.pt`, `model_rnn.pt`, and `model_rnn.pt`.

1 Sentence Classification - Problem Definition

Natural language processing, as we have discussed in class, can provide the ability to work with the meaning of written language. As an illustration of that, in this assignment we will build models that classify a sentence as objective (a statement based on facts) or *subjective* (a statement based on opinion).

In class we have described the concept and method to convert words (and possibly groups of words) into a vector (also called an *embedding*) that represents the meaning of the word. In this assignment we will make use of word vectors that have already been created (actually, *trained*), and use them as the basis for the three classifiers that you will build. The word vectors will be brought into your program and used to convert each word into a vector.

When working from text input, we need introduce some terminology from the NLP domain: each word is first *tokenized* - i.e. made into word *tokens*. This first step has some complexity – for example, "I'm" should be separated to "I" and "am", while "Los Angeles" should be considered together as a single word/token. After tokenization each word is converted into an identifying number (which is referred to both as its *index* or simply as a *word token*). With this index, the correct word vector can retrieved from a lookup table, which is referred to as the *embedding matrix*.

These indices are passed into different neural network models in this assignment to achieve the classification of a sentence – that it is *subjective* or *objective* – as illustrated below:

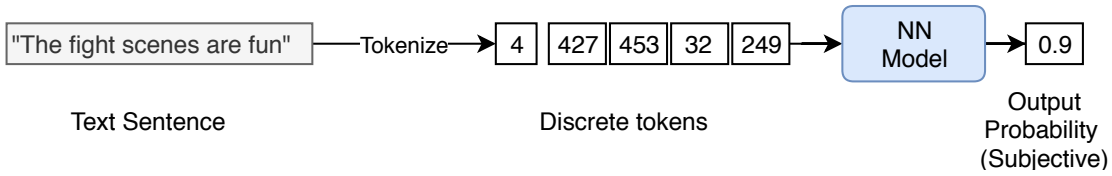

Figure 1: High Level diagram of the Assignment 4 Classifiers for Subjective/Objective

Note: the first 'layer' of the neural network model will actually be the step that converts the index/token into a word vector. (This could have been done on all of the training examples, but that would hugely increase the amount of memory required to store the examples). From the first layer on, the neural network deals only with the word vectors.

2 Setting Up Your Environment

2.1 Installing Libraries

In addition to PyTorch, we will be using two additional libraries:

- **torchtext** (`https://pytorch.org/tutorials/beginner/text_sentiment_ngrams_tutorial.html`): This package consists of both data processing utilities and popular datasets for natural language, and is compatible with PyTorch. We will be using `torchtext` to process the text inputs into numerical inputs for our models.
- **SpaCy** (`https://spacy.io/`): For 'tokenizing' English words. A text input is a sequence of symbols (letters, spaces, numbers, punctuation, etc.). The process of tokenization separates the text into units (such as words) that have linguistic significance, as described above in Section 1.

If you are using Google Colab, these packages are already installed, but you still must install the specific english library for your colab environment (every time) with the following code in the notebook (the exclamation mark causes the code to run in the shell containing your code):

```
!python -m spacy download en
```

For those using Jupyter Notebook or plain python, you'll need to install these two packages using the following commands:

```
conda install -c pytorch torchtext
conda install -c conda-forge spacy
   python -m spacy download en
```

2.2 Dataset

We will use the Subjectivity dataset [2], introduced in the paper by Pang and Lee [5]. The data comes from portions of movie reviews from Rotten Tomatoes [3] (which are assumed *all* be subjective) and summaries of the plot of movies from the Internet Movie Database (IMDB) [1] (which are assumed *all* be objective). This approach to labeling the training data as objective and subjective may not be strictly correct, but will work for our purposes.

