HW_2

February 7, 2023

1 Homework 2: PyTorch and Self-Attention

In this homework, you will begin exploring PyTorch, a neural network library that will be used throughout the remainder of the semester.

The PDF file for instructions can be found here.

You can toggle the outline on the left hand side to jump around sections more easily.

Due date: Tuesday February 7 at 11:59 PM

1.1 Setup

```
/usr/local/opt/python@3.9/bin/python3.9: Error while finding module
specification for 'nltk.downloader' (ModuleNotFoundError: No module named
'nltk')
```

When looking up pytorch documentation, it may be useful to know which version of torch you are running.

```
1.10.0
```

1.2 IMPORTANT: GPU is not enabled by default

You must switch runtime environments if your output of the next block of code has an error saying "ValueError: Expected a cuda device, but got: cpu"

Go to Runtime > Change runtime type > Hardware accelerator > GPU

```
Running on cpu
```

1.3 Deliverable 1: PyTorch and FFNN

1.3.1 Data Processing

Let's begin by loading our datasets and the 50-dimensional GLoVE word embeddings.

```
--2023-02-07 14:28:59--
https://raw.githubusercontent.com/dbamman/nlp23/main/HW2/train.txt
Resolving raw.githubusercontent.com (raw.githubusercontent.com)…
2606:50c0:8000::154, 2606:50c0:8002::154, 2606:50c0:8003::154, …
Connecting to raw.githubusercontent.com
(raw.githubusercontent.com)|2606:50c0:8000::154|:443… connected.
```

```
HTTP request sent, awaiting response… 200 OK
Length: 6645801 (6.3M) [text/plain]
Saving to: ‘train.txt.1’

train.txt.1         100%[===================>]   6.34M  5.84MB/s    in 1.1s

2023-02-07 14:29:01 (5.84 MB/s) - ‘train.txt.1’ saved [6645801/6645801]

--2023-02-07 14:29:02--
https://raw.githubusercontent.com/dbamman/nlp23/main/HW2/dev.txt
Resolving raw.githubusercontent.com (raw.githubusercontent.com)…
2606:50c0:8000::154, 2606:50c0:8002::154, 2606:50c0:8003::154, …
Connecting to raw.githubusercontent.com
(raw.githubusercontent.com)|2606:50c0:8000::154|:443… connected.
HTTP request sent, awaiting response… 200 OK
Length: 1309909 (1.2M) [text/plain]
Saving to: ‘dev.txt.1’

dev.txt.1           100%[===================>]   1.25M  --.-KB/s    in 0.07s

2023-02-07 14:29:03 (17.6 MB/s) - ‘dev.txt.1’ saved [1309909/1309909]

--2023-02-07 14:29:03--
https://raw.githubusercontent.com/dbamman/nlp23/main/HW2/glove.6B.50d.50K.txt
Resolving raw.githubusercontent.com (raw.githubusercontent.com)…
2606:50c0:8000::154, 2606:50c0:8002::154, 2606:50c0:8003::154, …
Connecting to raw.githubusercontent.com
(raw.githubusercontent.com)|2606:50c0:8000::154|:443… connected.
HTTP request sent, awaiting response… 200 OK
Length: 21357789 (20M) [text/plain]
Saving to: ‘glove.6B.50d.50K.txt.1’

glove.6B.50d.50K.tx 100%[===================>]  20.37M  9.31MB/s    in 2.2s

2023-02-07 14:29:07 (9.31 MB/s) - ‘glove.6B.50d.50K.txt.1’ saved
[21357789/21357789]
```

1.3.2 Demo: Logistic regression

First, let’s code up Logistic Regression in PyTorch so you can see how the general framework works.

Average Embedding Representation Let’s train a logistic regression classifier where the input is the average GloVe embedding for all words in a review.

```
/Library/Frameworks/Python.framework/Versions/3.7/lib/python3.7/site-
packages/ipykernel_launcher.py:41: UserWarning: Creating a tensor from a list of
numpy.ndarrays is extremely slow. Please consider converting the list to a
```

```
single numpy.ndarray with numpy.array() before converting to a tensor.
(Triggered internally at  ../torch/csrc/utils/tensor_new.cpp:201.)
```

1.3.3 Question 1: PyTorch and embeddings (writeup)

Here, you can see that a PyTorch implementation of logistic regression is offered for your reference. Study this script carefully: start with how we load and access the GloVe embeddings, and the most critical ingredients of a neural net in PyTorch include a class that defines the architecture (pay attention to the `forward` method), an optimizer (`torch.optim.Adam`), and a loss function, `torch.nn.CrossEntropyLoss()`, which combines the softmax function `torch.nn.LogSoftmax()` and negative log-likelihood `torch.nn.NLLLoss()` (see documentation).

A notable difference from HW1 here is that the input of this model is the average of GloVe embeddings for all words in a movie review. What is the difference between averaged embeddings and BoW? What are the advantages or disadvantages of each? Discuss in no more than 100 words.

