BME646 and ECE60146: Homework 9

Spring 2023
Due Date: 11:59pm, Apr 10, 2024
TA: Akshita Kamsali (akamsali@purdue.edu)

Turn in typed solutions via BrightSpace. Additional instructions can be found at BrightSpace. Late submissions will be accepted with penalty: **-10 points per-late-day, up to 5 days**.

1 Introduction

The initial step in any Natural Language Processing (NLP) task involves text preprocessing, particularly tokenization. Tokenization breaks down a stream of text into meaningful units called tokens, such as words or sentences. This process is essential as it transforms unstructured text data into a format suitable for analysis.

Tokenization is fundamental to NLP pipelines, as it divides text into discrete units, enabling their representation as vectors for machine learning. This conversion from raw text to numerical data facilitates further analysis and processing.

Following tokenization, the next step is to extract embeddings for these tokens. Word embeddings, for instance, capture the semantic meaning of words in a numerical form, facilitating various NLP tasks.

2 Getting Ready for This Homework

Before embarking on this homework, do the following:

1. Carefully review Slide 46 through 59 of the Week 12 slides on "Recurrent Neural Networks for Text Classification and Data Prediction" [1]. Make sure you understand how gating is done in GRU to address the problem of vanishing gradients that are caused by the long chains of feedback in a neural network.

2. Review the Week 13 slides on "Word Embeddings and Sequence-to-Sequence Learning" [2]. In particular, pay attention to Slide 39 through 49 on word2vec and fastText. Make yourself familiar with their use and advantages over one-hot vector encoding.

3. Download the text dataset provided on Brightspace. The provided dataset is "Financial Sentiment Analysis" dataset [4]. The dataset has three sentiments, namely, `["positive", "neutral", "negative"]`. This makes its a 3 class classification problem.

4. Install `transformers` library into your conda environment, as this will be used to extract subword tokens, subsequently, word embeddings.

3 Programming Tasks

3.1 Tokenization

Your first task in this HW would be to tokenize the data. The steps are:

1. Depending on the specific task at hand, text can be tokenized at various levels such as character, subword, word, or sentence level. For this assignment, we will focus on tokenization at the word and subword levels.

2. To tokenize text at the word level, each word is separated at whitespace boundaries. This can be achieved using the built-in `split()` function. The following code snippet illustrates how this process can be implemented:

```
import csv

# this is an example of how to read a csv file line by line
# This snipped only shows the processing on the first 4 entries
sentences = []
sentiments = []
count = 0
with open('data.csv', 'r') as f:
    reader = csv.reader(f)
    # ignore the first line
    next(reader)
    for row in reader:
        count += 1
        sentences.append(row[0])
        sentiments.append(row[1])
        if count == 4:
            break

print(sentences)
```

```
# ["The GeoSolutions technology will leverage Benefon 's GPS solutions by providing Location Based Search Technology , a Communities Platform , location relevant multimedia content and a new and powerful commercial model .",
# '\$ESI on lows, down \$1.50 to \$2.50 BK a real possibility',
# "For the last quarter of 2010 , Componenta 's net sales doubled to EUR131m from EUR76m for the same period a year earlier , while it moved to a zero pre-tax profit from a pre-tax loss of EUR7m .",
# 'According to the Finnish-Russian Chamber of Commerce , all the major construction companies of Finland are operating in Russia .']

print(sentiments)
# ['positive', 'negative', 'positive', 'neutral']

# tokenize the sentences word by word
word_tokenized_sentences = [sentence.split() for sentence in sentences]
print(word_tokenized_sentences[:2])
# [['The', 'GeoSolutions', 'technology', 'will', 'leverage', 'Benefon', "'s", 'GPS', 'solutions', 'by', 'providing', 'Location', 'Based', 'Search', 'Technology', ',', 'a', 'Communities', 'Platform', ',', 'location', 'relevant', 'multimedia', 'content', 'and', 'a', 'new', 'and', 'powerful', 'commercial', 'model', '.'], ['$ESI', 'on', 'lows,', 'down', '$1.50', 'to', '$2.50', 'BK', 'a', 'real', 'possibility']]
# pad the sentences to the same length
# here I chose the max of all the sentences. You may set it to a hard number such
```

```
                                    as 64, 128 etc.
max_len = max([len(sentence) for sentence in
                                    word_tokenized_sentences])
padded_sentences = [sentence + ['[PAD]'] * (max_len - len(
                                    sentence)) for sentence in
                                     word_tokenized_sentences]
print(padded_sentences[:2])
#[['The', 'GeoSolutions', 'technology', 'will', 'leverage
                                    ', 'Benefon', "'s", 'GPS',
                                     'solutions', 'by', '
                                    providing', 'Location', '
                                    Based', 'Search', '
                                    Technology', ',', 'a', '
                                    Communities', 'Platform',
                                    ',', 'location', 'relevant
                                    ', 'multimedia', 'content
                                    ', 'and', 'a', 'new', 'and
                                    ', 'powerful', 'commercial
                                    ', 'model', '.', '[PAD]',
                                    '[PAD]', '[PAD]', '[PAD]',
                                     '[PAD]', '[PAD]', '[PAD
                                    ]'], ['$ESI', 'on', 'lows
                                    ,', 'down', '$1.50', 'to',
                                     '$2.50', 'BK', 'a', 'real
                                    ', 'possibility', '[PAD]',
                                     '[PAD]', '[PAD]', '[PAD
                                    ]', '[PAD]', '[PAD]', '[
                                    PAD]', '[PAD]', '[PAD]',
                                    '[PAD]', '[PAD]', '[PAD]',
                                     '[PAD]', '[PAD]', '[PAD
                                    ]', '[PAD]', '[PAD]', '[
                                    PAD]', '[PAD]', '[PAD]',
                                    '[PAD]', '[PAD]', '[PAD]',
                                     '[PAD]', '[PAD]', '[PAD
                                    ]', '[PAD]', '[PAD]']]
```