3 Preparing the data (10 points)

3.1 Create train/validation/test splits

The data for this assignment was provided in the file you downloaded from Quercus. It contains the file `data.tsv`, which is a *tab*-separated-value (TSV) file. It contains 2 columns, `text` and `label`. The `text` column contains a text string (including punctuation) for each sentence (or fragment or multiple sentences) that is a data sample. The `label` column contains a binary value {0,1}, where 0 represents the objective class and 1 represents the subjective class.

As discussed in class, we will now use proper data separation, dividing the available data into *three* datasets: training, validation and *test*. Write a Python script (either `split_data.py` or a notebook code $\text{split}_d ata.ipynb) to split the \texttt{data.tsv} into 3 files$:

`train.tsv`: this file should contain 64% of the total data

`validation.tsv`: this file should contain 16% of the total data

`test.tsv`: this file should contain 20% of the total data

In addition, it is crucial to **make sure that there are equal number of examples in the two classes** in each of the train, validation, and test set. **Have your script/notebook print out the number in each, and provide those numbers in your report**. Use this to check and make sure the balance is there!

Finally, create a *fourth* dataset, called `overfit.tsv` also with equal class representation, that contains only 50 training examples for use in debugging your models below.

3.2 Process the input data

The `torchtext` library is very useful for handling natural language text; we will provide the basic processing code to bring in the dataset and prepare it to be converted into word vectors. If you wish to learn more detail on this, the following tutorial the includes example uses of the library: `https://medium.com/@sonicboom8/sentiment-analysis-torchtext-55fb57b1fab8`. The code described in this section is already present in the skeleton code file `main.py`.

Below is a description of the code in the skeleton `main.py` that pre-processes the data:

1. The Field object tells torchtext how each column in the TSV file will be processed when passed into the `data.TabularDataset` object. The following code instantiates two `torchtext.data.Field` objects, one for the "text" (sentences) and one for the"label" columns of the TSV data:

   ```
   TEXT = data.Field(sequential=True,lower=True,tokenize='spacy',include_lengths=True)
   LABELS = data.Field(sequential=False, use_vocab=False)
   ```

   Details: `https://torchtext.readthedocs.io/en/latest/data.html#torchtext.data.Field`

2. Next we load the train, validation, and test datasets to become datasets as was done in the previous assignments, with the torchtext method `data.TabularDataset.splits`. This method is designed specifically for text input. `main.py` uses the following code, which assumes that the `tsv` files are in the folder `data`:

   ```
   train_data, val_data, test_data = data.TabularDataset.splits(
               path='data/', train='train.tsv',
               validation='validation.tsv', test='test.tsv', format='tsv',
               skip_header=True,
               fields=[('text', TEXT), ('label', LABELS)])
   ```

   Details: `https://torchtext.readthedocs.io/en/latest/data.html#torchtext.data.TabularDataset`

3. Next we need to create an object that can be *enumerated* (Python-style) to be used in the training loops - these are the objects that produce each batch in the training loop. The objects in each batch are accessed using the `.text` field and the `.label` field that was specified in the above line.

   The iterator for the train/validation/test splits created earlier is done using the `data.BucketIterator` as shown below. This class will ensure that, within a batch, the size of the sentences will be as similar as possible, to avoid as much padding of the sentences as possible.

   ```
   train_iter, val_iter, test_iter =
       data.BucketIterator.splits((train_data, val_data, test_data),
       batch_sizes=(args.batch_size, args.batch_size, args.batch_size),
       sort_key=lambda x: len(x.text), device=None,
       sort_within_batch=True, repeat=False)
   ```

4. The `Vocab` object will contain the index (also called `word token`) for each unique word in the data set. This is done using the `build_vocab` function, which looks through all of the given sentences in the data:

```
TEXT.build_vocab(train_data,val_data, test_data)
```

Details: `https://torchtext.readthedocs.io/en/latest/data.html#torchtext.data.Field.build_vocab`

4 Baseline Model and Training (10 points)

Using the `models.py` code, you will first implement and train the *baseline* model (given below), which was discussed in class. Some of the code below will be re-usable for the other two models.