Average embeddings represents a text by taking the word embedding for each word in the text, and averaging them to represent the text in a compact format, while BoW creates a vector where each word has a value that represents the frequency or presence of a word. Average embeddings allows one to capture relationships between words, while BoW is computationally efficient. Average embeddings are vulnerable to outlier words, and BoW is unable to capture semantic relations between words.

A complete training loop is here for your reference. There's nothing for you to implement here but you might want to make sure you understand the standard procedure. Remember that you **do not** have to finish training; none of your answers will depend on the model accuracy.

```
Epoch 0, dev accuracy: 0.648
Epoch 5, dev accuracy: 0.683
Epoch 10, dev accuracy: 0.702
Epoch 15, dev accuracy: 0.722
Epoch 20, dev accuracy: 0.729
Epoch 25, dev accuracy: 0.733
Epoch 30, dev accuracy: 0.737
Epoch 35, dev accuracy: 0.737
Epoch 40, dev accuracy: 0.739
Stopping training; no improvement on dev data after 10 epochs
```

1.3.4 Question 2: FFNN (TODO)

For this question, we want to add a hidden layer to the logistic regression classifier above. Implement Eqn. (7.13) in J&M SLP3 and let the non-linearity g be tanh. Your implementation should be similar to the `LogisticRegressionClassifier` above. Let's pick 20 for the size of the hidden layer – this is provided in the `__init__()` function below (`hidden_dim=20`), so you don't need to worry about it.

Note that in the J&M terminology, a "two-layer" network has one hidden layer, which is what you will be implementing. You should fill in the parts between "`#BEGIN SOLUTION`" and "`#END SOLUTION`"

```
Epoch 0, dev accuracy: 0.651
Epoch 5, dev accuracy: 0.744
Epoch 10, dev accuracy: 0.749
Epoch 15, dev accuracy: 0.747
Epoch 20, dev accuracy: 0.743
Epoch 25, dev accuracy: 0.745
Epoch 30, dev accuracy: 0.744
Epoch 35, dev accuracy: 0.741
Epoch 40, dev accuracy: 0.740
Stopping training; no improvement on dev data after 30 epochs
```

1.4 Deliverable 2: Attention

The self-attention mechanism is often thought of as one of the most transformative ideas in modern NLP. Its full form in Transformer, as introduced in "Attention is All You Need" (NIPS 2017) is rather involved. This deliverable aims to prepare you for it.

We will start with the simplest form of self-attention: scaled dot-product self-attention. The goal is to try to understand the roles that query, key, and value vectors play in attending to the input sequence: conceptually, what do they aim to achieve and improve on? How do you code this in Python?

1.4.1 Question 3: the concept of self-attention (writeup only)

Self-attention may be one of those less intuitive concepts you have encountered in this course.

In the era of ChatGPT, you might be curious if it can *tutor* you and answer your questions about a difficult concept in NLP.

This question is your chance to play around with it and start thinking about the power and limitations of such tools. First, make sure that you have worked through $§$10.1 in SLP 3 and the relevant lecture on self-attention. Then, you can go to https://chat.openai.com/chat, ask ChatGPT questions, and most importantly, evaluate its answers. Here's an example prompt:

> Why do we need the self-attention mechanism in NLP?

It yields the following answer:

> Self-attention is a mechanism in deep learning models for processing sequences, where each element in a sequence attends to all other elements in a weight-based manner to compute its representation. This enables the model to dynamically weigh the importance of different elements in the sequence for the current task, and allows for more context-aware representations compared to traditional recurrent neural networks.

Is this correct? Substantiate your reasoning in less than 100 words. The answer is not entirely accurate. self-attention is used to highlight the relevancy of the words, and not just a broad representation. In addition, self-attention is a technique applied in a neural network, and so it would not make much sense to compare a technique to recurrent neural networks.

1.4.2 Intition of attention

First, let's go through the basics of implementing the steps in attention outside of any model. We'll do that in Numpy. Using the example from lecture, let's assume we have three sets of parameters W^Q, W^K and W^V.

```
(2, 37) (2, 37) (2, 2)
```

Let's also assume we have an input sentence that's 5 tokens long; each token is represented as an embedding of length `input_embedding_size` (here, 2). That 5-word sentence, then, is represented as as 5×2 matrix `sent`. If we multiply `sent` by W^Q, the result is a 5×37 query matrix.

```
(5, 2) (5, 37)
```