3. Subword level tokenization, also known as wordpiece tokenization employed in models like BERT, offers the advantage of breaking down less frequent words into subwords that occur more frequently. Below is code snippet demonstrating how one can perform subword tokenization as used in BERT:

```
from transformers import DistilBertTokenizer
model_ckpt = "distilbert-base-uncased"
distilbert_tokenizer = DistilBertTokenizer.from_pretrained
                                    (model_ckpt)

# bert encode returns the tokens as ids.
```

```
# i have set the max length to what we have padded the sentences to in word tokens
# you are free to choose any size but be consistent so that you may use the same model for training.
bert_tokenized_sentences_ids = [distilbert_tokenizer.encode(sentence, padding='max_length',
                                truncation=True, max_length=max_len) for sentence in sentences]

print(bert_tokenized_sentences_ids[:2])
# [[101, 1996, 20248, 19454, 13700, 2015, 2974, 2097, 21155, 3841, 12879, 2239, 1005, 1055, 14658, 7300, 2011, 4346, 3295, 2241, 3945, 2974, 1010, 1037, 4279, 4132, 1010, 3295, 7882, 14959, 4180, 1998, 1037, 2047, 1998, 3928, 3293, 2944, 102], [101, 1002, 9686, 2072, 2006, 2659, 2015, 1010, 2091, 1002, 1015, 1012, 2753, 2000, 1002, 1016, 1012, 2753, 23923, 1037, 2613, 6061, 102, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]
bert_tokenized_sentences_tokens = [distilbert_tokenizer.convert_ids_to_tokens(sentence) for sentence in bert_tokenized_sentences_ids]
print(bert_tokenized_sentences_tokens[:2])
# [['[CLS]', 'the', 'geo', '##sol', '##ution', '##s', 'technology', 'will', 'leverage', 'ben', '##ef', '##on', "'", 's', 'gps', 'solutions', 'by', 'providing', 'location', 'based', 'search', 'technology', ',', 'a', 'communities', 'platform', ',', 'location', 'relevant', 'multimedia', 'content', 'and', 'a', 'new', 'and
```

```
                                    ', 'powerful', 'commercial
                                    ', 'model', '[SEP]'], ['[
                                    CLS]', '$', 'es', '##i', '
                                    on', 'low', '##s', ',', '
                                    down', '$', '1', '.', '50
                                    ', 'to', '$', '2', '.', '
                                    50', 'bk', 'a', 'real', '
                                    possibility', '[SEP]', '[
                                    PAD]', '[PAD]', '[PAD]',
                                    '[PAD]', '[PAD]', '[PAD]',
                                     '[PAD]', '[PAD]', '[PAD
                                    ]', '[PAD]', '[PAD]', '[
                                    PAD]', '[PAD]', '[PAD]',
                                    '[PAD]', '[PAD]']]
```

4. You will observe that tokens have an extra `[CLS]` and `[SEP]` token when used with the bert tokenizer. Some words are split into subwords. Example: `"solution"` is split into `"##sol"` and `"##ution"`.

3.2 Word Embeddings

Following tokenization, the next step involves generating word embeddings. Before proceding further, it's important to note that while the BERT tokenizer provides token IDs, our word tokenization process yields only the words themselves. Therefore, before extracting embeddings, we need to create token IDs for the word tokens. The code snippet shows one way of doing this:

```
vocab = {}
vocab['[PAD]'] = 0
for sentence in padded_sentences:
    for token in sentence:
        if token not in vocab:
            vocab[token] = len(vocab)

# print(vocab)

# convert the tokens to ids
padded_sentences_ids = [[vocab[token] for token in sentence]
                                    for sentence in
                                    padded_sentences]
print(padded_sentences_ids[:2])
# [[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17,
                                    18, 19, 16, 20, 21, 22, 23, 24,
                                     17, 25, 24, 26, 27, 28, 29, 0,
                                     0, 0, 0, 0, 0, 0], [30, 31, 32
```

```
                                , 33, 34, 35, 36, 37, 17, 38,
                                39, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
                                , 0, 0, 0, 0, 0, 0, 0, 0]]
```