4.1 Loading GloVe Vector and Using Embedding Layer

As mentioned in Section 1, we will make use of word vectors that have already been created/trained. We will use the GloVe [6] pre-trained word vectors in an "embedding layer" (which is just that "lookup matrix" described earlier) in PyTorch in two steps:

1. (As given in the skeleton file `main.py` code) Using the vocab object from Section 3.2, item number 4, download (the first time this is run) and load the vectors that are downloaded into the vocab object, as follows:

   ```
   TEXT.vocab.load_vectors(torchtext.vocab.GloVe(name='6B', dim=100))
   vocab = TEXT.vocab
   ```

   You can see the shape of the complete set of word vectors by printing out the the shape of the vectors object as follows, which will be the number of unique words in all the training sets and the embedding dimension (word vector size).

   ```
   print("Shape of Vocab:",TEXT.vocab.vectors.shape)
   ```

   This loads word vectors into a GloVe class (see documentation `https://torchtext.readthedocs.io/en/latest/vocab.html#torchtext.vocab.GloVe`) This GloVe model was trained with six billion words to produce a word vector size of 100, as described in class. This will download a rather large **862 MB** zip file into the folder named `.vector_cache`, which might take some time; this file expands into a 3.3Gbyte set of files, but you will only need one of those files, labelled `glove.6B.100d.txt`, and so you can delete the rest (but don't delete the file `glove.6B.100d.txt.pt` that will be created by main.py, which is the binary form of the vectors). Note that `.vector_cache` folder, because it starts with a `'.'`, is typically not a visible folder, and you'll have to make it visible with an operating system-specific view command of some kind. (Windows, Mac) Once downloaded your code can now access the vocabulary object within the text field object by calling `.vocab` attribute on the text field object.

2. The step that converts the input words from an index number (a word token) into the word vector is actually done inside the nn.module model class. So, when defining the layers in your model class, you must add an embedding layer with the function `nn.Embedding.from_pretrained`, and pass in `vocab.vectors` as the argument where `vocab` is the Vocab object. The code for this is shown below in the model section, and is given in the skeleton file `models.py`.

   Details: `https://pytorch.org/docs/stable/generated/torch.nn.Embedding.html`

4.2 Baseline Model

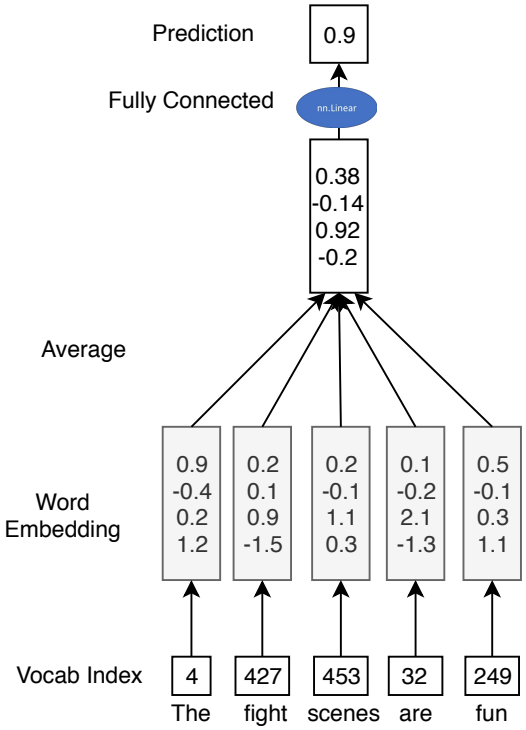

Figure 2: A simple baseline architecture

The *baseline* model was discussed in class and is illustrated in Figure 2. It first converts each of the word tokens into a vector using the GloVe word embeddings that were downloaded. It then computes the average of those word embeddings in a given sentence. The idea is that this becomes the 'average' meaning of the entire sentence. This is fed to a fully connected layer which produces a scalar output with sigmoid activation (which is computed inside the BCEWithLogitsLoss losss function) to represent the probability that the sentence is in the subjective class.

