Assignment 4: Subjective/Objective Sentence Classification Using Word Vectors and NLP

Deadline: Thursday, November 8, 2018 at 11:59pm

This assignment must be done individually. This assignment is out of **100 points**. You can find the mark breakdown in each 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. Learn to process text data using the `torchtext` library
2. Use a pre-trained word embedding for your neural network model
3. Implement a *basic*, *recurrent* and *convolutional* neural network architectures for text classification
4. Do a full train-validate-test data split.
5. Build a simple, interactive application using the three models.

What To Submit

You should hand in the following files:

- A PDF file `assignment4-writeup.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 headings.
- The code files `split_data.py`, `main.py`, `models.py`, `subjective_bot.py`, which should make use of the bare-bones structure given in the starter package `assign4.zip` provided in this assignment, 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 it 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 try to determine if a sentence is objective (a statement based on facts) or *subjective* (a statement based on opinions of the speaker/writer).

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 be produce (recall, actually, trained), and use them as the basis for the classifiers you will build, and simply import a lookup table (a matrix) that converts a known word into a vector.

When working from text input, we need introduce a little terminology from the NLP domain: each word is first *tokenized*, and converted to an identifying number (called its *index*). This is a little more complicated than it might seem, and is not just simply splitting by white spaces. For example, "I'm" should be separated to "I" and "am", while "Los Angeles" should be considered together as a single word/token. Wth the right index for the token/word, the appropriate vector is retrieved from the lookup table, which is also called an *embedding matrix*.

It is these vectors that you will pass into three different kinds of Neural Networks in this assignment to achieve classification of subjective or objective, as illustrated below:

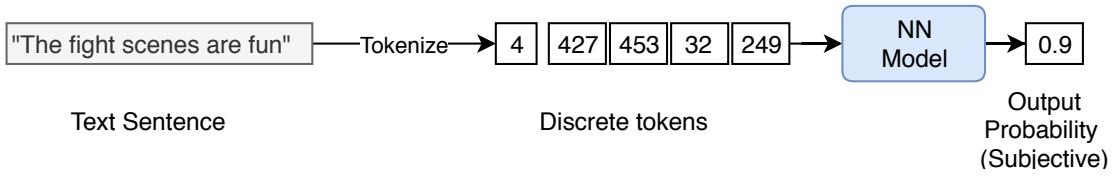

Figure 1: High Level diagram of the Assignment 4 Classifiers for Subjective/Objective

2 Setting Up Your Environment

2.1 Installing Libraries

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

- **torchtext** (link): This package consists of data processing utilities and popular datasets for natural language, compatible with PyTorch. We will be using `torchtext` to process the text inputs into a numerical inputs for our models.
- **SpaCy** (link): For 'tokenizing' English words. A text input is a sequence of symbols (letters, space, numbers, punctuations, etc.). The process of tokenization segments the text into units (such as words) that have linguistic significance, as described above in Section 1.

