Homework 07

IANNWTF 21/22

Submission until 12 Dec 23:59 via https://forms.gle/n6ERdhYx3uBPzuGn9

Welcome back! You should have learned quite a lot about RNNs in the courseware.

There are some problems with RNNs that will make this homework a bit special.

First, most problems we would be using RNNs for simply are a bit too large in scope to easily train in reasonable time for a homework. Thus, we will ask you to implement your own dataset using a toy task. This is actually a useful skill - when you might end up working with more complex customized architectures, you might actually want to test if they work in principle on a small very easy dataset before you throw it onto a big one and lose sight of where you should start tweaking and debugging.

Secondly, we will ask you to not use the keras implementation of a LSTM cell, but instead write your own implementation of a LSTM cell. The learning goal for you here is twofold: On the one hand and most importantly, this allows you to go through implementing an LSTM in detail, which is really important in understanding how it works after all. On the other hand, it is the perfect playground to work on some advanced implementation skills, which while not necessary for a running LSTM cell are very useful for one that runs fast - and understanding these tools will help you to generally build better running and scaling Tensorflow models. Applying these advanced implementation approaches for your LSTM implementation are not necessary to pass this homework, but they are required to obtain an *outstanding* this week.

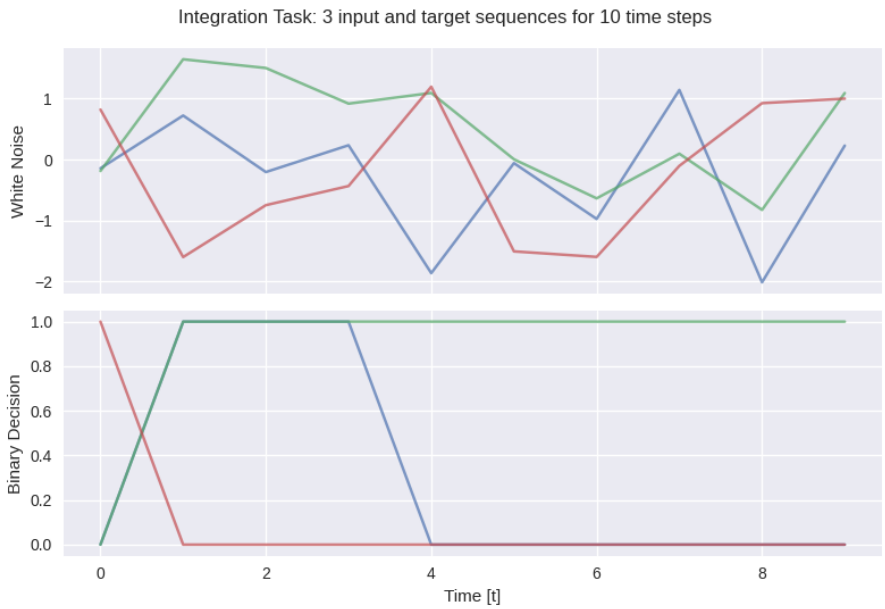
We know this homework can be a bit tricky, but believe us, we have the best intentions: We really just want you to understand some very important concepts throughout. Thus, if you struggle, feel free to drop by at our Coding Support, ask questions on Element or use the self-help telegram chat. We tutors and most certainly some other motivated students will try our best to help you!

1 The task

1.1 Task Description

The task is a simple integration task: The input consists of noise over a certain amount of time steps, let us say that n_t denotes the noise signal at time step t. The target is again a binary decision on whether the integral of the white noise up to that time step t is positive or negative.

See an example for 3 noise-target pairs in the image below.



In the image, we see a many-to-many task, but we will write a many-to-one training pipeline, thus only caring about the integral at the last time step.

1.2 Generate a Tensorflow Datset

You can of course now go on and compute a big bunch of examples and store them in a huge array or dataframe or whatever you like. However, that would be rather inefficient - as we work with random noise input, we can compute the data on the fly.

Thus, we would like you to work with **generators** [1] and the tensorflow function `tf.data.Dataset.from_generator()`.

There are a few simple steps to take:

- **Write a `integration_task(seq_len, num_sapmles)` generator function:** This function should for `num_samples` times [2] **yield** a random noise signal of the size seq_len [3] (sequence length, thus the number of time steps

we are considering) and a target, namely if the sum of the noise signal is greater or smaller than 1. [4]

You make your life easier if you do not try to handle empty dimensions![5]

- **Write a wrapper generator** `my_integration_task()`: As it is easier to pass a generator to the tensorflow `tf.data.Dataset.from_generator()` method that does not take any arguments, write a wrapper function which internally iterates to integration_task with a specified `seq_len` and num_samples and yields the function's yield (yes sounds stupid and complicated, but it's really easier that way).

  Choose `num_samples` to be rather high [6], as you do not want to overfit and you may want to start with a small sequence length for easier debugging. In the end, you should however be able to deal with a sequence length of 25 at least.

- **Create the Tensorflow Dataset:** Pass on your wrapper generator to `tf.data.Dataset.from_generator(, output_signature= tf.TensorSpec(shape, dtype)`.

1.3 Create a Data pipeline

Now you should have a Tensorflow Dataset object and thus should be able to apply all the necessary preprocessing steps to create an efficient pipeline. [7]

2 The network

While there are great implementations in tensorflow/keras for LSTMs, this week you are asked to build your own! So you are restricted from using any tensorflow/keras inbuilt of LSTM (or RNN!). You can however use Dense layers for the matrix multiplications.

To do so, there are several steps you need to take:

1. **Implement a** `LSTM_Cell` which is called by providing the input for a *single* times step and a tuple containing (hidden_state, cell_state).