Now, let us move onto extracting embeddings.

```python
from transformers import DistilBertModel
import torch

model_name = 'distilbert/distilbert-base-uncased'
distilbert_model = DistilBertModel.from_pretrained(model_name)

# extract word embeddings
# we will use the last hidden state of the model
# you can use the other hidden states if you want
# the last hidden state is the output of the model
# after passing the input through the model
word_embeddings = []
# convert padded sentence tokens into ids
for tokens in padded_sentences_ids:
    input_ids = torch.tensor(tokens).unsqueeze(0)
    with torch.no_grad():
        outputs = distilbert_model(input_ids)

    word_embeddings.append(outputs.last_hidden_state)

print(word_embeddings[0].shape)
# torch.Size([1, 39, 768])

# subword embeddings extraction
subword_embeddings = []
for tokens in bert_tokenized_sentences_ids:

    input_ids = torch.tensor(tokens).unsqueeze(0)
    with torch.no_grad():
        outputs = distilbert_model(input_ids)

    subword_embeddings.append(outputs.last_hidden_state)

print(subword_embeddings[0].shape)
# torch.Size([1, 39, 768])
```

3.3 Sentiment Analysis Using `torch.nn.GRU`

In this task, we ask you to carry out the same sentiment prediction task but with PyTorch's GRU implementation. The steps are:

1. First, familiarize yourself with the documentation of the `torch.nn.GRU` module [3]. Also, you should go through Slide 67 through 87 of the Week 12 slides to understand how you may feed in an entire sequence of embeddings at once when using `torch.nn.GRU`. Using the module in such manner may help you speed up training dramatically.

2. Before training, split your dataset into 80:20 ratio for train and test sets respectively. Create your custom dataloaders which returns word embeddings and the sentiment as a one-hot vector of 3 dimensions corresponding to each emotion.

3. Perform the sentiment prediction experiment using `torch.nn.GRU` similar to that presented in the lecture slides. Perform quantitative evaluation of the unidirectional RNN on the sequestered test set.

4. Now, repeat the above step with PyTorch's bidirectional GRU (*i.e.* with `bidirectional=True`). Note that you'll also need to adjust several other places in your RNN to accommodate the change of shape for the hidden state. Does using a bidirectional scan make a difference in terms of test performance?

5. In your report, report the overall accuracy and the confusion matrix produced by your RNN that is based on `torch.nn.GRU` as well as its bidirectional variant. Also, you should include plots of the training losses for the two RNNs. Write a paragraph comparing the test performances of the both RNN implementations that you have done.

4 Extra Credit (25 points)

Repeat the above sentiment analysis on the text classification dataset on the course website. The link to download the dataset is below:

https://engineering.purdue.edu/kak/distDLS/text_datasets_for_DLStudio.tar.gz

1. First of all, extract the dataset and analuyse the files inside them. You may refer to Slides 11 through 15 of the Week 12 slides to familiarize yourself with how the datasets are organized.

2. Refer to `Examples/text_classification_with_GRU_word2vec.py` on how to create dataloaders for this dataset. You may report results on

embedding sizes 200 and 400 in the dataset, (i.e. `sentiment_dataset_train_200.tar.gz` and `sentiment_dataset_train_400.tar.gz`, and corresponding test datasets.)

3. Train and test your Unidirectional and Bidirection RNNs you created in 3. You may refer to DLStudio code on how to create your training loop.

4. Repeat steps 3 and 5 from Section 3.

5 Submission Instructions

Include a typed report explaining how did you solve the given programming tasks. For HW9, you need to submit the following:

1. Your pdf must include a description of

   - The figures and descriptions as mentioned in Sec. 3, and 4 if you choose to do the extra credit.
   - Your source code. Make sure that your source code files are adequately commented and cleaned up.

2. Turn in a pdf file a typed self-contained report with source code and results. Rename your .pdf file as hw9_<First Name><Last Name>.pdf

3. Turn in a zipped file, it should include all source code files (only .py files are accepted). Rename your .zip file as hw9_<First Name><Last Name>.zip .

4. **Do NOT submit your network weights.**

5. **Do NOT submit your dataset.**

6. For all homeworks, you are encouraged to use `.ipynb` for development and the report. If you use `.ipynb`, please convert it to `.py` and submit that as source code.

7. You can resubmit a homework assignment as many times as you want up to the deadline. Each submission will overwrite any previous submission. **If you are submitting late, do it only once on BrightSpace.** Otherwise, we cannot guarantee that your latest submission will be pulled for grading and will not accept related regrade requests.

8. The sample solutions from previous years are for reference only. **Your code and final report must be your own work.**

9. To help better provide feedback to you, make sure to **number your figures and tables**.

References

[1] Recurrent Neural Networks for Text Classification and Data Prediction, . URL https://engineering.purdue.edu/DeepLearn/pdf-kak/RNN.pdf.

[2] Word Embeddings and Sequence-to-Sequence Learning, . URL https://engineering.purdue.edu/DeepLearn/pdf-kak/Seq2Seq.pdf.

[3] GRU. URL https://pytorch.org/docs/stable/generated/torch.nn.GRU.html.

[4] Pekka Malo, Ankur Sinha, Pyry Takala, Pekka Korhonen, and Jyrki Wallenius. Good debt or bad debt: Detecting semantic orientations in economic texts, 2013.