Exercise Sheet 1: Recurrent Models

Due on 28.06.2024, 10:00

Tao Hu (tao.hu@lmu.de)

Important Notes:

1. **Email**: Frequently check your email address registered for Moodle. All notifications regarding the course will be sent via Moodle.
2. **Due date**: The exercise sheets will usually be uploaded around 1 week in advance of the due date, which is indicated at the top of the exercise sheet.
3. **Submission**: Your submission should consist of one single ZIP file which includes a PDF file and the corresponding codes. Both the PDF file and ZIP file should contain your surname and your matriculation number (*Surname-MatriculationNumber.zip*) for grading purposes. You may use Jupyter [1] for exporting your python notebooks as PDF, but you still have to hand in your *.ipynb* or *.py* files for us to test your code. For this exercise, please export the *.ipynb* as a PDF file and include that in the ZIP file. **Submissions that fail to follow the naming convention will not be graded.**

This exercise may look intimidating by its length, however, this is because it contains general information about submissions and also many explanations and hints that will help you through the exercise.

General Information:

All programming exercises must be completed in Python. The proposed solution for the exercises will be compiled in Python 3 (3.8 or above). You may use standard Python libraries or Anaconda (open source distribution for Python) to complete your exercises. We will be using PyTorch[2] as our main deep learning framework.

If you have any problems or questions about the exercise, you are welcome to use the student forum on the lecture Moodle page, as most of the time other students might have similar question. For technical issues about the course (for example, in case you cannot upload the solution to Moodle) you can write an email to the person responsible for the exercise (indicated at the top of the exercise sheet).

[1] https://jupyter.org/
[2] https://pytorch.org/

For this task, you will compare vanilla Recurrent Neural Networks (RNN) with Long-Short Term Networks (LSTM). PyTorch has a large amount of building blocks for recurrent neural networks. However, to get you familiar with the concept of recurrent connections, in this assignment you will implement a vanilla RNN and LSTM from scratch. The use of high-level operations such as `torch.nn.RNN`, `torch.nn.LSTM` and `torch.nn.Linear` is not allowed.

In this first assignment, we will focus on very simple sequential training data for understanding the memorization capability of recurrent neural networks. More specifically, we will study *palindrome* numbers. Palindromes are numbers which read the same backward as forward, such as:

303

4224

175282571

682846747648286

We can use a recurrent neural network to predict the next digit of the palindrome at every timestep. While very simple for short palindromes, this task becomes increasingly difficult for longer palindromes. For example when the network is given the input 68284674764828_ and the task is to predict the digit on the _ position, the network has to remember information from 14 timesteps earlier. If the task is to predict the last digit only, the intermediate digits are irrelevant. However, they may affect the evolution of the dynamic system and possibly erase the internally stored information about the initial values of input. In short, this simple problem enables studying the memorization capability of recurrent networks.

For the coding assignment, in the file `part1/dataset.py`, we have prepared the class `PalindromeDataset` which inherits from `torch.utils.data.Dataset` and contains the function `generate_palindrome` to randomly generate palindrome numbers. You can use this dataset directly in PyTorch and you do not need to modify contents of this file. Note that for short palindromes the number of possible numbers is rather small, but we ignore this sampling collision problem for the purpose of this assignment.

Task 1: Vanilla RNN in PyTorch (8P)

The vanilla RNN is formalized as follows. Given a sequence of input vectors $\mathbf{x}^{(t)}$ for $t = 1, \dots T$, the network computes a sequence of hidden states $\mathbf{h}^{(t)}$ and a sequence of output vectors $\mathbf{p}^{(t)}$ using the following equations for timesteps $t = 1, \dots, T$:

$$\mathbf{h}^{(t)} = \tanh(\mathbf{W}_{hx}\mathbf{x}^{(t)} + \mathbf{W}_{hh}\mathbf{h}^{(t-1)} + \mathbf{b}_h) \tag{1}$$

$$\mathbf{p}^{(t)} = \mathbf{W}_{ph}\mathbf{h}^{(t)} + \mathbf{b}_p \tag{2}$$

As you can see, there are several trainable weight matrices and bias vectors. $\mathbf{W}_{hx}$ denotes the input-to-hidden weight matrix, $\mathbf{W}_{hh}$ is the hidden-to-hidden (or recurrent) weight matrix, $\mathbf{W}_{ph}$ represents the hidden-to-output weight matrix and the $\mathbf{b}_h$ and $\mathbf{b}_p$ vectors denote the biases. For the first timestep $t = 1$, the expression $\mathbf{h}^{(t-1)} = \mathbf{h}^{(0)}$ is replaced with a special vector $\mathbf{h}_{init}$ that is commonly initialized to a vector of zeros. The output value $\mathbf{p}^{(t)}$ depends on the state of the hidden layer $\mathbf{h}^{(t)}$ which in its turn depends on all previous state of the hidden layer. Therefore, a recurrent neural network can be seen as a (deep) feed-forward network with shared weights.

