Homework 07

IANNwTF

December 10, 2022

DISCLAIMER: Section 2.1.1 of this homework is being rewritten to include a better approach of finding the targets. You can still make this solution work, but we advise you to check back in a few days to download the revised version of this homework.

This homework's deadline is *22.12., 23:59*.

Submit your homework via https://forms.gle/ApAZ5ubY8ewgNmJA9

Remember to review the work done by two of your fellow groups!

Contents

1 Recap on Reviews

Welcome back to your seventh assignment in IANNwTF. Here's just a short refresher on how to do reviews.

In addition to handing in your homework, you will have to review last week's homework of two groups and note down which group you reviewed on the homework submission form. This requires you to find two other groups and have a short (10 min) meeting where you go over each others homework submission, discuss it and then write a short summary of what you discussed on each others forum pages. We recommend using the Q&A timeslots for this purpose, but you can meet however and whenever you like. The main review part of this should be the discussion you have together. The written review in the forum should merely be a recap of the main points you discussed so that we can see that you did something and the reviewed group has access to a reminder of the feedback they received. **If there are any open questions regarding your or any other groups code afterwards, please feel invited to discuss this with us in the QnA sessions, so we can help you sort out how a perfect solution would have looked like!** Just to make this obvious: We will not penalize any submission (by you or any of the groups you reviewed) based on such questions. The purpose is helping you understand and code better only.

As to how a discussion could look like, you could for example have the group being reviewed walking the other two groups through their code. The other two groups should try to give feedback, e.g. "I like how you designed your data pipeline, looks very efficient.", "For this function you implemented there actually exists a method in e.g. NumPy or TensorFlow you could've used instead." or "Here you could've used a slightly different network architecture and higher learning rate to probably achieve better results.", and note down their main points in the forum.

Important! Not every member of every group has to be present for this. You could for example implement a rotation where every group member of yours only has to participate in a review every third week.

2 Assignment: Implement LSTM

Welcome back! This week, you will build your own recurrent neural network! You will do this by combining a CNN with LSTM cells. Finally, you will need to train the resulting network on specially prepared sequences of MNIST data. For the most part, you should be able to reuse the CNN which you implemented previously. The LSTM cells, however, you need to implement from scratch. Once your LSTM cells are ready, you will wrap them with a RNN wrapper layer that handles the 'unrolling' for you. This should help you understand the inner workings of an RNN cell, as well as teach you how custom RNNs are built in TensorFlow.

If you struggle with any of the following tasks, remember to make use of all the resources available to you. Come to our Q&A sessions, post your questions in the forum, and don't forget that most precious resource of them all: your fellow students!

2.1 Prepare the Dataset

Once again, you'll be dealing with the MNIST dataset, but with another twist. Instead of classifying handwritten digits, your network must learn to, in an alternating manner, add and subtract subsequent digits from a total sum. That should look something like this: A sequence of these 4 images...

4	1	0	7

should be associated with the following operations...

+4=4	4-1=3	3+0=3	3-7=-4

which result in these targets:

4	3	3	-4

For training you will have to divide the images up into sequences that will be fed into your model. The shape of your tensors should be (batch, sequence-length, features). After each sequence ends the value of the first digit in the next sequence should reset. You can try out different sequence lengths in combination with batch sizes to see what works best. To create the sequences you could simply `.batch()` twice, once to create the sequences and the second time to create the batches.

Once you are done, your network should be able to predict the correct target at each time step.

2.1.1 Finding the targets

DISCLAIMER: The solution described here is sub-optimal and this section is being reworked. You can still make it work, but we advise you to check back in a few days to download the revised version of this homework.

One way to replace the targets is to generate a list containing the new values from the old ones, use
`tf.data.Dataset.from_tensor_slices()` to convert the list to a dataset, zip the original MNIST dataset and your new targets together with
`tf.data.Dataset.zip(train_ds, train_new_targets)` and then drop/replace the old targets in your data pipeline using
`data = data.map(lambda img, target: (img[0], target))`.

