C3_W4_Assignment

October 31, 2020

1 Assignment 4: Question duplicates

Welcome to the fourth assignment of course 3. In this assignment you will explore Siamese networks applied to natural language processing. You will further explore the fundamentals of Trax and you will be able to implement a more complicated structure using it. By completing this assignment, you will learn how to implement models with different architectures.

1.1 Outline

Overview In this assignment, concretely you will:

- Learn about Siamese networks
- Understand how the triplet loss works
- Understand how to evaluate accuracy
- Use cosine similarity between the model's outputted vectors
- Use the data generator to get batches of questions
- Predict using your own model

By now, you are familiar with trax and know how to make use of classes to define your model. We will start this homework by asking you to preprocess the data the same way you did in the previous assignments. After processing the data you will build a classifier that will allow you to identify whether to questions are the same or not.

You will process the data first and then pad in a similar way you have done in the previous assignment. Your model will take in the two question embeddings, run them through an LSTM, and then compare the outputs of the two sub networks using cosine similarity. Before taking a deep dive into the model, start by importing the data set.

Part 1: Importing the Data ### 1.1 Loading in the data

You will be using the Quora question answer dataset to build a model that could identify similar questions. This is a useful task because you don't want to have several versions of the same question posted. Several times when teaching I end up responding to similar questions on piazza, or on other community forums. This data set has been labeled for you. Run the cell below to import some of the packages you will be using.

```
In [ ]: import os
        import nltk
        import trax
        from trax import layers as tl
        from trax.supervised import training
        from trax.fastmath import numpy as fastnp
        import numpy as np
        import pandas as pd
        import random as rnd

        # set random seeds
        trax.supervised.trainer_lib.init_random_number_generators(34)
        rnd.seed(34)
```

Notice that for this assignment Trax's numpy is referred to as `fastnp`, while regular numpy is referred to as `np`.

You will now load in the data set. We have done some preprocessing for you. If you have taken the deeplearning specialization, this is a slightly different training method than the one you have seen there. If you have not, then don't worry about it, we will explain everything.

```
In [ ]: data = pd.read_csv("questions.csv")
        N=len(data)
        print('Number of question pairs: ', N)
        data.head()
```

We first split the data into a train and test set. The test set will be used later to evaluate our model.

```
In [ ]: N_train = 300000
        N_test  = 10*1024
        data_train = data[:N_train]
        data_test  = data[N_train:N_train+N_test]
        print("Train set:", len(data_train), "Test set:", len(data_test))
        del(data) # remove to free memory
```

As explained in the lectures, we select only the question pairs that are duplicate to train the model. We build two batches as input for the Siamese network and we assume that question $q1_i$ (question i in the first batch) is a duplicate of $q2_i$ (question i in the second batch), but all other questions in the second batch are not duplicates of $q1_i$.
The test set uses the original pairs of questions and the status describing if the questions are duplicates.

```
In [ ]: td_index = (data_train['is_duplicate'] == 1).to_numpy()
        td_index = [i for i, x in enumerate(td_index) if x]
        print('number of duplicate questions: ', len(td_index))
        print('indexes of first ten duplicate questions:', td_index[:10])
```

```
In [ ]: print(data_train['question1'][5])  #  Example of question duplicates (first one in dat
        print(data_train['question2'][5])
        print('is_duplicate: ', data_train['is_duplicate'][5])
```

```
In [ ]: Q1_train_words = np.array(data_train['question1'][td_index])
        Q2_train_words = np.array(data_train['question2'][td_index])

        Q1_test_words = np.array(data_test['question1'])
        Q2_test_words = np.array(data_test['question2'])
        y_test  = np.array(data_test['is_duplicate'])
```

Above, you have seen that you only took the duplicated questions for training our model. You did so on purpose, because the data generator will produce batches ($[q1_1, q1_2, q1_3, ...]$, $[q2_1, q2_2, q2_3, ...]$) where $q1_i$ and $q2_k$ are duplicate if and only if $i = k$.