The code for this `Baseline` class is given below, and is also provided in the skeleton file `models.py`. Read it and make sure you understand it.

```
class Baseline(nn.Module):
    def __init__(self, embedding_dim, vocab):
```

```
        super(Baseline, self).__init__()

        self.embedding = nn.Embedding.from_pretrained(vocab.vectors)
        self.fc = nn.Linear(embedding_dim, 1)

    def forward(self, x, lengths=None):
        #x has shape [sentence length, batch size]

        embedded = self.embedding(x)

        average = embedded.mean(0) # [sent len, batch size, emb dim]

        output = self.fc(average).squeeze(1)

        # Note - using the BCEWithLogitsLoss loss function
        # performs the sigmoid function *as well* as well as
        # the binary cross entropy loss computation
        # (these are combined for numerical stability)

        return output
```

4.3 Training the Baseline Model

Using the `main.py` code, write a training loop to iterate through the training dataset and train the baseline model. Use the hyperparameters given in Table 1. Note that we have not used the `Adam` optimizer yet in this course; it will be discussed in a later lecture. The Adam optimizer is invoked the same way as the `SGD` optimizer, using `optim.Adam`.

Hyperparameter	**Value**
Optimizer	Adam
Learning Rate	0.001
Batch Size	64
Number of Epochs	25
Loss Function	BCEWithLogitsLoss()

Table 1: Hyperparameters to Use in Training the Models

The objects `train_iter, val_iter, test_iter` and `overfit_iter` described in Section 3.2 are the iterable objects that will produce the batches of `batch_size` in each training inner loop step. The `torchtext.data.batch.Batch` object is given by the iterator, from which you can obtain both the text input and the length of the sentence sequences from the `.text` field of the `Batch` object, as follows, assuming that `batch` is the object returned from the iterator:

```
batch_input, batch_input_length = batch.text
```

Where `batch_input` is the set of text sentences in the batch.

The details on this object can be found in `https://spacy.io/usage/spacy-101#annotations-token`.

4.4 Overfitting to debug

As was done in Assignment 3, debug your model by using *only* the very small `overfit.tsv` set (described above, which you'll have to turn into a dataset and iterator as shown in the given code), and see if you can *overfit* your model and reach a much higher training accuracy than validation accuracy. (The baseline model won't have enough parameters that you can get an accuracy of 100%; the cnn and rrnn models will have enough). You will need more than 25 epochs to succeed in overfitting. Recall that the purpose of doing this is to be able to make sure that the input processing and output measurement is working. So, do not proceed into the next sections until you've achieved 100% training accuracy, because it will be harder to debug with more data, and whatever the problem is can more easily be found here (such as mislabelled data, or errors in the data handling; for more suggestions on types of errors, see Tutorial 4 from October 6.

It is also recommended that you include some useful logging in the loop to help you keep track of progress, and help in debugging.

Provide the training loss and accuracy plot for the overfit data in your Report.

4.5 Full Training Data

Once you've succeeded in overfitting the model, then use the full training dataset to train your model, using the hyper-parameters given in Table 1.

Using your `main.py` code (in notebook or raw python form) write an evaluation loop to iterate through the validation dataset to evaluate your model. We recommend that you call the evaluation function in the training loop (perhaps every epoch or two) to make sure your model isn't overfitting. Keep in mind if you call the evaluation function too often, it will slow down training.

Give the training and validation loss and accuracy curves vs. epoch in your **Report**, and report the final test accuracy. Evaluate the test data and provide the accuracy result in your **Report**.

4.6 Saving and loading your model

In your code, save the model with the lowest validation error with `torch.save(model, 'model_baseline.pt')`. You will need to submit this file. See `https://pytorch.org/tutorials/beginner/saving_loading_models.html` for detail on saving and loading.