To optimize the trainable weights, the gradients of the RNN are computed via back-propagation through time (BPTT). The goal is to calculate the gradients of the loss $\mathcal{L}$ with respect to the model parameters $\mathbf{W}_{hx}, \mathbf{W}_{hh}$ and $\mathbf{W}_{ph}$ (biases omitted). Similar to training a feed-forward network, the weights and baises are updated using SGD or one of its variants. Different from feed-forward networks, recurrent networks can give output logits $\hat{\mathbf{y}}^{(t)}$ at every timestep. In this assignment the outputs will be given by the softmax function, *i.e.* $\hat{\mathbf{y}}^{(t)} = \text{softmax}(\mathbf{p}^{(t)})$. For the task of predicting the final palindrome number, we compute the standard cross-entropy loss *only* over the last timestep:

$$\mathcal{L} = -\sum_{k=1}^{K} \mathbf{y}_k \log \hat{\mathbf{y}}_k \tag{3}$$

Where k runs over the number of classes ($K = 10$ because we have ten digits). In this expression, $\mathbf{y}$ denotes a one-hot vector of length K containing true labels.

Question 1.1 (2P)

Recurrent neural networks can be trained using backpropagation through time. Similar to feed-forward networks, the goal is to compute the gradients of the loss w.r.t. $\mathbf{W}_{ph}$, $\mathbf{W}_{hh}$ and $\mathbf{W}_{hx}$. Write down an expression for the gradient $\frac{\partial \mathcal{L}^{(T)}}{\partial \mathbf{W}_{ph}}$ in terms of the variables that appear in Equations 1 and 2.

Do the same for $\frac{\partial \mathcal{L}^{(T)}}{\partial \mathbf{W}_{hh}}$. What difference do you observe in temporal dependence of the two gradients. Study the latter gradient and explain what problems might occur when training this recurrent network for a large number of timesteps.

Solution 1.1

For the weight matrix $\mathbf{W}_{ph}$ (hidden to output) the gradient only depends on the current timestep. However, for the other weight matrices $\mathbf{W}_{hx}$ and $\mathbf{W}_{hh}$ the gradient contains a summation term over the previous timesteps. The final answers are given here, the intermediate steps are not: https://goo.gl/uXvL78.

Question 1.2 (2P)

Implement the vanilla recurrent neural network as specified by the equations above in the file `vanilla_rnn.py`. For the *forward* pass you will need Python's `for`-loop to step through time. You need to initialize the variables and matrix multiplications yourself without using high-level PyTorch building blocks. The weights and biases can be initialized using `torch.nn.Parameter`. The *backward* pass does not need to be implemented by hand, instead you can rely on automatic differentiation and use the RMSProp optimizer for tuning the weights. We have prepared boilerplate code in `part1/train.py` which you should use for implementing the optimization procedure.

Solution 1.2

Proper weight initialization for the RNN is important for fast convergence. Providing an initial state of zeros is crucial for making it work.

1. 0.5 points for correct hidden init (e.g zeros) and reinitialized at every forward
2. 0.5 points for correct param init with torch.nn.Parameter
3. 0.5 points for correct forward pass with for loop; correct support of batches
4. 0.5 points if the matrix multiplication makes sense (do not overcheck stuff here)

Solution 1.2 and 1.6

These tables are not updated with the numbers obtained using the new PyTorch implementation. When I compiled this table last year I did not carefully tune the hyper-parameters. In practice, students should be able to obtain better numbers.

Table 1. Vanilla RNN versus LSTM on the task of predicting the final digit. For this experiment the following settings were used. Hidden layer size: **64**, learning rate: **0.05**, batch size: 128, train steps: 2000, gradient clip: 10.

Seq. Length	Vanilla RNN	LSTM
5	0.70	0.94
10	0.24	0.93
20	0.11	0.92
50	0.12	0.11

Table 2. Vanilla RNN versus LSTM on the task of predicting the final digit. For this experiment the following settings were used. Hidden layer size: **256**, learning rate: **0.02**, batch size: 128, train steps: 2000, gradient clip: 10.

Seq. Length	Vanilla RNN	LSTM
5	0.74	0.97
10	0.22	0.96
20	0.11	0.95
50	0.11	0.90
100	0.11	0.11

Question 1.3 (2P)

As the recurrent network is implemented, you are ready to experiment with the memorization capability of the vanilla RNN. Given a palindrome of length T, use the first $T-1$ digits as input and make the network predict the last digit. The network is *successful* if it correctly predicts the last digit and thus was capable of memorizing a small amount of information for T timesteps.