Let's print to see what your data looks like.

```
In [ ]: print('TRAINING QUESTIONS:\n')
        print('Question 1: ', Q1_train_words[0])
        print('Question 2: ', Q2_train_words[0], '\n')
        print('Question 1: ', Q1_train_words[5])
        print('Question 2: ', Q2_train_words[5], '\n')

        print('TESTING QUESTIONS:\n')
        print('Question 1: ', Q1_test_words[0])
        print('Question 2: ', Q2_test_words[0], '\n')
        print('is_duplicate =', y_test[0], '\n')
```

You will now encode each word of the selected duplicate pairs with an index. Given a question, you can then just encode it as a list of numbers.

First you tokenize the questions using `nltk.word_tokenize`. You need a python default dictionary which later, during inference, assigns the values 0 to all Out Of Vocabulary (OOV) words. Then you encode each word of the selected duplicate pairs with an index. Given a question, you can then just encode it as a list of numbers.

```
In [ ]: #create arrays
        Q1_train = np.empty_like(Q1_train_words)
        Q2_train = np.empty_like(Q2_train_words)

        Q1_test = np.empty_like(Q1_test_words)
        Q2_test = np.empty_like(Q2_test_words)
```

```
In [ ]: # Building the vocabulary with the train set         (this might take a minute)
        from collections import defaultdict

        vocab = defaultdict(lambda: 0)
        vocab['<PAD>'] = 1

        for idx in range(len(Q1_train_words)):
            Q1_train[idx] = nltk.word_tokenize(Q1_train_words[idx])
            Q2_train[idx] = nltk.word_tokenize(Q2_train_words[idx])
            q = Q1_train[idx] + Q2_train[idx]
            for word in q:
                if word not in vocab:
                    vocab[word] = len(vocab) + 1
        print('The length of the vocabulary is: ', len(vocab))
```

```
In [ ]: print(vocab['<PAD>'])
        print(vocab['Astrology'])
        print(vocab['Astronomy'])  #not in vocabulary, returns 0
```

```
In [ ]: for idx in range(len(Q1_test_words)):
            Q1_test[idx] = nltk.word_tokenize(Q1_test_words[idx])
            Q2_test[idx] = nltk.word_tokenize(Q2_test_words[idx])
```

```
In [ ]: print('Train set has reduced to: ', len(Q1_train) )
        print('Test set length: ', len(Q1_test) )
```

1.2 Converting a question to a tensor
You will now convert every question to a tensor, or an array of numbers, using your vocabulary built above.

```
In [ ]: # Converting questions to array of integers
        for i in range(len(Q1_train)):
            Q1_train[i] = [vocab[word] for word in Q1_train[i]]
            Q2_train[i] = [vocab[word] for word in Q2_train[i]]
```

```
for i in range(len(Q1_test)):
    Q1_test[i] = [vocab[word] for word in Q1_test[i]]
    Q2_test[i] = [vocab[word] for word in Q2_test[i]]
```

```
In [ ]: print('first question in the train set:\n')
        print(Q1_train_words[0], '\n')
        print('encoded version:')
        print(Q1_train[0],'\n')

        print('first question in the test set:\n')
        print(Q1_test_words[0], '\n')
        print('encoded version:')
        print(Q1_test[0])
```

You will now split your train set into a training/validation set so that you can use it to train and evaluate your Siamese model.

```
In [ ]: # Splitting the data
        cut_off = int(len(Q1_train)*.8)
        train_Q1, train_Q2 = Q1_train[:cut_off], Q2_train[:cut_off]
        val_Q1, val_Q2 = Q1_train[cut_off: ], Q2_train[cut_off:]
        print('Number of duplicate questions: ', len(Q1_train))
        print("The length of the training set is:  ", len(train_Q1))
        print("The length of the validation set is: ", len(val_Q1))
```

1.3 Understanding the iterator

Most of the time in Natural Language Processing, and AI in general we use batches when training our data sets. If you were to use stochastic gradient descent with one example at a time, it will take you forever to build a model. In this example, we show you how you can build a data generator that takes in $Q1$ and $Q2$ and returns a batch of size `batch_size` in the following format $([q1_1, q1_2, q1_3, ...], [q2_1, q2_2, q2_3, ...])$. The tuple consists of two arrays and each array has `batch_size` questions. Again, $q1_i$ and $q2_i$ are duplicates, but they are not duplicates with any other elements in the batch.