5 Convolutional Neural Network (CNN) (10 points)

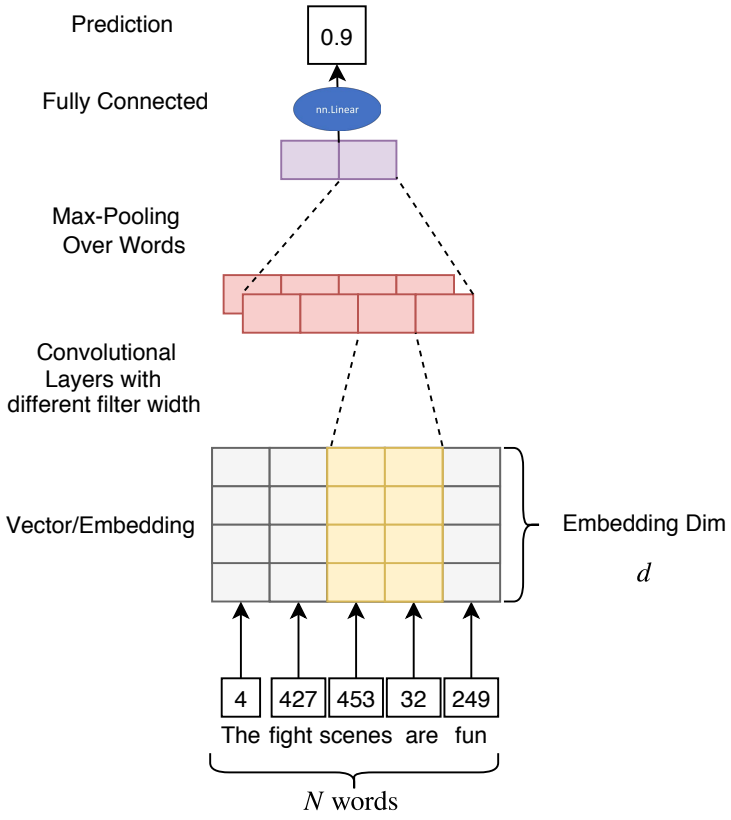

Figure 3: A convolutional neural network architecture

The second architecture, described in class and illustrated in Figure 3, is to use a CNN-based architecture that is inspired by Yoon et al. [4]. Yoon first proposed using CNNs in the context of NLP. You will write the code for the `CNN` model class within the `models.py` starter code with the following specifications:

1. Group together all the vectors of the words in a sentence to form a `embedding_dim * N` matrix. Here `N` is the number of words (and therefore tokens) in the sentence. Different sentences will have different lengths, which is unusual for a CNN but that will be dealt with in the final pooling step. Note that `embedding_dim` is the size of the word vector, 100.

2. The architecture consists of two convolutional layers that both operate on the word vector group created above, but with different kernel sizes. The kernel sizes are `[k, embedding_dim]`, and you should use the following values for $k \in 2, 4$. Use 50 kernels for each of the two kernel sizes. Note that this organization of convolutional layers is different from your prior use of CNNs in which one layer fed into the next; these are operating on the same input. Note also, that, even though the kernel sizes span the entire embedding dimension, you can still use the nn.conv2d method, and explicitly specify the size of a kernel using the `kernel_size=(kx,ky)` notation.

3. Use the ReLU activation function on the convolution output.

4. To handle the variable sentence lengths, we perform a `MaxPool` operation on the convolution layer output (after activation), along the sentence length dimension. That is, compute the maximum across the entire sentence length, and get one output feature/number from each sentence for each kernel.

5. Concatenate the outputs from the maxpool operations above to form a fixed length vector of dimension 100 – because each of the two kernel sizes is used 50 times each in the two different convolutional layers.

