Paul Voigtlaender <voigtlaender@vision.rwth-aachen.de>
Francis Engelmann <engelmann@vision.rwth-aachen.de>

Exercise 6: Recurrent Neural Networks

due **before** 2018-02-01

Important information regarding the exercises:

- The exercises are not mandatory. Still, we strongly encourage you to solve them! All submissions will be corrected. If you submit your solution, please read on:
- Use the L^{2}P system to submit your solution. You will also find your corrections there.
- Due to the large number of participants, we require you to submit your solution to L^{2}P **in groups of 3 to 4 students**. You can use the **Discussion Forum** on L^{2}P to organize groups.
- If applicable submit your code solution as a zip/tar.gz file named `mn1_mn2_mn3.{zip/tar.gz}` with your **matriculation numbers** (`mn`).
- Please do **not** include the data files in your submission!
- Please upload your pen & paper problems as PDF. Alternatively, you can also take pictures (.png or .jpeg) of your hand written solutions. Please make sure your handwriting is legible, the pictures are not blurred and taken under appropriate lighting conditions. All non-readable submissions will be discarded immediately.

Question 1: Basic RNN models ..... $(\Sigma = 15)$

This exercise deals with the implementation of a basic RNN and LSTM cell. To this end, you are given a code framework which will be completed in the following tasks. The implementation is based on the abstract class `RNNCell` that defines the basic interface for our RNNs. You should complete the interface implementation for the child classes `BasicRNNCell` and `MyLSTMCell`.

Specifically, this comprises implementing a `__call__` method for each cell which basically adds an unrolled RNN to TensorFlow's graph that runs on a given input. The outputs are fed into a linear layer on top of the cell in order to map the cell output to the desired dimension. The parameters of this layer are shared among all steps.

The network will then be trained to continue a sine curve of an unknown frequency after being presented a few initial samples. To this end, ground truth samples from a training sine curves are presented to the cell which in every time step should then predict the next sample using the linear layer on top. The prediction error is measured by a mean squared error over the entire training sequence.

After training the cell, we will graphically evaluate its capability of continuing a test sine curve. Therefore, a few samples from the beginning of the test curve are fed into the cell in order to initialize the state. We retrieve the final initialization state and the last output which corresponds to our first prediction. Finally, we iteratively run the cell on the current state and current (predicted) sine value in order to generate the remaining curve.

We will also need the python package `matplotlib` so, before starting, make sure to install it using `pip install matplotlib`.

(a) Add the variables needed for a basic RNN cell to `BasicRNNCell.__init__`. Remember to use an adequate initialization. **(1 pt)**

(b) As mentioned above, we use `RNNCell.__call__` to run a RNN cell on an input starting from a given state. This state is described by a tuple whose entries are tensors where dimension 0 is used for batch processing and dimension 1 corresponds to the state information. The input is a tensor with dimensions as follows: 0: Batch, 1: Time steps, 2: Input value for the batch and time step. The function should return a list of output tensors corresponding to the cell output at every time step and the state tensor after the last step. Implement this for a basic RNN cell in `BasicRNNCell.__call__`. **(4 pts)**
Hint: Basic means: $\mathbf{h}_t = \begin{pmatrix}\mathbf{x}_t & \mathbf{h}_{t-1}\end{pmatrix} W + \mathbf{b}$, where the brackets denote concatenation.
Hint: The state is a `MyBasicRNNState` tuple (see BasicRNNCell.py).

(c) Now, we will implement our version of a basic LSTM cell. Start by adding the variables to `MyLSTMCell.__init__`. **(2 pts)**

(d) As for the basic RNN cell, implement the `MyLSTMCell.__call__` function. **(5 pts)**
Hint: The state is described by a `MyLSTMState` tuple containing hidden and cell state, respectively (see MyBasicRNNState.py).

(e) As mentioned above, we want our cell to generate a sine curve within a range of frequencies. Therefore, implement the mean squared error calculation in `train.get_loss_sin`. **(1 pt)**
Hint: The ground truth is a 2d tensor where rows are used for the batch and columns correspond to sine evaluation steps.
Hint: If `RNNCell.__call__` is implemented correctly, the ith column is the ground truth value for the cell prediction at the ith step. No shifting needed.

(f) Finally, implement `train.get_train_op` which should return a gradient descent based training operation. Remember that in general gradient clipping may be useful to cope with exploding gradients. You can now test the sine curve generation. One possible configuration that may work is set as default. If you try the LSTM, you may want to have a fairer comparison in terms of number of parameters. How can you achieve this? **(1 pt)**
Hint: You may want to use `tf.clip_by_global_norm` and a norm of 5.0

(g) We now want to challenge the memory capacities of our basic RNN cell and our implementation of the LSTM. To this end, you should implement a memory task which tests the length of past to present dependencies that the cell can learn. Therefore, the cell is trained to run on the sequence **(1 pt)**

$$\begin{pmatrix} r_1 & \cdots & r_{\text{to\_remember\_len}} & \underbrace{\text{␣} \quad \cdots \quad \text{␣}}_{\text{blank\_separation\_len } times} & \# & r_1 & \cdots & r_{\text{to\_remember\_len}} \end{pmatrix}$$

with $r_1, \ldots, r_{\text{to\_remember\_len}}$ chosen from $\{0, \ldots, 8\}$ uniformly at random while generating the output

$$\begin{pmatrix} & \underbrace{\text{␣} \quad \cdots \quad \text{␣}}_{\text{to\_remember\_len+blank\_separation\_len+1 } times} & & r_1 & \cdots & r_{\text{to\_remember\_len}} \end{pmatrix}.$$

Thus, the challenge is to remember the sequence at the beginning over a certain number of blanks and to start recalling after the delimiter.
First, complete `train.get_loss_memory` using an appropriate loss and return a single loss value for the given batch of outputs. You should simply consider the outputs as being equally import so no weighting is necessary.

(h) Implement the method `test.run_memory_test` which evaluates the memory capability of the trained cell. After running the test iterator initializer you can evaluate the given loss on the test dataset. Besides, it may be more interesting to calculate the accuracy of recalling the first digits. Therefore, also calculate the accuracy for the last to_remember_len outputs. Which model does perform better?
Hint: See `train.run_training` to see how the test data can be consumed.
Hint: You might want to try to increase the number of training epochs for this task.