The command `next(data_generator)`returns the next batch. This iterator returns the data in a format that you could directly use in your model when computing the feed-forward of your algorithm. This iterator returns a pair of arrays of questions.

Exercise 01

Instructions:
Implement the data generator below. Here are some things you will need.

- While true loop.
- if `index >= len_Q1`, set the `idx` to 0.
- The generator should return shuffled batches of data. To achieve this without modifying the actual question lists, a list containing the indexes of the questions is created. This list can be shuffled and used to get random batches everytime the index is reset.
- Append elements of $Q1$ and $Q2$ to `input1` and `input2` respectively.
- if `len(input1) == batch_size`, determine `max_len` as the longest question in `input1` and `input2`. Ceil `max_len` to a power of 2 (for computation purposes) using the following command: `max_len = 2**int(np.ceil(np.log2(max_len)))`.

- Pad every question by `vocab['<PAD>']` until you get the length `max_len`.
- Use yield to return `input1, input2`.
- Don't forget to reset `input1, input2` to empty arrays at the end (data generator resumes from where it last left).

In [ ]:
```python
# UNQ_C1 (UNIQUE CELL IDENTIFIER, DO NOT EDIT)
# GRADED FUNCTION: data_generator
def data_generator(Q1, Q2, batch_size, pad=1, shuffle=True):
    """Generator function that yields batches of data

    Args:
        Q1 (list): List of transformed (to tensor) questions.
        Q2 (list): List of transformed (to tensor) questions.
        batch_size (int): Number of elements per batch.
        pad (int, optional): Pad character from the vocab. Defaults to 1.
        shuffle (bool, optional): If the batches should be randomnized or not. Default
    Yields:
        tuple: Of the form (input1, input2) with types (numpy.ndarray, numpy.ndarray)
        NOTE: input1: inputs to your model [q1a, q2a, q3a, ...] i.e. (q1a,q1b) are dup
              input2: targets to your model [q1b, q2b,q3b, ...] i.e. (q1a,q2i) i!=a ar
    """

    input1 = []
    input2 = []
    idx = 0
    len_q = len(Q1)
    question_indexes = [*range(len_q)]

    if shuffle:
        rnd.shuffle(question_indexes)

    ### START CODE HERE (Replace instances of 'None' with your code) ###
    while True:
        if idx >= len_q:
            # if idx is greater than or equal to len_q, set idx accordingly
            # (Hint: look at the instructions above)
            idx = len_q
            # shuffle to get random batches if shuffle is set to True
            if shuffle:
                rnd.shuffle(question_indexes)

        # get questions at the `question_indexes[idx]` position in Q1 and Q2
        q1 = Q1[question_indexes[idx]]
        q2 = Q2[question_indexes[idx]]

        # increment idx by 1
        idx += 1
        # append q1
```

```
                input1.append(q1)
                # append q2
                input2.append(q2)
                if len(input1) == batch_size:
                    # determine max_len as the longest question in input1 & input 2
                    # Hint: use the `max` function.
                    # take max of input1 & input2 and then max out of the two of them.
                    max_len = max(max([len(q) for q in input1]),max([len(q) for q in input2]))
                    # pad to power-of-2 (Hint: look at the instructions above)
                    max_len = 2**int(np.ceil(np.log2(max_len)))
                    b1 = []
                    b2 = []
                    for q1, q2 in zip(input1, input2):
                        # add [pad] to q1 until it reaches max_len
                        q1 = q1 + [pad] * (max_len - len(q1))
                        # add [pad] to q2 until it reaches max_len
                        q2 = q2 + [pad] * (max_len - len(q2))
                        # append q1
                        b1.append(q1)
                        # append q2
                        b2.append(q2)
                    # use b1 and b2
                    yield np.array(b1), np.array(b2)
            ### END CODE HERE ###
                    # reset the batches
                    input1, input2 = [], []  # reset the batches
```

```
In [ ]: batch_size = 2
        res1, res2 = next(data_generator(train_Q1, train_Q2, batch_size))
        print("First questions  : ",'\n', res1, '\n')
        print("Second questions : ",'\n', res2)
```

Note: The following expected output is valid only if you run the above test cell ***once*** (first time). The output will change on each execution.