Install the two packages using the following commands:

```
pip install torchtext 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 broad choice of labelling - 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 (3 points)

In the assignment zip file, you are given `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 punctuations) for the portions of a sentence. The `label` column contains a binary value {0,1}, where 0

represents the objective class while 1 represents the subjective.

As discussed in class, we will now move to doing proper data separation, into three datasets, training, validation and test. Write a Python script `split_data.py` to split the `data.tsv` into 3 files:

1. `train.tsv`: this file should contain 64% of the total data
2. `validation.tsv`: this file should contain 16% of the total data
3. `test.tsv`: this file should contain 20% of the total data

In addition, make sure that there are equal number of examples between the two classes in each of the train, validation, and test set, **and have your script print out the number in each, and provide those numbers in your report**.

3.2 Process the input data (7 points)

The `torchtext` library is very useful for dealing with input that is natural language text. Before coding with this library, we suggest readin this useful tutorial the includes example uses of the library: `http://anie.me/On-Torchtext/` [This Section needs to be improved.]

Implement the following inside `main.py` to preproces the data:

1. The Field object tells torchtext how each column in the TSV file will be process when passed onto the `data.TabularDataset` object. Instantiate two `torchtext.data.Field` objects, for the "text" and "label" columns of the TSV data. For the text field, these arguments should be set to `True`: `sequential`, `include_lengths`, and the `tokenize` parameter should be set to the string `'spacy'`. For the label field, only `sequential=False` and `use_vocab=False` needs to be explicitly specified. See the documentation on Field: link
2. Load the train, validation, and test splits using the method `data.TabularDataset.splits`. You will need to provide the `path`, `train`, `validation`, `test`, `skip_header`, `format`, and `fields` arguments. See documentation on TabularDataset link.
3. Create a batch iterator for the train/validation/test splits created earlier. Use the `data.BucketIterator` class. 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 in a batch. You will need to provide the values for the arguments `sort_key`, `sort_within_batch=True`, and `repeat=False`.
4. To create a `Vocab` object for for the text field object, call the `build_vocab` function (see documentation link) on the text Field object, passing in as argument the `train_data` from the `TabularDataset` earlier.

4 Defining the Models (24 points)

In the `models.py` file, you will implement the following three architectures for sentence classification: (1) baseline model, (2) a recurrent neural network, and (3) a convolutional neural network.

4.1 Loading GloVe Vector and Using Embedding Layer (4 points)

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" in PyTorch in two steps [This section needs to be improved]:

1. (In `main.py`) Take the vocab object from Section 3.2, item number 4, and call the `load_vector` method (See documentation link), passing in a GloVe class (see documentation link). Use a GloVe model with the **name `6B` and an embedding size of `100` dimensions**. This will download a rather large **822 MB** file into `./.vector_cache` folder, which might take some time. You can now access the vocabulary object within the text field object by calling `.vocab` attribute on the text field object.

2. (In `models.py`) When defining the layers in your model class (`nn.Module`), define an embedding layer with the function `nn.Embedding.from_pretrained`, and pass in `vocab.vectors` as the argument where `vocab` is the Vocab object for the text field. See documentation on `Embedding` class here: link

4.2 Baseline Model (4 points)

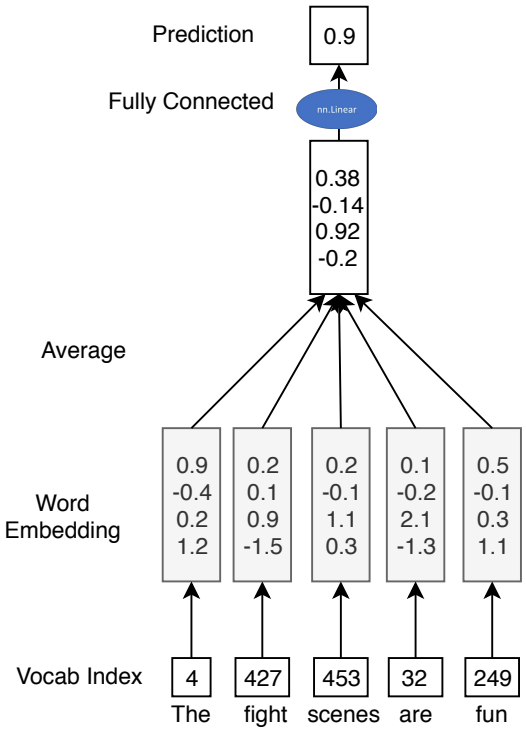

Figure 2: A simple baseline architecture

In the first model, called the *baseline* model and illustrated in Figure 2, we first convert each of the word-tokens into a vector using the pretrained GloVe word embedding. Then, you will simply take the average of those word embeddings, which gives the 'average' meaning of the entire sentence. This is fed to a fully connected layer which produces a scalar output with sigmoid activation to represent the probability that the sentence is in the subjective class. Of course you'll be training

this network to do that well!

Implement this `Baseline` class in `models.py` file.

4.3 Recurrent Neural Network (RNN) (8 points)

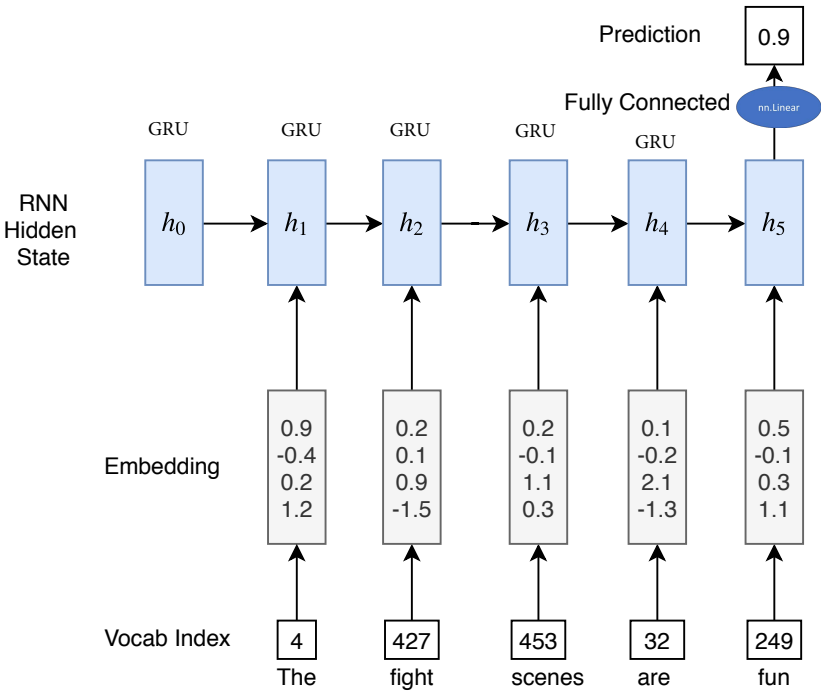

Figure 3: A recurrent neural network architecture

The second architecture, illustrated in Figure 3, 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). Then 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 3 and computes the new hidden state as a function of the previous hidden state and the word vector. In this way the newly computed hidden state retains information from the previous 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 essentially fulfill the function of the blue boxes in 3. The GRU takes in the hidden state and the input,

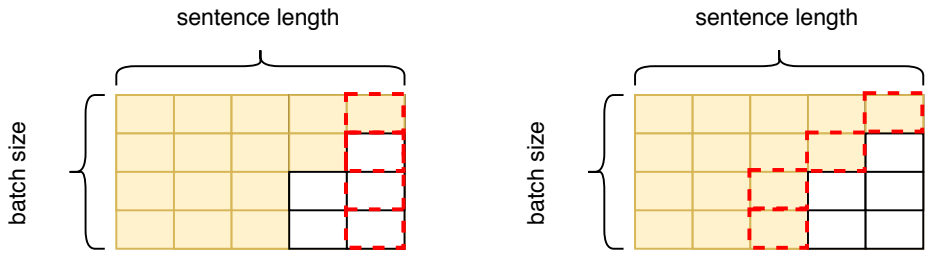

Figure 4: In a batch, each square represents the RNN hidden state at different words in the sequence (the columns) and input example in the batch (the rows). The shaded cells represent that there is an input token at that word in the sequence, while a white filled cell indicates a padded input (with zero). **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.

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, this will be the wrong hidden states, because the sentence ended earlier, as shown on the left side of Figure 4. Instead, you can use PyTorch's `nn.utils.rnn.pack_padded_sequence` function (see documentation link) 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 4 (Right)), not simply the hidden state at the maximum length sentence for all sequences.

4.4 Convolutional Neural Network (CNN) (8 points)

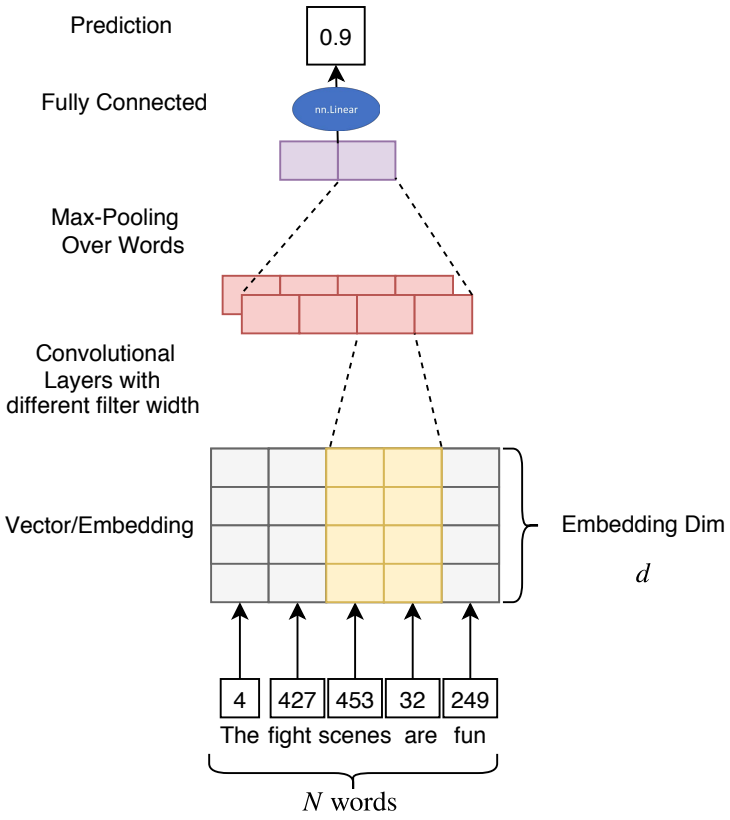

Figure 5: A convolutional neural network architecture

The third architecture, illustrated in Figure 5, is to use a CNN-based architecture. You will implement an architecture inspired by Yoon et al. [4]. Implement the `CNN` class in `models.py` file with these specifications:

1. Group together all the vectors of the words in a sentence - the vectors of those words - to form a `embedding_dim x num_token` matrix. Note that `embedding_dim` is the size of the word vector, 100.
