4B_Natural_Language_Processing_Assignment

May 12, 2021

1 PyTorch Assignment: Natural Language Processing (NLP)

Duke Community Standard: By typing your name below, you are certifying that you have adhered to the Duke Community Standard in completing this assignment.

Name:

1.0.1 Text Classification

In Notebook 4A, we built a sentiment analysis model for movie reviews. That particular sentiment analysis was a two-class classification problem, where the two classes were whether the review was positive or negative. Of course, natural language comes in all sorts of different forms; sometimes we want to perform other types of classification.

For example, the AG News dataset contains text from 127600 online news articles, from 4 different categories: World, Sports, Business, and Science/Technology. AG News is typically used for topic classification: given an unseen news article, we're interested in predicting the topic. For this assignment, you'll be training several models on the AG News dataset. Unlike the quick example we trained in Notebook 4A, however, we're going to *learn* the word embeddings. Since you may be unfamiliar with AG News, we're going to walk through how to load the data, to get you started.

1.0.2 Loading AG News with Torchtext

The AG News dataset is one of many included Torchtext. It can be found grouped together with many of the other text classification datasets. While we can download the source text online, Torchtext makes it retrievable with a quick API call*. If you are running this notebook on your machine, you can uncomment and run this block:

[ ]:
```
# import torchtext

# agnews_train, agnews_test = torchtext.datasets.text_classification.
↪DATASETS["AG_NEWS"](root="./datasets")
```

*At the time this notebook was created, Torchtext contains a small bug in its csv reader. You may need to change one line in the source code, as suggested here to successfully load the AG News dataset.

Unfortunately, Torchtext assumes we have network connectivity. If we don't have network access, such as notebooks running in Coursera Labs, we need to reimplement some Torchtext functionality. Skip this next block if you were able to successfully run the previous code:

[ ]:
```
import torchtext

ngrams = 1
train_csv_path = './datasets/ag_news_csv/train.csv'
test_csv_path = './datasets/ag_news_csv/test.csv'
vocab = torchtext.vocab.build_vocab_from_iterator(
    torchtext.datasets.text_classification._csv_iterator(train_csv_path,␣
 ↪ngrams))
train_data, train_labels = torchtext.datasets.text_classification.
 ↪_create_data_from_iterator(
        vocab, torchtext.datasets.text_classification.
 ↪_csv_iterator(train_csv_path, ngrams, yield_cls=True), False)
test_data, test_labels = torchtext.datasets.text_classification.
 ↪_create_data_from_iterator(
        vocab, torchtext.datasets.text_classification.
 ↪_csv_iterator(test_csv_path, ngrams, yield_cls=True), False)
if len(train_labels ^ test_labels) > 0:
    raise ValueError("Training and test labels don't match")
agnews_train = torchtext.datasets.TextClassificationDataset(vocab, train_data,␣
 ↪train_labels)
agnews_test = torchtext.datasets.TextClassificationDataset(vocab, test_data,␣
 ↪test_labels)
```

Let's inspect the first data example to see how the data is formatted:

[ ]:
```
print(agnews_train[0])
```

We can see that Torchtext has each example as a tuple, with the first element being the label (0, 1, 2, or 3), and the second element the text data. Notice that the text is already "tokenized": the words of the news article have been represented as word IDs, with each number corresponding to a unique word.

In previous notebooks, we've used `DataLoaders` to handle shuffling and batching. However, if we directly try to feed these dataset objects into a `DataLoader`, we will face an error when we try to draw our first batch. Can you figure out why? Here's a hint:

[ ]:
```
print("Length of the first text example: {}".format(len(agnews_train[0][1])))
print("Length of the second text example: {}".format(len(agnews_train[1][1])))
```

Because each example is a news snippet, they can vary in length. This is natural, as humans don't stick to consistent sentence length while writing. This creates a bit of a problem while batching, as default tensors expect the size of each dimension to be consistent.

How do we fix this? The common solution is to perform padding and/or truncation, picking a maximum sequence length L. Inputs longer than the maximum length are truncated (i.e. $x_{t>L}$ are

discarded), and shorter sequences have zeros padded to the end until they are all of length of L. We'll focus on padding here, for simplicity.

