Introduction to Deep Learning 2023
Homework 3:
Twitter Sentiment Classification using Recurrent Neural Networks and Word Embeddings

Due Date: 15.03.2023

Introduction

In the final homework assignment, you will again implement a sentiment classifier for tweets, but this time using a recurrent neural networks as your model, and pretrained word embeddings as your input.

1 Sentiment Analysis Using Recurrent Networks

This time, we will model the sentiment classification task using a recurrent architecture: We will be training a recurrent layer, which consume the words in the tweet as input, and output a state representation of this sequence. On top of this recurrent layer, we will implement a feed-forward component, which takes this state representation as input, and outputs the predicted sentimet. This is known as a many-to-one architecture, consuming a sequence of inputs, and giving out a single output eventually. Our new model is going to be composed of three sub-parts:

1. The Embedding Layer
The first layer is going to take the tokens[1] in our sequence, and encode them into their embeddings.

2. The Recurrent Layer
This layer is going to take the embeddings for each token in the sequence, and process

[1]Note that a token can be a word, subword, punctuation mark, etc, depending on your implementation.

them internally one by one, while updating its internal state according to each incoming character. (This internal processing will be handled by the relevant pytorch module.)

You can choose whether to use an LSTM, GRU, or vanilla RNN for the recurrent layer. You are encouraged to experiment with more than one type and see how they perform comparatively. You can find all these different layers readily implemented within the `pytorch.nn` module. Note that we will be using more "abstract" pytorch layers (eg. LSTM), and not the more "lower-level" layers called X-Cell (eg. LSTMCell). The difference between the two is as follows: The LSTMCell corresponds to one step of a long-short term memory computation, it takes in one input, updates its step once, and produces one output. On the other hand, the LSTM layer runs over an entire sequence from beginning to end, and gives us the final output and the final updated state eventually. Using the LSTM will enable us to let pytorch handle the entire time steps, and we will instead just feed it the input, and eventually collect the corresponding output.

3. The Classifier (Linear Layer + `log_softmax`)
On top of the recurrent layer, we are going to implement a fully-connected feed forward layer, which takes in the output of the recurrent layer as its input, and returns the predicted sentiment out of two possibilities, which are Positive vs. Negative. We will finally pass these outputs through a `log_softmax` function to receive the log probabilities. This part will be similar to the various classifiers we have been implementing so far.

Note that if there are N tokens in the sequence, the recurrent layer is going to output N output values, which is dependent on the sequence length. However, for the classifier, which is a FFNN, we need a fixed-length vector. Therefore you need to "summarize" the sentence representation as a fixed-width vector. Two possible options are:

1. Using the final output of the RNN for this purpose, assuming that this is the summarization of the sentence after seeing all the tokens in the sentence.
2. Using a pooling (eg. max or average pool) over all the N outputs of the RNN, obtaining a summary by for example averaging the outputs for all of the N tokens.

These two strategies are very standard, but you can also come up with your own strategy if you wish. You are encouraged to try different strategies and see how each of them fares.

Getting Started

First download and unpack the archive `intro_to_dl_assignment_3.zip`

```
$ unzip intro_to_dl_assignment_3.zip
```

This results in a directory `intro_to_dl_assignment_3` with the following structure

```
--data/
----sent140.train.mini.csv
----sent140.train.midi.csv
----sent140.dev.csv
----sent140.test.csv
--src/
----DL_hw3.py
----DL_hw3.ipynb
----data.py
```

Important! The file `sent140.train.csv` is too big for Moodle. Please download from this link instead.

The original data being composed of 1.6M tweets, it is divided into three sets, `sent140.dev.csv` and `sent140.test.csv` composed of 10,000 tweets each, and `sent140.train.csv` containing the remaining 1.58M tweets. `sent140.train.mini.csv` (10,000 tweeets) and `sent140.train.midi.csv` (100,000 tweets) are also provided for ease of development.

Twitter Data

We will be working with the twitter data collected and annotated by [1] and made available at `https://www.kaggle.com/kazanova/sentiment140`. It includes tweets with a binary annotation of positive or negative sentiment.