6. Finally, similar to the baseline architecture, use a fully connected layer to a scalar output with sigmoid activation to represent the probability that the sentence is in the subjective class. (Recall that the BCEwithLogitsLoss function computes the sigmoid as part of the loss; to actually determine the probability when printing out an answer, you'll need to separately apply a sigmoid on the neural network output value.

5.1 Overfit, Training and Test

Once you've created the code for the CNN model, follow the same processes described in Sections 4.3, 4.4, 4.5 and 4.6, except change the file name of the saved model to be `model_cnn`.

6 Recurrent Neural Network (RNN) (10 points)

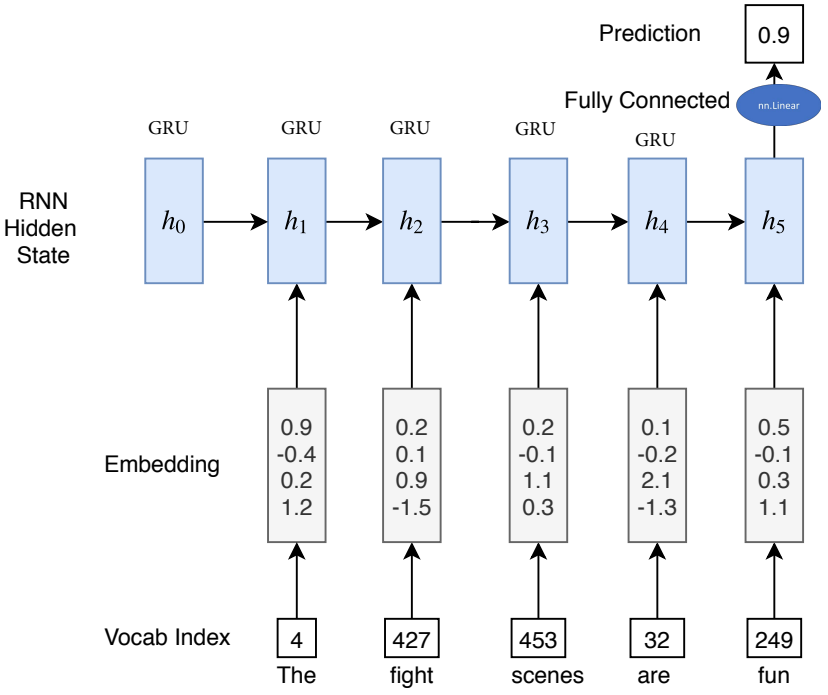

Figure 4: A Recurrent Neural Network Architecture

The third architecture, illustrated in Figure 4, is to use a Recurrent Neural Network (RNN)-based architecture. A recurrent neural network has a hidden state h_0 that is initialized at the start of a sequence of inputs (typically to all zeroes). The network takes in the input - the vector corresponding to a word in the sentence, x_t at each 'step' of the sequence, as illustrated in Figure 4. It computes the new hidden state as a function of the previous hidden state and input the word vector. In this way the newly computed hidden state retains information from the previous inputs and hidden states, as expressed in this equation:

$$h_t = f(h_{t-1}, x_t) \tag{1}$$

The final hidden state (h_T, where T is the number of words in the sentence), is produced after the full sequence of words is processed, and *is a representation of the sentence* just as the average produced in the baseline above is a representation of the sentence. Similar to the baseline, we then use a fully connected layer to generate a single number output, together with sigmoid activation to produce the probability that the sentence is in the subjective class.

Here are some guidelines to help you implement the `RNN` model in `models.py` file:

1. You should use the Gated Recurrent Unit (GRU) as the basic RNN cell, which will fulfill the function of the blue boxes in Figure 4. The GRU takes in the hidden state and the input, and produces a new hidden state. In the `__init__` function of your `RNN` model, use the `nn.GRU` cell (from Pytorch). Set the **embedding dimension to be 100** (as that is the size of the word vectors) and select the **hidden dimension to be 100**.

