George Lydakis <lydakis@vision.rwth-aachen.de>
Stephanie Kaes <kaes@vision.rwth-aachen.de>

Exercise 6: Recurrent Neural Networks

due **before** 2024-01-31

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 Moodle 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 Moodle **in groups of 3 to 4 students**. You can use the **Discussion Forum** on Moodle 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 model and BPTT ($\Sigma = 9$)

This exercise deals with the implementation of a basic RNN and LSTM cell. It will introduce you to the challenges of implementing a basic RNN from scratch by providing guidance for the key steps in the process, namely forward propagation and backward propagation through time (BPTT). This will be followed by a test case, where the network is trained to remember an arbitrary given sequence across several time steps, and outputting the same sequence after reading a specific symbol. This kind of application tests the memory capabilities of the RNN, as well as the ability to learn the concept of first remembering and then repeating.

Requirements: The following packages are required: `numpy`, `torch` and `matplotlib`. Please install any missing packages using `pip install "package"` (without colons) from the command line.

(a) **Forward propagation of basic RNN:** As a first task, you are asked to complete the code of the `BasicRNNCell.fprop` function. Remember that the equations for h_t, the hidden state at time t are given by: **(2 pts)**

$$\mathbf{h}_{t+1} = \mathrm{ReLU}\left(\mathbf{W}_{hh} \cdot \mathbf{h}_t + \mathbf{W}_{hx} \cdot \mathbf{x}_t + \mathbf{b}_h\right)$$

Where $\mathbf{W}$ are the weight matrices and $\mathbf{x}$ the input. Take note, that for simplicities sake, $\mathbf{h}_0$ is always initialized to zero. In a real application environment, the state of $\mathbf{h}_0$ would be an additional parameter that can be passed to the function. The linear mapping to the output variable y

$$\mathbf{y}_t = \mathbf{W}_{yh} \cdot \mathbf{h}_t$$

is done using the separate class `LinearLayer` so that multiple layers of `BasicRNNCells` may be stacked on top of each other.

Hint: For information about the interface you may want to take a look at the implementation for the linear layer already finished in `LinearLayer.py` as well as `BasicRNNCell.__init__` for the variable names.

(b) **BPTT of basic RNN:** Now you are asked to implement BPTT by completing the code in `BasicRNNCell.bprop`. In the following, $\hat{\mathbf{h}}_t$ will denote the derivative with respect to the hidden state at time t before the ReLU is applied. The BPTT steps for the forward-propagation above can be written as follows: **(3 pts)**

$$\begin{aligned}
\partial \mathbf{h}_t &= W_{yh}^T \partial \mathbf{y}_t + \mathbf{W}_{hh}^T \cdot \partial \hat{\mathbf{h}}_{t+1} \\
\partial \hat{\mathbf{h}}_t &= \mathrm{ReLU}_t\left(\partial \mathbf{h}_t\right) \\
\partial \mathbf{W}_{hh} &+= \partial \hat{\mathbf{h}}_t \cdot \mathbf{h}_{t-1}^T \\
\partial \mathbf{W}_{hx} &+= \partial \hat{\mathbf{h}}_t \cdot \mathbf{x}_t^T \\
\partial \mathbf{b}_h &+= \partial \hat{\mathbf{h}}_t \\
\partial \mathbf{x}_t &= \mathbf{W}_{hx}^T \cdot \partial \hat{\mathbf{h}}_t
\end{aligned}$$

The += is necessary in order to accumulate the gradient over time. It is important not to calculate the ReLU function on the derivatives, but to apply the same ReLU function, that was previously applied in timestep t. This means, that you need to remember which values were set to 0 in all timesteps t. In the backpropagation you need to set the same values of $\partial \hat{\mathbf{h}}_t = 0$ as these values cannot pass down a derivative, since they were set to zero in the forward-propagation. It is also of note, that in the code the derivatives dys which are given as an input to the bprop function are the derivatives that are passed down from higher layers, i.e. the linear layer, and have already been multiplied with W_{yh}^T, i.e. dys_t will be the result of $W_{yh}^T \cdot \partial \mathbf{y}_t$. Additionally, $\partial \hat{\mathbf{h}}_{t+1}$ should be initialized to zero for $t = t_{\text{end}}$. Once you have completed implementing the forward- and backward-propagation, you can use the provided `run_check_grads` function to verify, that both are working as intended.

Hint: For information about the interface you may want to take a look at the implementation for the linear layer already finished in `LinearLayer.py`.

(c) **Exploding gradient:** As was already introduced in the lecture, one big flaw of recurrent networks, is that the gradient of parameter matrices may vanish or become extremely large. After you have completed the previous tasks, use the function `run_exploding_grads_test` to visualize how quickly the mean absolute value of the gradient can increase. You may also try out different numbers for the hidden layer, sequence length and input vocabulary to see how this affects the magnitude of the gradient. **(0 pts)**

(d) **Gradient clipping:** As a method for mitigation of exploding gradients, gradient clipping was introduced. This method computes the norm of the gradient after BPTT and scales it to a fixed value. The pseudo-code for the algorithm is presented in the following **(1 pt)**

```
if $||\texttt{grad}|| > \textit{threshhold}$ then
    grad $\leftarrow \frac{\textit{threshhold}}{||\texttt{grad}||}$ grad
```

Your task is to implement `common.clip_gradients`. After you finished your implementation use `run_exploding_grads_test` to make sure that the gradient clipping has taken effect correctly.