Each line in the csv files includes 6 fields:

- An integer label, indicating the sentiment of the tweet. 0: Negative, 1: Positive
- Tweet ID (not important for us)
- Timestamp of the tweet (not important for us)
- Query flag (not important for us)
- User name (not important for us)
- Tweet text

Some sample lines from `sent140.train.csv` are:

```
0,"2013116670","Tue Jun 02 21:57:24 PDT 2009","NO_QUERY","Seb_Bass_Tien",
"Too much things and worries are keeping me awake. Can't deal with all of
that at the same time. "
1,"2063653222","Sun Jun 07 03:32:25 PDT 2009","NO_QUERY","Luvelii","Last
week of school!!! Can't wait until Friday!!! "
0,"2227701106","Thu Jun 18 13:31:31 PDT 2009","NO_QUERY","katiegrrl85","I
```

`have been having the best conversation all day...I wish it didn't have to end "`

The Starter Code

Let's look at the main program in `DL_hw3.py`. It

1. reads in the Twitter sentiment data sets using the `torchtext` library,
2. loads the `glove.twitter.27B.200d` pretrained embeddings (27B means they have been trained on 27B tokens from twitter data, 200d means the embeddings have a vector size of 200 dimensions),
3. initializes a Recurrent Neural Network model, loss function and optimizer, and
4. trains the model for 5 epochs on the training data, checking the performance on the development data.

The data is prepared for processing using the `torchtext` library. Lets see how the data is prepared step by step:

```
# tokenizer function using spacy
tok = spacy.load('en_core_web_sm',disable=['parser', 'tagger', 'ner'])
def tokenizer(s):
    return [w.text.lower() for w in tok(tweet_clean(s))]

def tweet_clean(text):
    # remove non alphanumeric character
    text = re.sub(r'[^A-Za-z0-9]+', ' ', text)
    # remove links
    text = re.sub(r'https?:/\/\S+', ' ', text)
    return text.strip()
```

This part of the code prepares a tokenizer using the spacy library, as well as a cleaning function for the typically noisy tweet data. The tokenizer is going to be run on the tweet texts while reading in the data.

! **Note that you may need to download the related spacy module using:**

```
python -m spacy download en_core_web_sm
```

The `src/data.py` file presents:

1. A `get_data` function to read the data files

2. A `TwitterDataset` class that is responsible with preparing a single data item, tokenizing it and converting the tokens into related GloVe indices
3. A `collate_fn` that parepares the GloVe indices from different length tweets into a single tensor batch by padding them to equal length
4. A `get_dataset` that prepares the GloVe vocabulary for doing the token to index conversion (adding also an <unk> token to stand for all out-of-vocabulary tokens), and then uses all of this functionality to create the dataset loaders to be used in the main code.

`get_dataset` function eventually returns three dataset loaders, `train_loader, dev_loader, test_loader`, and the GloVe embeddings, modified to include the <unk> token, that you can use to initialize your Embedding layer.

Each batch is read from the laoders using the iterator functionality will have the fields `inputs, labels, lengths`. The last one is necessary only if you use the `pack_padded_sequence` functionality (See Section 6).

2 Assignment: Using Pretrained Embeddings

The starter code downloads and uses `glove.twitter.27B.200d` pretrained embeddings (27B meaning they have been trained on 27B tokens from twitter data, 200d meaning they have dimensionality of 200).

`glove_embeddings` is a matrix of shape [`vocabulary_size`+1 × `embedding_dim`]. You must use this matrix in your model to initialize your embeddings layer:

```
nn.Embedding.from_pretrained(glove_embeddings)
```

The `+1` in the matrix row count comes from the augmentation to this matrix in `data.py` with an `<unk>` token, short for Unknown, that is assigned as the `default index`, and therefore will be mapped to by any token that doesn't exist in your embedding list. Tokens such as emojis or proper names are examples of this.

3 Assignment: Recurrent Model

Your first task is to implement the recurrent model. As mentioned above, it will include one embedding layer, one recurrent component for feature extraction, and one feed-forward component for classification.

The embedding layer should receive `batch_size`-many sequences (in our scenario each sequence is a tweet). The shape of its input tensor will be [`sequence_len` × `batch_size`].

It should convert all the tokens in the sequences to their corresponding embeddings, and return a tensor of shape [`sequence_len` × `batch_size` × `embedding_dim`].

The recurrent component is going to take the output of embedding layer, and going to process all the `batch_size`-many sequences internally. The the rnn and gru layers in pytorch return two outputs:

```
out, hidden = self.rnn(input)
```

and the lstm layer returns an additional cell state:

```
out, (hidden, cell) = self.lstm(input)
```

`out` is an output that holds all the history of the states that the lstm passed through while processing the input sequence. Its shape will be [`sequence_len` × `batch_size` × `num_directions * hidden_size`] (where the `num_directions` can be 1 for unidirectional and 2 for bidirectional recurrent layer). Note that for each sequence in the batch, it will include one state of size `num_directions * hidden_dim` for each letter in the sequence (hence it will have `sequence_len`-many of these states). `hidden` is of shape [`num_layers`*`num_directions` × `batch_size` × `hidden_size`] and will keep the hidden state at the end of the input (at time t =`sequence_len`). `cell`, if exists, will keep the cell state also at time t =`sequence_len`.