2. The architecture consists of convolutional layers where the kernel sizes are `[k, embedding_dim]`. Use the following values for $k \in 2, 4$ and use the number of filters as 50 for each kernel size.
3. Use the activation function for the convolution layer is ReLU.
4. To handle the variable sentence lengths, we perform a `MaxPool` operation on the convolution later activations, along the sequence length dimension.
5. We concatenate the activations from different kernel sizes together to form a fixed length vector of dimension 100.
6. Similarly to the previous two architectures, we use a fully connected layer to a scalar output with sigmoid activation to represent the probability that the sentence is in the subjective class

5 Training & Evaluation (9 points)

5.1 Training (4 points)

In `main.py` write a training loop to iterate through the training dataset and train your 3 models. It is recommended that you include some useful logging in the loop to help you keep track of progress, and help in debugging. You should use the following hyperparameters:

Hyperparameter	Value
Optimizer	Adam
Learning Rate	0.001
Batch Size	64
Epochs	25

Table 1: Hyperparameters for training the models

Note: When you obtain a `torchtext.data.batch.Batch` object from the iterator, you can access the actual text input and the length of the sentence sequences by calling `.text` on the `Batch` object. See documentation here: link

5.2 Evaluation and Test (4 points)

In `main.py` write an evaluation loop to iterate through the validation dataset to evaluate your model. It is recommended 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.

Evaluate the test data, for reporting as described in Section 7

5.3 Saving and loading your model (1 point)

In `main.py`, for each architecture, save the model with the lowest validation error with `torch.save(model, 'filename.pt')`, where the file name should be in the form `model_[baseline|rnn|cnn]`, such as `model_baseline`. You will need to load these files in the next section.

6 Testing on Your Own Sentence (7 points)

Let's have some fun with our model! In this section, you will write a Python script `subjective_bot.py` 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 22.

Specifically, running the command `python subjective_bot.py` will:

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)
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 back the saved parameters for the 3 models that you have trained
3. Implement a `tokenizer` function, which takes in a string input and outputs a list of string tokens. Use the SpaCy tokenizer, calling the `nlp = spacy.load('en')` then use `nlp` to tokenize the sentence. An example can be seen in the SpaCy documentation: link.
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. See documentation here: link.
5. Convert the list of token integers into a `torch.LongTensor` with the shape `[L,1]`, where `L` is the number of tokens.
6. Create a tensor for the length of the sentence with the shape `[1]`. 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.

7 Grading 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 4 (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 in the convolution activations, 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.