Start with short palindromes ($T = 5$), train the network until convergence and record the accuracy. Repeat this by gradually increasing the sequence length and create a plot that shows the accuracy versus palindrome length. As a sanity check, make sure that you obtain a near-perfect accuracy for $T = 5$ with the default parameters provided in `part1/train.py`.

Solution 1.3

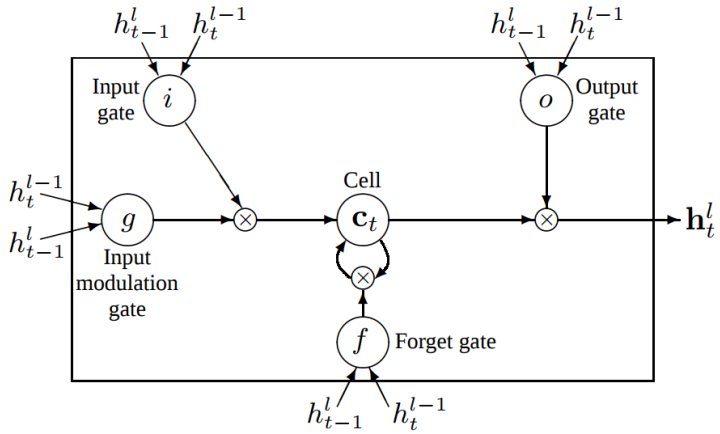
See optimization note in Solution 1.2.
My implementation converges quickly for palindrome lengths < 15. After that training becomes more difficult for the Vanilla RNN. When increasing the sequence length, the learning rate should be lowered. I started with a learning rate of 0.005 and gradually decreased it when training on longer sequences. Training should be successful in less than 2000 steps (128 batch size). This should not be too hard for the students.

1. 1 points for coherent/detailed explanation of plots and experiments, including hyperparameter configuration
2. 1 points for reporting results with increasing T values and expected decay of performances with large Ts (until 15 at least should reach 100%).

Question 1.4 (2P)

To improve optimization of neural networks, many variants of stochastic gradient descent have been proposed. Two popular optimizers are RMSProp and Adam and/or the use of momentum. In practice, these methods are able to converge faster and obtain better local minima. In your own words, write down the benefits of such methods in comparison to vanilla stochastic gradient descent. Your answer needs to touch upon the concepts of **momentum** and **adaptive learning rate**.

Figure 1. A graphical representation of LSTM memory cells (Zaremba *et al.* (ICLR, 2015))



Solution 1.4

1. 1 point for correct explanation of momentum
2. 1 point for correct explanation of adaptive learning rate; and general comparison wrt SGD.

Task 2: Long-Short Term Network (LSTM) in PyTorch (8P)

As you have observed after implementing the previous questions, training a vanilla RNN for remembering its inputs for an increasing number of timesteps is difficult. The problem is that the influence of a given input on the hidden layer (and therefore on the output layer), either decays or blows up exponentially as it unrolls the network. In practice, the *vanishing gradient problem* is the main shortcoming of vanilla RNNs. As a result, training a vanilla RNNs to consistently learn tasks containing delays of more than ~ 10 timesteps between relevant input and target is difficult. To overcome this problem, many different RNN architectures have been suggested. The most widely used variant is the Long-Short Term Network (LSTMs). An LSTM (Figure 1) introduces a number of gating mechanisms to improve gradient flow for better training. Before continuing, please read the following blogpost: **Understanding LSTM Networks**.

In this assignment we will use the following LSTM definition:

$$\mathbf{g}^{(t)} = \tanh(\mathbf{W}_{gx}\mathbf{x}^{(t)} + \mathbf{W}_{gh}\mathbf{h}^{(t-1)} + \mathbf{b}_g) \tag{4}$$
$$\mathbf{i}^{(t)} = \sigma(\mathbf{W}_{ix}\mathbf{x}^{(t)} + \mathbf{W}_{ih}\mathbf{h}^{(t-1)} + \mathbf{b}_i) \tag{5}$$
$$\mathbf{f}^{(t)} = \sigma(\mathbf{W}_{fx}\mathbf{x}^{(t)} + \mathbf{W}_{fh}\mathbf{h}^{(t-1)} + \mathbf{b}_f) \tag{6}$$
$$\mathbf{o}^{(t)} = \sigma(\mathbf{W}_{ox}\mathbf{x}^{(t)} + \mathbf{W}_{oh}\mathbf{h}^{(t-1)} + \mathbf{b}_o) \tag{7}$$
$$\mathbf{c}^{(t)} = \mathbf{g}^{(t)} \odot \mathbf{i}^{(t)} + \mathbf{c}^{(t-1)} \odot \mathbf{f}^{(t)} \tag{8}$$
$$\mathbf{h}^{(t)} = \tanh(\mathbf{c}^{(t)}) \odot \mathbf{o}^{(t)} \tag{9}$$

$$\mathbf{p}^{(t)} = \mathbf{W}_{ph}\mathbf{h}^{(t)} + \mathbf{b}_p \tag{10}$$
$$\hat{\mathbf{y}}^{(t)} = \text{softmax}(\mathbf{p}^{(t)}). \tag{11}$$

