Assignment 2: Subjective/Objective Sentence Classification Using MLP and CNN

Deadline: Tuesday, October 11, 2022 at 9:00pm
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. 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. Make use of pre-trained word vectors as a basis for classifying text
2. Implement an Multi-Layer Perceptron and a Convolutional Neural Network architecture for text classification. You'll explore some relevant hyperparameters, and look at the resulting trained features for insights.
3. Practice doing qualitative analysis of results.
4. Explore quantitative methods to explain the results.
5. Build an interactive application using **Gradio** `https://gradio.app`.

What To Submit

You should hand in the following files:

- A PDF file `assignment2.pdf` containing answers to the written questions in this assignment. You should number your answer to each question in the form `Question X.Y.Z`, where X is the section of this assignment, Y is the subsection, and Z is the numbered element in the question. **You should include the specific written question itself** and then provide your answer.
- You should submit separate code and model files for the code and models that you wrote as specified in Sections 5.1, 5.5 and 6.2.

1 Subjective/Objective Classification Problem Definition

The first lectures of this course, and the first assignment showed how textual words can be turned into word vectors that represent the meaning of a word. These form the basis of all forms of modern deep-learning based NLP. 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). To begin, make sure you understand the distinction between these two words - look up a few definitions of

these words, when used as adjectives.

We will make use of word vectors that have already been created (as you now know, *trained*), and use them as the basis for the two classifiers that you will build.

As in Assignment 1, each word is first *tokenized* in which each word (or portion of a 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 word vector corresponding to the word can be retrieved from a lookup table, which is referred to as the *embedding matrix*.

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

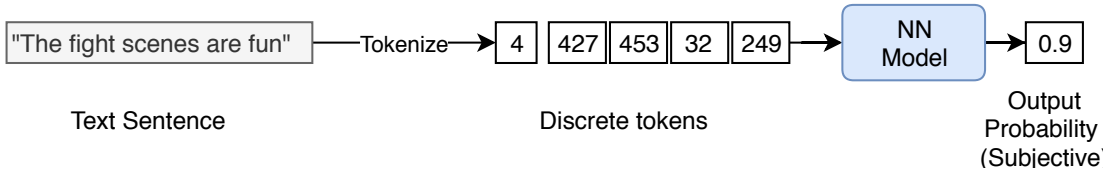

Figure 1: High Level diagram of the Assignment 2 Classifiers for Subjective/Objective

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 greatly 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 Software Environment and Dataset

2.1 Software Environment

The software environment you'll be using in this assignment is the same as that used in Assignment 1, including **PyTorch**, **torchtext** and **SpaCy**. The document given as part of Assignment 1, "Course Software Infrastructure and Background" tells you what to use and install.

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. Note that some of these sentences may be labelled incorrectly, and discovering those incorrect labels is part of this assignment.

3 Preparing the data (5 points)

3.1 Human Data Review

The data for this assignment was provided in the file `data.tsv` that you downloaded from Quercus. This is a *tab*-separated-value (TSV) file that contains two 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.

Do/answer each of the following questions:

1. Review the data to see how it is organized in the file. How many examples are in the file `data.tsv`?
2. Select two random examples each from the positive set (subjective) and two from the negative set. For all four examples, explain, in English, why it has the given label. [1 point]
3. Find one example from each of the positive and negative sets that you think has the incorrect label, and explain why each is wrong [2 points].

3.2 Create train/validation/test splits and Overfit Dataset

You will need to divide the available data into *three* main datasets: training, validation and test (make sure you know why all model training needs three datasets, and not 2!). You can use `train_test_split` from the `sklearn` library (or some other splitter if you prefer) 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

Finally, create a *fourth* dataset, called `overfit.tsv` also with equal class representation, that contains only 50 training examples for use in debugging, as discussed in Section 4.4 below. This dataset will be used separately from the other three, so it can overlap with those datasets.

You should (programmatically) verify that there are equal number of examples in the two classes in each of the datasets, and that you did not accidentally put the same sample in more than one of the training, validation and test sets. Your code should report both of these checks. Submit your code to perform these functions in the file `A2P3_2.py`. [2 points]

3.3 Processing the Training Data

We have provided starter code that processes the training, validation and test data for you, once it has been split into the files described above. It can be found in the `A2_Starter.py` given with this assignment. It uses the Pytorch `DataLoader` class to process the input data in this assignment, as described in this tutorial. The code described below is present in the skeleton code file `A2_Starter.py`. Read this section together with the code, and make sure you understand what the code is doing.

1. The `torchtext.vocab` loads the GloVe vectors, in a manner similar to Assignment 1.

```
glove = torchtext.vocab.GloVe(name="6B", dim=args.emb_dim)
```

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 `A2_Starter.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.

The `glove` object contains the index (also called `word token`) for most words in the data set. For those words that do not occur in GloVe's vocabulary range, we substitute it for the last word in the vocabulary.