2. As usual, during training, we send *batches* of sentences through the network at one time, with just one call to the model forward function. One issue with this is that the sentence lengths will differ within one batch. The shorter sentences are padded from the end onward to the longest sentence length in the batch. PyTorch's GRU module can take in a batch of several sentences and return the hidden states for all of the (words $\times$ batch size) in one call without a for-loop, as well as the last hidden states (which is the one we use to generate the answer).

   However, there is a problem if you simply use the last hidden states returned: for the shorter setences, these will be the wrong hidden states, because the sentence ended earlier, as shown on the left side of Figure 5. Instead, use the `nn.utils.rnn.pack_padded_sequence` function (see documentation `https://pytorch.org/docs/stable/nn.html?highlight=pack_padded_sequence#torch.nn.utils.rnn.pack_padded_sequence`) to pack the word embeddings in the batch together and run the RNN on this object. The resulting final hidden state from the RNN will be the correct final hidden state for each sentence (see Figure 5 (Right)), not simply the hidden state at the maximum length sentence for all sentences.

6.1 Overfit, Training and Test

Once you've created the code for the RNN model, follow the same processes described in Sections 4.3, 4.4, 4.5 and 4.6, except change the file name of the saved model to be `model_rnn`.

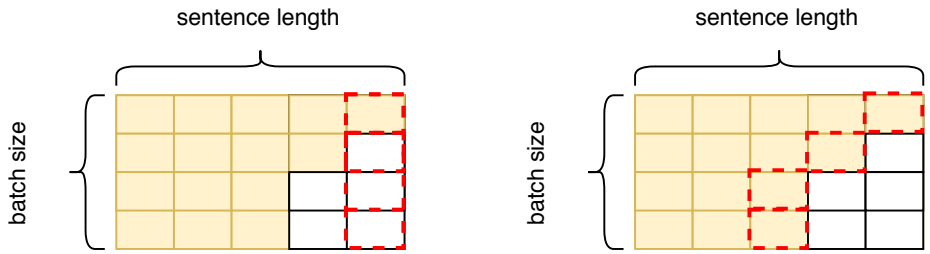

Figure 5: In a batch, each square represents the RNN hidden state at different words in the sequence (the columns) and input example in the batch (the rows). The shaded cells represent that there is an input token at that word in the sequence, while a white filled cell indicates a padded input (with a word vector containing zeroes). **Left**: If we naively took the hidden at the last column (at the maximum time step), then we will be getting the hidden state when the RNN has been fed padding vectors (zeroes). **Right**: We want to get the RNN hidden state at the last word for each sequence.

7 Testing on Your Own Sentence (10 points)

In this section, you will write a Python script `subjective_bot.py` or notebook `subjective_bot.ipynb` that prompts the user for a sentence input on the command line, and prints the classification from each of the three models, as well as the probability that this sentence is subjective. This was demonstrated in class, in Lecture 20. Specifically, the 'bot' should:

1. Print "`Enter a sentence`" to the console, then on the next line, the user can type in a sentence string (with punctuations, etc.). (Hint: you can use the built-in `input()` function; also use `import readline` when you do that.)
2. For each model trained, print to the console a string in the form of "`Model [baseline|rnn|cnn]: [subjective|objective] (x.xxx)`", where `x.xxx` is the prediction probability that the sentence is subjective in the range [0,1] up to 3 decimal places.
3. Print "`Enter a sentence`" prompt again, in an infinite loop until the user decides to terminate the Python program.

An example output on the console is given below:

```
Enter a sentence
What once seemed creepy now just seems campy

Model baseline: subjective (0.964)
Model rnn: subjective (0.999)
Model cnn: subjective (1.000)

Enter a sentence
```

The script can be broken down into several steps that you should implement:

1. Obtain the `Vocab` object by performing the same preprocessing that was done in Part 3. This object will convert the string tokens into integer.