In these equations $\odot$ is element-wise multiplication and $\sigma(\cdot)$ is the sigmoid function. The first six equations are the LSTM's core part whereas the last two equations are just the linear output mapping. Note that the LSTM has more weight matrices than the vanilla RNN. As the forward pass of the LSTM is relatively intricate, writing down the correct gradients for the LSTM would involve a lot of derivatives. Fortunately, LSTMs can easily be implemented in PyTorch and automatic differentiation takes care of the derivatives.

Question 1.5 (3P)

(a) The LSTM extends the vanilla RNN cell by adding four gating mechanisms. Those gating mechanisms are crucial for successfully training recurrent neural networks. The LSTM has an *input modulation gate* $\mathbf{g}^{(t)}$, *input gate* $\mathbf{i}^{(t)}$, *forget gate* $\mathbf{f}^{(t)}$ and *output gate* $\mathbf{o}^{(t)}$. For each of these gates, write down a brief explanation of their purpose; explicitly discuss the non-linearity they use and motivate why this is a good choice.

(b) Given the LSTM cell as defined by the equations above and an input sample $\mathbf{x} \in \mathbb{R}^{T \times d}$ where T denotes the sequence length and d is the feature dimensionality. Let n denote the number of units in the LSTM and m represents the batch size. Write down the formula for the *total number* of trainable parameters in the *LSTM cell* as defined above.

Solution 1.5

(a) [1.5 points if explanation is there] Solution can be found here: https://arxiv.org/pdf/1506.00019.pdf
(See page 18 for a good explanation on the purpose of all gating mechanisms)

(b) [1.5 points if everything is correct. NB: the parameters for the linear output is optional, as not properly part of the LSTM cell] Note that the batch size and number of timesteps are not necessary.:

$$N = 4N_{hidden}^2 + 4N_{hidden}(1 + N_{input}) + (N_{hidden} + 1) \cdot N_{output} \tag{12}$$

or

$$N = 4N_{hidden}^2 + 4N_{hidden}(1 + N_{input}) \tag{13}$$

Question 1.6 (5P)

Implement the LSTM network as specified by the equations above in the file `lstm.py`. Just like for the Vanilla RNN, you are required to implement the model without any high-level PyTorch functions. You do not need to implement the *backward* pass yourself, but instead you can rely on automatic differentiation and use the RMSProp optimizer for tuning the weights. For the optimization part we have prepared the code in `train.py`.

Using the palindromes as input, perform the same experiment you have done in *Question 1.3*. Train the network until convergence. You might need to adjust the learning rate when increasing the sequence length. The initial parameters in the prepared code provide a starting point. Again, create a plot of your results by gradually increasing the sequence length. Write down a comparison with the vanilla RNN and think of reasons for the different behavior. As a sanity check, your LSTM should obtain near-perfect accuracy for $T = 5$.

Solution 1.2 and 1.6

For the experiments + plots, like in 1.3 for RNN:

1. coherent/detailed explanation of plots and experiments, including hyperparameter configuration
2. reporting results with increasing T values and expected decay of performances with large Ts (until 15 at least should reach 100%).

Table 3. Vanilla RNN versus LSTM on the task of predicting the final digit. For this experiment the following settings were used. Hidden layer size: **64**, learning rate: **0.05**, batch size: 128, train steps: 2000, gradient clip: 10.

Seq. Length	Vanilla RNN	LSTM
5	0.70	0.94
10	0.24	0.93
20	0.11	0.92
50	0.12	0.11

Table 4. Vanilla RNN versus LSTM on the task of predicting the final digit. For this experiment the following settings were used. Hidden layer size: **256**, learning rate: **0.02**, batch size: 128, train steps: 2000, gradient clip: 10.

Seq. Length	Vanilla RNN	LSTM
5	0.74	0.97
10	0.22	0.96
20	0.11	0.95
50	0.11	0.90
100	0.11	0.11

Important Note: Submit exactly one ZIP file via Moodle before the deadline. The ZIP file should contain your executable code and your report in PDF format. Make sure that it runs on different operating systems and use relative paths. Non-trivial sections of your code should be explained with short comments, and variables should have self-explanatory names. The PDF file should contain your written code, all figures, explanations and answers to questions. Make sure that plots have informative axis labels, legends, and captions. Missing plots will result in point reduction.