2. Next the code below (and is given in the starter code) loads the train, validation, and test datasets to become datasets using `TextDataset`, a class extending the pytorch's `Dataset`. This method is designed specifically for text input. `A2_Starter.py` uses the following code, which assumes that the `tsv` files are in the folder `data`:

```
train_dataset = TextDataset(glove, "train")
val_dataset = TextDataset(glove, "validation")
test_dataset = TextDataset(glove, "test")
```

Additional details on the Pytorch Dataset class can be found at: `https://pytorch.org/tutorials/beginner/basics/data_tutorial.html` Later, in Section 4.4 you'll also need to use "overfit" dataset you created in Section 3.2.

3. Next we need to create an object, `DataLoader`, that can be *enumerated* (Python-style) to be used in the training loops. The objects in each batch is specified by the "collate function", `collate_fn`. This code is given to you in the starter code, `A2_Starter.py`. (You can review data loading in the Pytorch documentation to learn more about the DataLoader, and the function `collate_fn`, but the code is given to you.)

   These DataLoaders are iterable objects that will produce the batches of `batch_size` samples in each training inner loop step. In the given implementation of the collate function, a tuple of two elements will be returned in each batch: An integer `torch.tensor` of size [length x batch_size], and a float `torch.tensor` of size [batch_size].

   Recall that, for the CNN model, the input strings of a given batch should all be of the same length, and so shorter strings are padded with zeroes. The collate function returns a *tokenized* and *padded* batch of text. Tokenization is done simply by splitting into words using whitespace, and looking up the token using `vocab.stoi.get`.

4 Baseline Model and Training (10 points)

Using the above starter code, design and train the baseline model described below.

4.1 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 as an embedding matrix. The step that converts the input words from an index number (the word token) into the word vector is usually done inside the `nn.Module` model class. So, when defining the layers in your model class, you must add a layer with the function `nn.Embedding.from_pretrained`, and pass in `vocab.vectors` as the argument where `vocab` is the Vocab object.

```
self.embedding = nn.Embedding.from_pretrained(vocab.vectors)
```

There is an optional argument, 'freeze' (default=True) in the 'from_pretrained' method. When set to False, the embedding vectors themselves are trainable together with the model's parameters. Details: `https://pytorch.org/docs/stable/generated/torch.nn.Embedding.html`. We will make use of this feature later, in Section 5.3.

4.2 Baseline Model

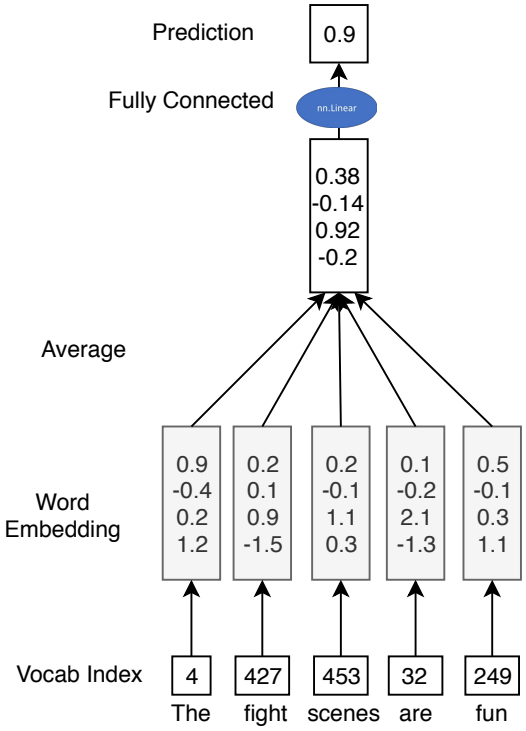

Figure 2: A simple baseline architecture

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 should be computed inside the BCEWithLogitsLoss losss function) to represent the probability that the sentence is in the subjective class.

4.3 Training the Baseline Model

Using the `A2_Starter.py` code, write a training loop to iterate through the training dataset and train the baseline model. Use the hyperparameters given in Table 1, but find a suitable batch size.

Hyperparameter	**Value**
Optimizer	Adam
Learning Rate	0.001
Number of Epochs	50
Loss Function	BCEWithLogitsLoss()

Table 1: Hyperparameters to Use in Training the Models

4.4 Overfitting to debug

One very useful way to debug your model is to see if you can *overfit* your model and reach a much higher training accuracy than validation accuracy. Use the `overfit.tsv` file that you created earlier to do this. The purpose of doing this is to be able to make sure that the input processing and output measurement is working. You will need more than 25 epochs to succeed in overfitting. (Note that it is hard to overfit the baseline model to get an accuracy of 100% because it doesn't have that many parameters; the CNN model in the next section will have enough to reach 100%). So, **do not** proceed into the next sections until you have some overfitting, 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).