If you think your implementation is correct and it is not matching the output, make sure to restart the kernel and run all the cells from the top again.

Expected Output:

```
First questions  :
 [[  30   87   78  134 2132 1981   28   78  594   21    1    1    1    1
     1    1]
 [  30   55   78 3541 1460   28   56  253   21    1    1    1    1    1
     1    1]]

Second questions :
 [[  30  156   78  134 2132 9508   21    1    1    1    1    1    1    1
     1    1]
 [  30  156   78 3541 1460  131   56  253   21    1    1    1    1    1
     1    1]]
```

Now that you have your generator, you can just call it and it will return tensors which correspond to your questions in the Quora data set.Now you can go ahead and start building your neural network.

Part 2: Defining the Siamese model

1.1.1 2.1 Understanding Siamese Network

A Siamese network is a neural network which uses the same weights while working in tandem on two different input vectors to compute comparable output vectors.The Siamese network you are about to implement looks like this:

You get the question embedding, run it through an LSTM layer, normalize v_1 and v_2, and finally use a triplet loss (explained below) to get the corresponding cosine similarity for each pair of questions. As usual, you will start by importing the data set. The triplet loss makes use of a baseline (anchor) input that is compared to a positive (truthy) input and a negative (falsy) input. The distance from the baseline (anchor) input to the positive (truthy) input is minimized, and the distance from the baseline (anchor) input to the negative (falsy) input is maximized. In math equations, you are trying to maximize the following.

$$\mathcal{L}(A, P, N) = \max\left(\|\mathrm{f}(A) - \mathrm{f}(P)\|^2 - \|\mathrm{f}(A) - \mathrm{f}(N)\|^2 + \alpha, 0\right)$$

A is the anchor input, for example $q1_1$, P the duplicate input, for example, $q2_1$, and N the negative input (the non duplicate question), for example $q2_2$. α is a margin; you can think about it as a safety net, or by how much you want to push the duplicates from the non duplicates.

Exercise 02

Instructions: Implement the `Siamese` function below. You should be using all the objects explained below.

To implement this model, you will be using `trax`. Concretely, you will be using the following functions.

- `tl.Serial`: Combinator that applies layers serially (by function composition) allows you set up the overall structure of the feedforward. docs / source code
  - You can pass in the layers as arguments to `Serial`, separated by commas.
  - For example: `tl.Serial(tl.Embeddings(...), tl.Mean(...), tl.Dense(...), tl.LogSoftmax(...))`
- `tl.Embedding`: Maps discrete tokens to vectors. It will have shape (vocabulary length X dimension of output vectors). The dimension of output vectors (also called d_feature) is the number of elements in the word embedding. docs / source code
  - `tl.Embedding(vocab_size, d_feature)`.
  - `vocab_size` is the number of unique words in the given vocabulary.
  - `d_feature` is the number of elements in the word embedding (some choices for a word embedding size range from 150 to 300, for example).
- `tl.LSTM` The LSTM layer. It leverages another Trax layer called `LSTMCell`. The number of units should be specified and should match the number of elements in the word embedding. docs / source code
  - `tl.LSTM(n_units)` Builds an LSTM layer of n_units.