Remember that you have to set the target of the first sample of each batch to the old value and not keep summing up over the whole dataset. Since generating this new dataset should not be the main focus of this homework we have prepared a sample solution for you. This is just one of many solutions though, you could also for example try to use a generator to generate the new targets. Feel free to implement your own if you feel like it.

```
def new_target_fnc(ds, sequence_len):
  """
  Creates list of new targets by alternately adding and subtracting
  The first digit is added, the second subtracted, the third added, etc
  Parameters
  ----------
  ds : TensorFlowDataset
    original mnist dataset containg images and targets as a tuple.
  sequence_len : int
    indicates at which point the sum has to reset for the new sequence
  Returns
  -------
  l : list
    list containing the new targets
  """
  l = list()
  for i, elem in enumerate(ds):
    if (i % sequence_len) == 0:
      l.append(int(elem[1]))
    else:
      if (i % 2) == 0:
        l.append(int(l[i-1] + elem[1]))
      else:
        l.append(int(l[i-1] - elem[1]))
  return l
```

2.1.2 Create the data pipeline

For your data pipeline, all the regular steps needed for image processing apply. It would make sense to replace the old targets with the new ones using `.map`. Remember to create the sequences before batching and most importantly before shuffling!

2.2 The CNN & LSTM Network

The first part of your model should have a basic CNN structure. This part should extract vector representations from each MNIST image using Conv2D layers as well as (global) pooling or Flatten layers. A Conv2D layer can be called on a batch of sequences of images, where the time dimension is in the second axis. The time dimension will then be processed like a second batch dimension (search for "extended batch shape" in the Conv2D layer documentation page for an example).

While Conv2D layers accept our (batch, sequence-length, image) data structure with their extended batch size functionality, for the pooling layers to work correctly you will have to wrap them in TensorFlow's TimeDistributed layers.

Once you have encoded all images as vectors, the shape of the tensor should be (batch, sequence-length, features), which can be fed to a non-convolutional standard LSTM.

2.3 LSTM AbstractRNNCell layer

For the LSTM, we want to not use the (optimized) keras implementation, but instead we want you to be able to implement the LSTM logic that is applied at each time-step yourselves. For this, we want to subclass the AbstractRNNCell layer and implement its methods and define the required properties. Those are **state_size**, **output_size**, and **get_initial_state**, which determines the initial hidden and cell state of the LSTM (usually tensors filled with zeros).

The LSTM-cell layer's call method should take one (batch of) feature vector(s) as its input, along with the "states", a list containing the different state tensors of the LSTM cell (cell state and hidden state!).

In the call method, the layer then uses these two states together with the vector representation of the current MNIST image in the sequence, and updates both the hidden state, and the cell state (in the way that it is done in an LSTM). The returns should be the output of the LSTM, to be used to compute the model output for this time-step (usually the hidden state), as well as a list containing the new states (e.g. [new_hidden_state, new_cell_state]).

2.4 Wrapping the LSTM-Cell layer with an RNN layer

Since the LSTM cell only provides the computation for one time-step, you would need to write a wrapper layer around it, that applies it to every time-step in the sequence, aggregating the outputs and states in a for loop.

In Tensorflow you should use the RNN layer for this, tf.keras.layers.RNN takes an instance of your LSTM cell as the first argument in its constructor. You also need to specify whether you want the RNN wrapper layer to return the output of your LSTM-cell for every time-step or only for the last step (with the argument return_sequences=True). This is generally task-dependent, so think about what makes most sense in this case. For speed-ups (at the cost of memory usage) you can set the "unroll" argument to True.

The "wrapper" RNN layer then takes the sequence of vector representations of the mnist images as its input (batch, seq_len, feature_dim).

2.5 Computing the model output

With the output of your RNN-wrapped LSTM-Cell, you can now compute the model predictions. Again depending on the task, you need to think about what your predictions should be (generally have one prediction per target that is associated with your sequence). Dense layers also behave in the same way as Conv2D layers - when there is an additional time-dimension (batch, time, features), they apply the same computation for every time-index and for every batch-index. So you could (if the task demands it) use the same Dense layer to predict targets for all time-steps. You likely do not want to have a Dense layer for each time-step's target prediction (potential for overfitting!).

2.6 Training

We still highly recommend to write another custom training-loop for this homework. However, if you want to follow the code presented in this week, feel free to already use the model.compile and model.fit methods for the first time. Make sure to track your experiments properly, save configs (e.g. hyperparameters) of your settings, save logs (e.g. with Tensorboard) and checkpoint your model's weights (or even the complete model). To visualize your results you can modify your training loop to also write metrics to lists, or rely on the default history callback that model.fit uses.

2.7 Questions

If any questions arise, chances are that other students are wondering about the same things. Use the studIP forum for this, and come to the Q&A sessions.

For an example of how to define a custom RNN and use the compile fit methods, have a second look at the notebook presented in this week's courseware.

Hint: to debug your code, you can exchange your own LSTM implementation with tf.keras.layers.LSTM and see if that fixes your issues. For better performance in your implementation, you may for instance want to initialize your LSTM's recurrent kernels with the orthogonal initializer:

tf.keras.initializers.Orthogonal.

Good luck and have fun!