2. Load the saved parameters for models you've trained: `model = torch.load('filename.pt')`

3. You'll need a `tokenizer` function, which takes in a string input and converts the words to tokens using the SpaCy as follows:

```
import spacy
def tokenizer(text):
    spacy_en = spacy.load('en')
    return [tok.text for tok in spacy_en(text)]
```

   To convert the sentence string that has been input:

```
tokens = tokenizer(sentence)
```

   Details: `https://spacy.io/usage/spacy-101#annotations-token`

4. To convert each string token to an integer, use the `.stoi` variable, which is a dictionary with string as the key and integer as the value, in the `Vocab` object as done here:

```
token_ints = [vocab.stoi[tok] for tok in tokens]
```

   Details: `https://torchtext.readthedocs.io/en/latest/vocab.html#torchtext.vocab.Vocab`

5. Convert the list of token integers into a `torch.LongTensor` with the shape `[L,1]`, where `L` is the number of tokens:

```
token_tensor = torch.LongTensor(token_ints).view(-1,1) # Shape is [sentence_len, 1]
```

6. Create a tensor for the length of the sentence with the shape `[1]`:

```
lengths = torch.Tensor([len(token_ints)])
```

   This will be needed when calling the RNN model.

7. You can convert the torch Tensor into a numpy array by calling `.detach().numpy()` on the torch tensor object before printing so that the print formatting will match the examples.

8 Experimental and Conceptual Questions (50 Points)

1. (5 points) After training on the three models, report the loss and accuracy on the train/validation/test in a total. There should be a total of 18 numbers. Which model performed the best? Is there a significant difference between the validation and test accuracy? Provide a reason for your answer.

2. (5 points) In the baseline model, what information contained in the original sentence is being ignored? How will the performance of the baseline model inform you about the importance of that information?

3. (15 points) For the RNN architecture, examine the effect of using `pack_padded_sequence` to ensure that we did indeed get the correct last hidden state (Figure 5 (Right)). Train the RNN and report the loss and accuracy on the train/validation/test under these 3 scenarios:

   (a) Default scenario, with using `pack_padded_sequence` and using the `BucketIterator`

   (b) Without calling `pack_padded_sequence`, and using the `BucketIterator`

   (c) Without calling `pack_padded_sequence`, and using the `Iterator`. What do you notice about the lengths of the sentences in the batch when using `Iterator` class instead?

   Given the results of the experiment, explain how you think these two factors affect the performance and why.

4. (10 points) In the CNN architecture, what do you think the kernels are learning to detect? When performing max-pooling on the output of the convolutions, what kind of information is the model discarding? Compare how this is different or similar to the baseline model's discarding of information.

5. (10 points) Try running the `subjective_bot.py` script on 4 sentences that you come up with yourself, where 2 are definitely objective/subjective, while 2 are borderline subjective/objective, according to your opinion. **Include your console output in the write up**. Comment on how the three models performed and whether they are behaving as you expected. Do they agree with each other? Does the majority vote of the models lead to correct answer for the 4 cases? Which model seems to be performing the best?

6. (5 points) Describe your experience with Assignment 4:

   (a) How much time did you spend on Assignment 4?

   (b) What did you find challenging?

   (c) What did you enjoy?

   (d) What did you find confusing?

   (e) What was helpful?

References

[1] Internet movie database. `https://www.imdb.com/`. Accessed: 2018-09-15.

[2] Movie review data. `https://www.cs.cornell.edu/people/pabo/movie-review-data/`. Accessed: 2018-09-15.

[3] Rotten tomatoes. `https://www.rottentomatoes.com/`. Accessed: 2018-09-15.

[4] Yoon Kim. Convolutional neural networks for sentence classification. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1746–1751. Association for Computational Linguistics, 2014.

[5] Bo Pang and Lillian Lee. A sentimental education: Sentiment analysis using subjectivity. In *Proceedings of ACL*, pages 271–278, 2004.

[6] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global vectors for word representation. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, 2014.