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. [1 point]

4.5 Full Training Data

Now you should use the full training dataset to train your model, using the hyper-parameters given in Table 1. Although this model will get better with more Epochs, 50 is enough to illustrate what it is doing.

Using your `A2_Starter.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**.

Answer this questions: In the baseline model, what information contained in the original sentence is being ignored? [1 points]

4.6 Extracting Meaning from the Trained Parameters

The dimension of the parameters in the linear neuron is the same size as the word embedding, which suggests that there is a meaning attributable to the learned parameters. You can explore that meaning using the function `print_closest_cosine_words` from Assignment 1. Use that function to determine the 20 closest words to that vector. You should see some words that make it clear what the classifier is doing. Do some of the words that you generated make sense? Explain. [4 points]

4.7 Saving and loading your model

Your trained model will be used later, in Section 6. You shoul save the parameters of your trained model with the lowest validation error using `torch.save(model.state_dict(), 'model_baseline.pt')`. See `https://pytorch.org/tutorials/beginner/saving_loading_models.html` for detail on saving and loading PyTorch networks.

5 Convolutional Neural Network (CNN) Classifier (15 points)

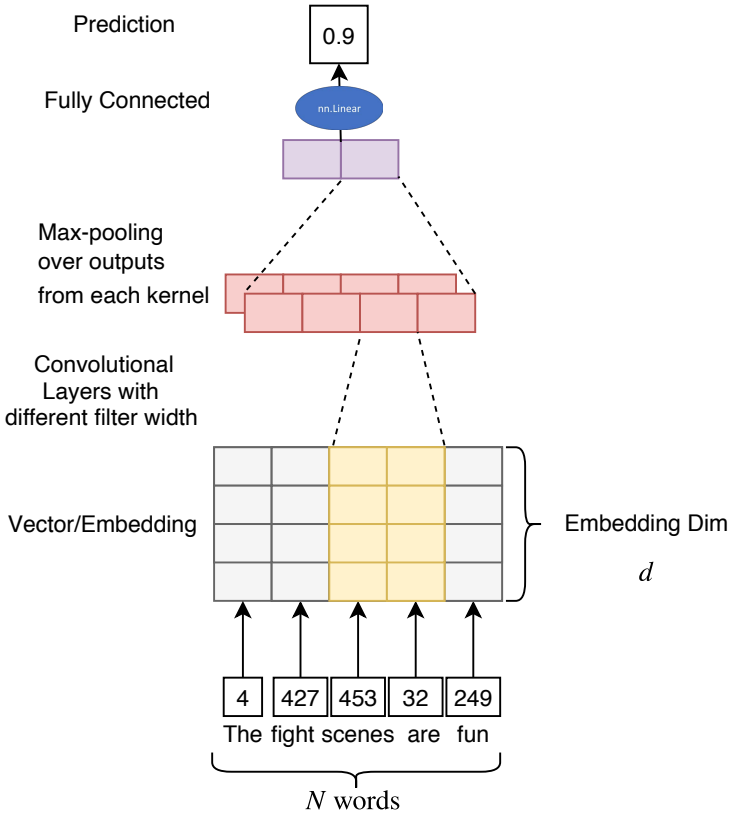

Figure 3: A convolutional neural network architecture

5.1 Submit Baseline Code

Submit your full code for this section in either in a notebook file named `A2_Baseline.ipynb` or in a zip file containing all your python files named `A2_Baseline.zip`. Your code should clearly state

how it should be run, and it should have easy-to-use arguments that allow any part of this Section to be run. [4 points]
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 from 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, but the batch loading function pads each sentence in a batch with zeroes to make them all the same length. 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_i`, embedding_dim]`. You should write your code to allow for the parameterization of k_1 and k_2 as well as the number of kernels n_1 and n_2. Note that this organization of convolutional layers is different from traditional computer vision-oriented CNNs in which one layer is fed into the next; these layers are operating on the same input. Note also, that, even though the kernel sizes span the entire embedding dimension, you can use the nn.Conv2d method, and explicitly specify the size of a kernel using the `kernel_size=(kx_size,ky_size)` parameter.

3. Set the convolutional kernel to **remove** the `bias` term. This is necessary for Section 5.3 below, in which you explore the words the kernels look like. This is done by setting the parameter `bias=False` when you instantiate the `Conv2d` layers in the network. This bias setting is `True` by default.

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

5. The network uses a `MaxPool` operation on the convolution layer output (after activation), along the sentence length dimension to reduce the amount of output produced from each kernel. That is, it computes the maximum across the entire sentence length, to obtain one output feature/number from each sentence for each kernel.

6. Concatenate the outputs from the maxpool operations above to form a vector of dimension $n_1 + n_2$ – be sure you correctly construct the shape of the resulting model.