   - `__init__(self,units)`: Remember again what gates an LSTM consists of and how to parametrize them. [8] Pay special attention to weight initialization: According to Josezefowicz et al. Setting the bias of the forget gate to one initially is very important for performance in training LSTMs. [9] Think about it for a moment - what effect would that have on the initial output values of this gate, and what effect will that have on the recurrent cell state? This is actually implemented in the default LSTM cell in Tensorflow - check the *unit_forget_bias* argument there for more information!

- `call(self, x, states)`: To implement the call function of the LSTM cell, think about the different pathways and how the input and the different states are combined. [10]

2. **Implement a LSTM layer**, which is created from one (or multiple for a multi-layer LSTM) LSTM Cell. This LSTM layer should operate on inputs with *multiple* time steps.

   - `__init__(self,cell)`: It is easier to start implementing single cell layers but you may think about ways to implement multi-layer LSTMs.
   - `call(self, x, states)` The call function takes the input over *multiple* time steps and creates (and returns!) the outputs over multiple time steps. The input is expected to be of shape [batch_size, seq_len, input_size], the respective output of shape [batch_size, seq_len, output_size] To achieve this you will have to "unroll" the LSTM.

     To learn more about "unrolling" if you are unsure what it means and want to know how to implement it efficiently, go to the subsection about Unrolling.
   - `zero_states(self, batch_size)`: Define a function that, given a batch size, resets the states of the LSTM, thus returns a tuple of states of the appropriate size filled with zeros. [11]

3. **Implement the final Model:** This model should be a wrapper around your lstm implementation.

   - `__init__()`: You may want to use one or multiple read_in layers and an output layer that takes the output of your LSTM and transforms it into your final prediction. [12]
   - `call(self,x)`: Here the input is over the whole sequence length. You should pass it through your read-in layers and then to your LSTM implementation [13] and finally to your read-out layer.

2.1 Unrolling a LSTM:

Unrolling the network is quite easy in Eager Execution mode: You could just create a python list, iterate over your sequence length, pass on each single time step to your LSTM cell and store the output for the specific time steps in the list and return it.

If you want to just pass the homework, you can do it like that, however, we advise you to think about the following:

To do this "unrolling" correctly (i.e. efficiently) in graph mode (graph mode is the name for what happens if you decorate a function with the "@tf.function" decorator) you will have to understand a bit more of how graph mode works. Specifically, you will probably have to solve the following sub-problems:

- In graph mode, we have to be extra careful to appropriately design *conditional - if_else* and *loops*. Specifically, for proper handling in graph mode,

the respective condition (or sequence) arguments have to be tensors. For unrolling the LSTM over multiple time steps you will probably need this. Check out this part of the Tensorflow guide to get familiar with the details!

- Typically, you would use a list to aggregate results over multiple time steps. Appending to typical python lists won't work (efficiently or not at all, depending on implementation details) in graph mode though it can work if you use @tf.function(experimental_relax_shapes=True) but this is experimental and not recommended. Check out the documentation of the so called TensorArray to find the typical solution and have a look at the later parts of the notebook we provide in the Implementing RNNs using Keras section.

For just passing this week you won't need to actually implement the points above (even though we strongly recommend you at least read about them), but for an outstanding we expect proper graph mode implementations!

3 Training

Training mostly stays the same to any other binary classification task you have tackled so far. However, there is a key difference: You get a prediction on the level of several time steps, but for your loss and your accuracy, we only want to consider the prediction of the very last time step [14].

Think of this task as a proof of concept task - getting some reasonable accuracy is really enough, no need to try and push it this week. So don't feel pressured to use any sort of more advanced optimization.

Be aware that training might take longer, especially if you decide not to make use of graph mode. It is enough if you can see a significant improvement in loss and accuracy after the second epoch of training, and it is very reasonable to get to at least 80% accuracy.

4 How-To Outstanding

Of course, the usual still holds: Your solution should be designed in a way, that they could serve as sample solutions to students that may not have successfully completed the task on their own. Also, people should be able to further work with your code. Thus provide context, explanations when necessary, especially comment on the meaning and the types(!) of any arguments of functions / methods that you implemented. **Explanation and typing of arguments will from now on be a hard requirement - we want you to get used to good conduct!**

Also, for an outstanding, your implementation should be *efficient*. This means:

- Use a generator to implement your Dataset and make your pipeline efficient.

- Implement the unrolling of the LSTM in Graph Mode.

There are a few things we would like you to think about (and for an outstanding, quickly comment on those questions):

- Can / should you use truncated BPTT here?
- Should you rather take this as a regression, or a classification problem?

Have fun!

Notes

[1] check out this link to see how that's done in a really easy way with the yield statement!

[2] You can just use a for loop iterating over `num_samples`.

[3] You may want to make use of `np.random.normal(size=).`

[4] You can use np.sum(axis=) along the last axis and compare it to 0. Remember your output should be at best of type 'integer' not 'boolean'

[5] Use np.expand_dims(,-1) before you yield. The output shapes should be (seq_len,1) and (1) for input and target respectiveley.

[6] For example, we chose 80.000.

[7] Shuffling, Batching.... As always.

[8] You can use a Dense layer for each gate with a hidden size specified by `units` and a sigmoid activation.

[9] To do so, check out the argument `bias_initializer=` of the keras Dense layer implemenation and the `tf.keras.initializers` objects.

[10] For the concatenation of the input and the `hidden_state` to pass on to the forget gate, you can make use of `tf.concat( (), axis=)`.

[11] The appropiate size would be `bacth_size, cell_units` and you may want to use `tf.zeros()`

[12] Consider that you have a binary classification task. How many units should your readout layer haven and what activation function should you use?

[13] Do not forget to initialize the LSTM states with zeros each time you call your model and pass those on to your LSTM

[14] You may want to use slicing, -1 is always the last irrespective of the size