Hint: In order to make sure that the changes take effect in the gradient, which is passed as an argument to the function, you should use the `/=` or `*=` assignment operators.

(e) **Memorization Task:** Finally, we want our RNN to learn to perform a simple memorization task. The goal is to take an input sequence and read until a specific character is encountered. After this point the RNN should output all the characters that were previously read in the correct order. The input sequence will look something like **(2 pts)**

$$\left(c_1 \quad \cdots \quad c_{\text{to\_remember\_len}} \quad \# \quad \underbrace{\textvisiblespace \quad \cdots \quad \textvisiblespace}_{\text{to\_remember\_len } \textit{times}} \quad \right)$$

whereas the target output sequence will look like

$$\left(\quad \underbrace{\textvisiblespace \quad \cdots \quad \textvisiblespace}_{\text{to\_remember\_len } \textit{times}} \quad \# \quad c_1 \quad \cdots \quad c_{\text{to\_remember\_len}}\right)$$

To this end you are required to implement the function `run_memory_test` which computes the accuracy of the network on a randomly generated test set of data. For data generation you may want to use the function `generate_data_memory`, which returns data and label matrices. Additionally, the function `run_forward_pass` may be helpful to you in order to obtain the soflty assigned classes. Remember, that the only interesting values for the accuracy measure are the ones generated after the delimiter has been read. Afterwards you may try to adjust the command line parameters (e.g. learning rate, number of steps) so that the test accuracy is maximized.

(f) **Vanishing gradient:** One method to fix the vanishing gradient issue that was presented, is to initialize $\mathbf{W}_{hh}$ to the identity matrix and to use ReLU non-linearities. As the latter is already implemented in the code, your task is to change the initialization of $\mathbf{W}_{hh}$ to the identity matrix in `BasicRNNCell.__init__`. Do you notice any change in the accuracy of the memorization task? **(1 pt)**

Question 2: LSTM in PyTorch .... $(\Sigma = 10)$

(a) **Implementing LSTM:** Now we will use PyTorch to implement our version of an LSTM cell as presented in the lecture. Start by adding the variables to `MyLSTMCell.__init__`. **(1 pt)**

Hint: You may want to take a look at the implementation in `BasicRNNCell.__init__` for a point of reference. Remember that an LSTM maintains state in two variables, the so-called "hidden state" and "cell state".

(b) Similar to the BasicRNN case, implement the `MyLSTMCell.__call__` function. **(5 pts)**
The input $\mathbf{x}$ 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 tensor of the same shape as $\mathbf{x}$ corresponding to the cell output at every time step. *Hint:* Take a look at the implementation of `BasicRNNCell.py`. The code here will be similar, the only difference being that the LSTM update equations are more complicated.

(c) Finally, add code to `loops.train_loop` such that for each batch a forward and backward pass is performed with the model, loss function and optimizer given as arguments. Remember that in general gradient clipping may be useful to cope with exploding gradients. You can now test the sine curve generation by setting the command line parameter --task 1. In this setup, the first 30 samples of a sin curve generated with a **(2 pts)**

random frequency and phase shift are given as an input to the network (so that it can have some initial information about the sine) and afterwards the network extrapolates the curve by feeding back its own predictions as its input. The results are then written out as png files in the folder of the source code. The configuration that uses the basic RNN is set as default. To use the LSTM, set the command line parameter --RNN LSTM. If you try the LSTM, you may want to have a fairer comparison in terms of number of parameters. How can you achieve this?
Hint: You may want to use gradient clipping with a norm of 5.0. Look up the appropriate function that does this in PyTorch.

(d) Now, we 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. For this task, the cell is trained to run on the sequence **(1 pt)**

$$\big(r_1 \quad \dots \quad r_{\text{to\_remember\_len}} \qquad \underbrace{\text{␣} \ \cdots \ \text{␣}}_{\text{blank\_separation\_len } \textit{times}} \qquad \# \quad r_1 \quad \dots \quad r_{\text{to\_remember\_len}}\big)$$

with $r_1, \dots, r_{\text{to\_remember\_len}}$ chosen from $\{0, \dots, 7\}$ uniformly at random while generating the output

$$\big(\qquad \underbrace{\text{␣} \ \cdots \ \text{␣}}_{\text{to\_remember\_len+blank\_separation\_len+1 } \textit{times}} \qquad r_1 \quad \dots \quad r_{\text{to\_remember\_len}}\big) \, .$$

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 `memory_task.memory_task_loss` using an appropriate loss (remember: this is a classification problem) and return a single scalar loss value obtained by summing over the individual losses for each element of the batch.

(e) Now we want to write a simple testing loop for our task, which should also evaluate the accuracy of the prediction. To this end, complete the function `memory_task.memory_task_acc` which should calculate the accuracy of the predicted outputs in the last to_remember_len timesteps of the sequences in a batch. Then, complete the function `loops.test_loop_mem_task`, which should use the created data loader to loop through the dataset, run the forward pass and compute the loss and accuracy for each batch using the appropriate functions. Which model performs better? **(1 pt)**
Hint: You might want to try to increase the number of training epochs for this task.

For this question, you'll find the tutorials and documentation of PyTorch (`https://pytorch.org/tutorials/`)quite useful. Additionally, you should be aware that there exist many useful libraries which hide away some of the "boilerplate" code we provided for you. One such library is PyTorch Lightning (`https://www.pytorchlightning.ai/`), which is not used for this basic exercise so as not to distract you from the process of converting deep learning math into code. However, if you want, you can take a look at what it offers.