The final linear layer is going to take a fixed-size vector summarizing the outputs of the recurrent layer, as mentioned above you can use for example the final output of the recurrent layer, or a pooling over all its outputs. This input should be reshaped into a [`batch_size` × `hidden_dim`] format, which is how batches are formatted for being used by feed-forward networks. Therefore, the final layer is going to accept an input of shape [`batch_size` × `hidden_dim`], and output the class scores in the shape [`batch_size` × `n_classes`]. You should then feed this output to a `log_softmax` function in the `forward()` function to obtain the class log probabilities.

Note: The official pytorch documentation uses a function `init_hidden()` to initialize a hidden state, and then feeds this initial hidden state to the recurrent layer (eg. lstm) as its second argument. This is useful when you want to learn initial states as well, which you can use to steer the network to a certain direction. For example, you may want the network to translate a sentence in one style (eg. formal) given a certain initial state, and in another style (eg. informal) given another initial state. The initial state can thus be used as a "cue" to the network. However, we don't start with such knowledge, so we will not use such a scheme. Therefore, you don't need to initialize the hidden state. Instead, we recommend using only the embeddings-layer-output as your single input, as shown above. When the second argument is omitted, the network creates a hidden state from scratch for every input, and equalizes it to the zero vector.

4 Assignment: Loss Function and Optimizer

As usual you need to implement a loss function and initialize the optimizer.

5 Assignment: Batch Training

Batching in recurrent networks is different from the batching we have been using in feed-forward networks. This is because now we have an additional dimension, that is the index over the elements of the sequence, to also think about.

The logic of batching in sequence-processing models is that we arrange the first element of the batch so that it holds the first elements of every sequence in the batch. The second element of the batch is going to hold the second element of every sequence in the batch, and so on. Therefore, the first dimension of the tensor will be of `sequence_len` size, the second dimension will be of `batch_size`-size, and the third dimension (which will keep the actual values) will be of `embedding_dim`-size.

There is one important practicality in the batching of sequential data. To be able to represent `batch_size`-many inputs in a single tensor, we have to make sure that all of them are of the same length. However, tweets in our data have varying lengths. Remember that this is not a problem if our recurrent layer is processing 1 sequence at a time (`batch_size` = 1). However, if we want to use a `batch_size` of greater than 1 (which we generally do), then we need to make sure all the sequences in the batch have the same length with each other, so that they can be tucked into the same tensor. We can achieve this by adding extra `<PAD>` tokens at the end of each sequence, until they are all the same length with the longest sequence in the batch. Note that *different* batches can have different sequence-lengths, they don't have to match with each other. The sequence-length of each batch is simply determined by the longest sequence it contains.

However, notice that there in an important inefficiency here: Since the length of the tweets in our batch are widely different, we have to pad each one to the longest word in it. Since our dataset can include very long tweets, we will likely need to pad almost all sequences to a great degree. This is both time-wise inefficient, and also introducing a bias into the data, which can be pretty severe in case we happen to have some long sequences.

A good trick to get out of this conundrum is to *sort* our dataset according to the sequence length after shuffling it at every epoch, and place sequences with similar lengths into the same batch. This is going to ensure that we will need a minimum amount of padding: Usually a single pad token will be enough for the shorter sequences in the batch.

6 Assignment: Packing Sequences

A very good solution the padding inefficiency problem is using `nn.utils.rnn.pack_padded_sequence()`. You are encouraged to use this functionality. What it does is to form concatenate together sequences of different lengths, keeping also the length of each sequence in mind, and pass this "packed" sequence to the recurrent module, so that the recurrent module will do only the necessary computations (and not the unnecessary ones required for the paddings). The pack function is called for packing the padded inputs when passing them to the recurrent module, and the returned value is then unpacked to continue with the processing, as in:

```
packed_embedded = nn.utils.rnn.pack_padded_sequence(embedded, text_lengths)
packed_output, (hidden, cell) = self.rnn(packed_embedded)
output, output_lengths = nn.utils.rnn.pad_packed_sequence(packed_output)
```

We hope you enjoy :)

Submission

Upload your `DL_hw3_USERNAME.py` or `DL_hw3_USERNAME.ipynb` file on Moodle.

References

[1] Go, A., Bhayani, R. and Huang, L., *Twitter sentiment classification using distant supervision*. CS224N Project Report, Stanford, 2009.