- `tl.Mean`: Computes the mean across a desired axis. Mean uses one tensor axis to form groups of values and replaces each group with the mean value of that group. docs / source code
  - `tl.Mean(axis=1)` mean over columns.
- `tl.Fn` Layer with no weights that applies the function f, which should be specified using a lambda syntax. docs / source doce
  - x -> This is used for cosine similarity.
  - `tl.Fn('Normalize', lambda x: normalize(x))` Returns a layer with no weights that applies the function `f`
- `tl.parallel`: It is a combinator layer (like `Serial`) that applies a list of layers in parallel to its inputs. docs / source code

In [ ]:
```python
# UNQ_C2 (UNIQUE CELL IDENTIFIER, DO NOT EDIT)
# GRADED FUNCTION: Siamese
def Siamese(vocab_size=len(vocab), d_model=128, mode='train'):
    """Returns a Siamese model.

    Args:
        vocab_size (int, optional): Length of the vocabulary. Defaults to len(vocab).
        d_model (int, optional): Depth of the model. Defaults to 128.
        mode (str, optional): 'train', 'eval' or 'predict', predict mode is for fast i

    Returns:
        trax.layers.combinators.Parallel: A Siamese model.
    """

    def normalize(x):  # normalizes the vectors to have L2 norm 1
        return x / fastnp.sqrt(fastnp.sum(x * x, axis=-1, keepdims=True))

    ### START CODE HERE (Replace instances of 'None' with your code) ###
    q_processor = tl.Serial(  # Processor will run on Q1 and Q2.
        tl.Embedding(vocab_size, d_model), # Embedding layer
        tl.LSTM(d_model), # LSTM layer
        tl.Mean(axis=1), # Mean over columns
        tl.Fn('Normalize', lambda x: normalize(x))  # Apply normalize function
    )  # Returns one vector of shape [batch_size, d_model].

    ### END CODE HERE ###

    # Run on Q1 and Q2 in parallel.
    model = tl.Parallel(q_processor, q_processor)
    return model
```

Setup the Siamese network model

In [ ]:
```python
# check your model
model = Siamese()
print(model)
```

Expected output:

```
Parallel_in2_out2[
  Serial[
    Embedding_41699_128
    LSTM_128
    Mean
    Normalize
  ]
  Serial[
    Embedding_41699_128
    LSTM_128
    Mean
    Normalize
  ]
]
```

1.1.2 2.2 Hard Negative Mining

You will now implement the `TripletLoss`. As explained in the lecture, loss is composed of two terms. One term utilizes the mean of all the non duplicates, the second utilizes the *closest negative*. Our loss expression is then:

$$\mathcal{L}\wr\int\int_{\infty}(\mathcal{A},\mathcal{P},\mathcal{N}) = \max\left(-cos(A,P) + mean_{neg} + \alpha, 0\right) \tag{1}$$
$$\mathcal{L}\wr\int\int_{\in}(\mathcal{A},\mathcal{P},\mathcal{N}) = \max\left(-cos(A,P) + closest_{neg} + \alpha, 0\right) \tag{2}$$
$$\mathcal{L}\wr\int\int(\mathcal{A},\mathcal{P},\mathcal{N}) = mean(Loss_1 + Loss_2) \tag{3}$$
$$\tag{4}$$

Further, two sets of instructions are provided. The first set provides a brief description of the task. If that set proves insufficient, a more detailed set can be displayed.
Exercise 03
Instructions (Brief): Here is a list of things you should do:

- As this will be run inside trax, use `fastnp.xyz` when using any `xyz` numpy function
- Use `fastnp.dot` to calculate the similarity matrix $v_1 v_2^T$ of dimension `batch_size` x `batch_size`
- Take the score of the duplicates on the diagonal `fastnp.diagonal`
- Use the `trax` functions `fastnp.eye` and `fastnp.maximum` for the identity matrix and the maximum.

More Detailed Instructions We'll describe the algorithm using a detailed example. Below, V1, V2 are the output of the normalization blocks in our model. Here we will use a batch_size of 4 and a d_model of 3. As explained in lecture, the inputs, Q1, Q2 are arranged so that corresponding inputs are duplicates while non-corresponding entries are not. The outputs will have the same pattern. This testcase arranges the outputs, v1,v2, to highlight different scenarios. Here, the first two outputs V1[0], V2[0] match exactly - so the model is generating the same vector for Q1[0] and Q2[0] inputs. The second outputs differ, circled in orange, we set, V2[1] is set to match V2[**2**],