AIML - CS 335

Lab 7: Curated Assortment of Bugs

22 September, 2022

Lecturer: Abir De

Important: Please read the instructions mentioned in the questions carefully. We have provided boilerplate code for each question. Please ensure that you make changes in the areas marked with TODO.

Please **read the comments in the code carefully** for detailed information regarding input and output format.

Reminder : In previous submissions, many students had imported some new libraries causing auto-grading to fail. These new libraries were perhaps auto-added by vscode. We request you to delete all such lines and make sure that you only add code in between `TODO`.

1 Ranking Loss Bug

Recall the midsem question on ranking loss. The description of the setup is repeated below for recap. We are provided a dataset, consisting of a set of queries Q (e.g., search queries you type in Google search) and a set of corpus items C (set of webpages). For every pair $(q, c) \in Q \times C$, we have binary ground truth relevance labels ($y(q, c) = +1$ (or -1) if c is relevant to q (or not)).

Let us further assume that there exists a model $\mathcal{M} : Q \times C \to \mathbb{R}$, which provides a similarity score $s \in \mathbb{R}$ for any query-corpus pair (q, c) as follows:

$$\mathcal{M}(q, c) = s$$

We now compute a *ranking loss* based on the current model predictions.

`naive_pairwise_ranking_loss` implements the non-tensorized version, and `tensorized_pairwise_ranking_loss` implements the tensorized version. However, we see that the assertion fails when checking equality of the outputs of the two implementations. **Your task is to find the bug in the tensorized implementation and fix it.**

Note: We would not have detected this bug, unless we had perform the assertion checks. Machine learning code can have multiple such bugs, which are silent and remain undetected. However, these bugs will mess up your training and results. Beware!!

2 Batched Implementation Bug

2.1 Setup: Set embeddings

Consider a dataset $\{S_1, S_2, \ldots, S_n\}$, where each S_i is a set of items. Each item in any of the sets, is represented as some D dimensional embedding, *i.e.,* $\in \mathbb{R}^D$ For example, each set S_i may represent a document, consisting of a set of words. In turn, each word is represented using a D dimensional embedding.
Note that each set may have a variable number of items. For examples, each of the documents has different word counts.
We have implemented a class `SetEmbed`, which generates an embedding representation for any set.
Consider a set $S_i = [\boldsymbol{x}_1, \boldsymbol{x}_2.., \boldsymbol{x}_N]^T$. This set consists of N items, each of which is an embedding vector $\boldsymbol{x}_i \in D$. The neural network generates the set embedding as follows:

$$emb = \text{Linear}\left(\sum_{i \in N} \text{Linear}(\text{ReLU}(\text{Linear}([\boldsymbol{x}_1, \boldsymbol{x}_2.., \boldsymbol{x}_N]^T)))\right) \tag{1}$$

Here, the inner LRL (Linear+Relu+Linear) layers, first transform the input matrix for $S_i \in \mathbb{R}^{N \times D}$ to some intermediate representatoin $\in \mathbb{R}^{N \times D_1}$. Subsequently, the summation aggregates the N item representations to obtain a single vector $\in \mathbb{R}^{D_1}$, which is subsequently passed through a Linear layer to obtain the final set embedding $emb \in \mathbb{R}^{D_2}$. The exact values of D_1 and D_2 are hardcoded.

2.2 Batched implementation

Our implementation in `SetEmbed` allows for batched processing for a bunch of sets. The input is now of shape (Batch Size, Max Set Size, D). Max Set Size is the maximum cardinality of the sets in the training dataset. For sets which have fewer elements that Max Set Size, the remaining rows (indicating non-existent items) are filled with zeros. For example:

$$[\boldsymbol{x}_1, \boldsymbol{x}_2.., \boldsymbol{x}_l, \vec{0}, \vec{0}, \ldots]^T \tag{2}$$

is the representation for a set which has l elements, where $l <$ Max Set Size.
`SetEmbed` outputs one embedding per input set. Therefore, for input of shape (Batch Size, Max Set Size, D), the output shape is (Batch Size, D_2). However there is a bug in the batched implementation. We see that if we generate embeddings for one set at a time without any $\vec{0}$ padding, then it does not match the output from the batched processing. This is clearly incorrect since the same weights and biases are used in both cases. However, the existence of $\vec{0}$ vectors seems to throw off the embedding generation procedure. **Your task is to fix this bug by adding relevant lines of codes in the marked section.** Your modification will allow for processing of variable

sized set inputs using batched processing.
Note: This scenario commonly occurs in practice, while designing models for variable sized inputs such as graphs and sets. One cannot write separate models for all possible input sizes. Hence, a common strategy is to pad the inputs to a common size (the maximum possible), and then use a shared model. However, one needs to be careful while using existing neural architectures for padded inputs.

2.3 Note

You can change below settings in `main` to try different variations.

```
input_dim = 12
set_sizes = [5,7,13]
```

The above sets embedding dimension of items D to 12. The `set_sizes` specifies the number of input sets (3 here) and the size per set. As you can see, the Max Set Size here is 13. Thus the first set will be padded with (13-5=) 8 rows of $\vec{0}$ and the second set will be padded with (13-7=) 6 rows of $\vec{0}$. The last set, being the largest will not have any $\vec{0}$ padding.

3 Simple Regression

We have given code for a simple 1D regression dataset that uses torch `autograd` functionality to learn the model. The code needs atleast 3 changes to make the model converge. The task is to fix the code so that the model trains properly.
Note: The given code also plots the model predictions every 5 epochs. This code should be commented out by writing `if False` when you submit the lab. **If autograder fails, we will assign 0**.

To fix the code, you will need to understand the `pytorch` model training steps very carefully. Here is an excellent tutorial on the same: https://pytorch.org/tutorials/beginner/pytorch_with_examples.html

4 Subset Selection

We have a data matrix X of size $\mathbb{R}^{N\times D}$ where N is the number of samples and D is the dimension of each sample. The task is to find the inner product of every sample with every other sample. We can observe that this inner product can be found by a simple matrix multiplication $P = XX^T$, where the $P[i, j]$ entry is the dot product of i^{th} and j^{th} sample. Making the task more interesting, suppose only few samples are relevant to us, given by a subset of indices. That is, let $S \subset [N]$, then the required inner product can be found by choosing the elements from the matrix X using S and storing it in matrix X_S. Then we can simply find $P_S = X_S X_S^T$. However, if we iterate over many such subsets, then we can observe that we perform many redundant computations which is

inefficient. You are required to develop a more efficient way to calculate the matrix P_S, given we are required to calculate P_S for many subsets S.

5 Assessment

We will be checking the following:

- Given Assertion checks are passing.

6 Submission instructions

Complete the functions in `assignment.py`. Make changes only in the places mentioned in comments. Do not modify the function signatures. Keep the file in a folder named `<ROLL_NUMBER>_L7` and compress it to a tar file named `<ROLL_NUMBER>_L7.tar.gz` using the command

```
tar -zcvf <ROLL_NUMBER>_L7.tar.gz <ROLL_NUMBER>_L7
```

Submit the tar file on Moodle. The directory structure should be -

```
<ROLL_NUMBER>_L7
| - - - - assignment.py
```

Replace `ROLL_NUMBER` with your own roll number. If your Roll number has alphabets, they should be in *"small"* letters.