7. Finally, similar to the baseline architecture, use a fully connected layer to produce a scalar output with Sigmoid activation to represent the probability that the sentence is in the subjective class. Note 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.

5.2 Overfit

Once you've finished coding the model, use the overfit dataset, and the parameters $k_1 = 2, n_1 = 50, k_2 = 4, n_2 = 50$ to make sure that you can overfit the model, as discussed in Section 4.4. Report the accuracy that you were able to achieve with the overfit dataset. [1 point]

5.3 Training and Parameter Exploration

Explore the parameter space of the CNN in the following steps, using the full dataset:

1. Here you should explore the normal hyper parameters for neural networks along with the specific ones in this CNN - k_1, n_1, k_2 and n_2. As a suggestion, start with $k_1 = 2, n_1 = 10, k_2 = 4, n_2 = 10$ and select the other hyperparameters. After that, explore different values of k_1, n_1, k_2, n_2 to achieve the best accuracy that you can. Report the accuracy and the full hyperparameter settings. Give the training and validation curves for that best model, and describe your overall hyperparmeter tuning approach. [4 points]

2. Re-run your best model, but allow the embeddings to be fine-tuned during the training, by setting the `freeze` parameter to `False` on the `nn.Embedding.from_pretrained` class. Report the accuracy of the result, and comment on the result. Save this model in a `.pt` file as you did in Section 4.7, for use below in Section 6. [2 points]

5.4 Extracting Meaning From Kernels

Observe that one of the kernel's dimension is the same size as the word embedding. Since the kernels are trained to learn values that should be present (or not present) in the input, they have an interpret-able meaning, as was the case in Section 4.6. You can explore that meaning using the function `print_closest_cosine_words` from Assignment 1. Use that function to determine the five closest words to each of the words in the the kernels trained in your best classifier. Do those words make sense? Do the set of words in each given kernel give a broader insight into what the model is looking for? Explain. [4 points]

5.5 Submit CNN Code

Submit your full code for this section in either in a notebook file named `A2_CNN.ipynb` or in a zip file containing all your python files named `A2_CNN.zip`. Your code should clearly state how it should be run, and it should have easy-to-use arguments that allow any part of this Section to be run. [4 points]

6 Web-based User Interface to Classify Input Sentences Using Gradio (5 points)

In this section, you will write code that interacts with your best model. You will use the **Gradio** software to build this user interface, as Gradio may come in handy during the course project. As shown in class, your Gradio interface to the model should provide a box for an input sentence, through a web page, and display the classification from each of the two models (your baseline and the best cnn), and well as the output "probability" that this sentence is subjective.

First, install the library and read about the basics of Gradio here: `https://gradio.app/getting_started/`. Then, write a Gradio-based python script/notebook that takes in a sentence, and outputs the result of your best Baseline and CNN networks (that you saved in a `.pt` file, as described in Sections 5.3 (part 5) and 4.7, and reload in a separate script/notebook. This output should be two things: the subjective/objective classification and the associated output probability for each of the two models.

Your code should process the input string in the following way, to create a tensor that can be input into the two models:

1. Load the `Vocab` object as in Section 3.3 using the `torchtext.vocab.GloVe` object.

2. Load the saved parameters for models you've trained:

```
checkpoint = torch.load('filename.pt')
model = torch.load_state_dict(checkpoint)
```

3. Using the code given below, split the input string into separate words, and convert the individual words into their tokens, and then into a tensor:

```
tokens = sentence.split()

# Convert to integer representation per token
token_ints = [glove.stoi.get(tok, len(glove.stoi)-1) for tok in tokens]

# Convert into a tensor of the shape accepted by the models
token_tensor = torch.LongTensor(token_ints).view(-1,1)
```

4. Pass this tensor through your models, and display, using gradio, the prediction results.

6.1 Run and Compare

Run your two best stored models on 4 sentences that you come up with yourself, where two of the sentences are definitely objective/subjective, and the other two are borderline subjective/objective, according to your opinion. **Include the input and output in your write up**. Comment on how the two models performed and whether they are behaving as you expected. Do they agree with each other? Which model seems to be performing the best? [1 point]

6.2 Submit Gradio Code

Submit your full code for this section in either in a notebook file named `A2_Gradio.ipynb` or in a zip file containing all your python files named `A2_Gradio.zip`. In addition, submit the two model files that you used for this section in files named `baseline.pt` and `cnn.pt`. Your code should clearly state how it should be run, and it should have easy-to-use arguments that allow any part of this Section to be run. [4 points]

References

[1] Internet movie database. `https://www.imdb.com/`. Accessed: 2022-09-25.

[2] Movie review data. `https://www.cs.cornell.edu/people/pabo/movie-review-data/`. Accessed: 2022-09-25.

[3] Rotten tomatoes. `https://www.rottentomatoes.com/`. Accessed: 2022-09-25.

[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.