Now let's also show how to perform the softmax operation on a matrix. Remember that the softmax function normalizes over a set of values $x = [x_1, ..., x_n]$ such that each $0 \leq x_i \leq 1$ and the sum of x = 1. Here, we have a 15×5 matrix m; if we perform the softmax over the columns of m (`axis=1`), each row will sum to 1.

```
**test_mamtrix**:
 [[0.75774054 0.91338009 0.99072117 0.26826807 0.31164955]
 [0.33288311 0.42120983 0.05332367 0.16573012 0.51192462]
 [0.02335728 0.28358279 0.87086216 0.22090578 0.41425459]
 [0.86413902 0.76811262 0.90006266 0.92857764 0.28078358]
 [0.93520885 0.51576825 0.26307401 0.87664501 0.60407663]
 [0.83903823 0.69834238 0.20700324 0.64181215 0.02235181]
 [0.53858011 0.97359089 0.48307387 0.78948512 0.42429879]
 [0.66500141 0.49681891 0.72247324 0.8052079  0.80317731]
 [0.23855963 0.0719058  0.19500952 0.02513395 0.33610596]
 [0.02146212 0.65027808 0.66900392 0.19329391 0.86623538]
 [0.12226466 0.90744404 0.97562323 0.00403214 0.61080923]
 [0.66488425 0.97121517 0.5878534  0.51773558 0.6343146 ]
 [0.61459832 0.73519934 0.85536737 0.00159646 0.97568087]
 [0.22364085 0.8430418  0.64856203 0.13774442 0.51443435]
 [0.62133009 0.18523791 0.61370943 0.00969719 0.280754  ]]
----------------------------------------------------------
**output**:
 [[0.21350016 0.24945465 0.26951343 0.13086485 0.1366669 ]
 [0.20447379 0.22335592 0.15460607 0.17299917 0.24456505]
 [0.13646739 0.17702858 0.31848984 0.16627353 0.20174066]
 [0.21883755 0.19880079 0.22684191 0.2334034  0.12211635]
 [0.26114706 0.17168186 0.13334621 0.2462925  0.18753237]
 [0.27298827 0.23715952 0.14509574 0.22412473 0.12063174]
 [0.17642345 0.27257065 0.16689766 0.2267373  0.15737095]
 [0.19219519 0.16244329 0.20356458 0.22112275 0.22067419]
 [0.21212723 0.17956413 0.20308734 0.17135897 0.23386233]
 [0.12046517 0.22591902 0.2301894  0.14304975 0.28037666]
 [0.12408873 0.27210179 0.29130051 0.11025152 0.20225745]
 [0.19541358 0.26545602 0.18092587 0.16867429 0.18953024]
 [0.1859771  0.20981466 0.23660512 0.10074793 0.26685518]
 [0.15054658 0.27968767 0.23025646 0.13815498 0.2013543 ]
 [0.25688509 0.16609114 0.25493489 0.139351   0.18273788]]
---------------------------------------------------------
**sum of each row**:
[1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
```

1.4.3 Question 4: Scaled dot product attention in Numpy (TODO)

(TODO). From all of this, you have the building blocks for implementing attention (outside of any model). Do so here by filling out the attention function below. Recall from lecture that attention given a query vector Q, key vector K and value vector V is given by the following equation:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V$$

You will calculate Q, K, V within the body of this function. The sole required argument to this function should be a 2D matrix $\in \mathbb{R}^{n \,\times\, \text{input\_embedding\_size}}$ for any arbitrary n (that is, corresponding to a sentence of arbitrary length). It should return a matrix of that same exact size that is the output of that attention process over the input, given the parameters specified below. d_k here is the size of the key vector (`query_key_size=37`).

```
(2, 37) (2, 37) (2, 2)
```

1.5 Question 5: Scaled dot product attention in PyTorch (TODO)

Data prep time again:

Now it's time to implement that as part of a model. Here we're going to embed attention within a larger model. For an input document of, say, 20 words, each represented by a 100-dimesional embedding, the input to attention is a 20×100 matrix; the output from attention is also a 20×100 matrix. In this larger model, we're going to average those output embeddings to generate a final document that's a single 100-dimensional vector; pass through a fully-connected dense layer to make a prediction.

Test yourself before proceeding (not graded, but always remind yourself of this kind of things): In our GloVe representation of movie review data, each padded review has ? tokens, represented by a ?-dimensional GloVe embedding?

Again, you can execute the code to self-check, but you don't need to finish training.

```
Epoch 0, dev accuracy: 0.536
Epoch 1, dev accuracy: 0.534
Epoch 2, dev accuracy: 0.536
Epoch 3, dev accuracy: 0.539
Epoch 4, dev accuracy: 0.538
Epoch 5, dev accuracy: 0.540
Epoch 6, dev accuracy: 0.540
Epoch 7, dev accuracy: 0.538
Epoch 8, dev accuracy: 0.538
Epoch 9, dev accuracy: 0.539
Epoch 10, dev accuracy: 0.539
Epoch 11, dev accuracy: 0.538
Epoch 12, dev accuracy: 0.539
Epoch 13, dev accuracy: 0.539
Epoch 14, dev accuracy: 0.539
Stopping training; no improvement on dev data after 10 epochs

Best Performing Model achieves dev accuracy of : 0.540
```

Congrats! You're officially done with this homework – be sure to check the "How to submit" section in the PDF.