We can perform this padding manually, but Pytorch has this functionality implemented. As an example, let's pad the first two sequences to the same length:

[ ]:
```
from torch.nn.utils.rnn import pad_sequence

padded_exs = pad_sequence([agnews_train[0][1], agnews_train[1][1]])
print("First sequence padded: {}".format(padded_exs[:,0]))
print("First sequence length: {}".format(len(padded_exs[:,0])))
print("Second sequence padded: {}".format(padded_exs[:,1]))
print("Second sequence length: {}".format(len(padded_exs[:,1])))
```

Although originally of unequal lengths, both sequences are now the same length, with the shorter one padded with zeros.

We'd like the `DataLoader` to perform this padding operation as part of its batching process, as this will allow us to effectively combine varying-length sequences in the same input tensor. Fortunately, `Dataloaders` let us override the default batching behavior with the `collate_fn` argument.

[ ]:
```
import numpy as np
import torch

def collator(batch):
    labels = torch.tensor([example[0] for example in batch])
    sentences = [example[1] for example in batch]
    data = pad_sequence(sentences)

    return [data, labels]
```

Now that we have our collator padding our sequences, we can create our `DataLoaders`. One last thing we need to do is choose a batch size for our `DataLoader`. This may be something you have to play around with. Too big and you may exceed your system's memory; too small and training may take longer (especially on CPU). Batch size also tends to influence training dynamics and model generalization. Fiddle around and see what works best.

[ ]:
```
BATCH_SIZE = 128

train_loader = torch.utils.data.DataLoader(agnews_train, batch_size=BATCH_SIZE,␣
↪shuffle=True, collate_fn=collator)
test_loader = torch.utils.data.DataLoader(agnews_test, batch_size=BATCH_SIZE,␣
↪shuffle=False, collate_fn=collator)
```

1.0.3 Simple Word Embedding Model

First, let's try out the Simple Word Embedding Model (SWEM) that we built in Notebook 4A on the AG News dataset. Unlike before though, instead of loading pre-trained embeddings, let's learn the

embeddings from scratch. Before we begin, it will be helpful to define a few more hyperparameters.

[ ]:
```
VOCAB_SIZE = len(agnews_train.get_vocab())
EMBED_DIM = 100
HIDDEN_DIM = 64
NUM_OUTPUTS = len(agnews_train.get_labels())
NUM_EPOCHS = 3
```

Once again, we're going to organize our model as a `nn.Module`. Instead of assuming the input is already an embedding, we're going to make learning the embedding as part of our model. We do this by using `nn.Embedding` to perform an embedding look-up at the beginning of our forward pass. Once we've done the look up, we'll have a minibatch of embedded sequences of dimension $L \times$ `BATCH_SIZE` $\times$ `EMBED_DIM`. For SWEM, remember, we take the mean* across the length dimension to get an average embedding for the sequence.

*Note: Technically we should only take the mean across the embeddings at the positions corresponding to "real" words in our input, and not for the zero paddings we artificially added. This can be done by generating a binary mask while doing the padding to track the "real" words in the input. Ultimately though, this refinement doesn't have much impact on the results for this particular task, so we omit it for simplicity.

[ ]:
```
import torch.nn as nn
import torch.nn.functional as F

class SWEM(nn.Module):
    def __init__(self, vocab_size, embedding_size, hidden_dim, num_outputs):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_size)

        self.fc1 = nn.Linear(embedding_size, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, num_outputs)

    def forward(self, x):
        embed = self.embedding(x)
        embed_mean = torch.mean(embed, dim=0)

        h = self.fc1(embed_mean)
        h = F.relu(h)
        h = self.fc2(h)
        return h
```

With the model defined, we can instantiate, train, and evaluate. Try doing so below! Because of the way we organized our model as an `nn.Module` and our data pipeline with a `DataLoader`, you should be able to use much of the same code as we have in other examples.

Note: Depending on you system, training may take up to a few hours, depending on how many training epochs you set. To see results sooner, you can train for less iterations, but perhaps at the cost of final accuracy. On the other hand, using a GPU (and GPU-enabled PyTorch) should enable full training in a couple minutes.

[ ]:
```
### YOUR CODE HERE ###
```

1.0.4 RNNs

SWEM takes a mean over the time dimension, which means we're losing any information about the order of the data sequence. How detrimental is this for document topic classification? Modify the SWEM model to use an RNN instead. Once you get an RNN working, try a GRU and LSTM as well.

[ ]:
```
### YOUR CODE HERE ###
```

1.0.5 Short Answer:

1. How do the RNN, GRU, and LSTM compare to SWEM for AG News topic classification? Are you surprised? What about classification might make SWEM so effective for topic classification?

`[Your answer here]`

2. How many learnable parameters do each of the models you've trained have?

`[Your answer here]`