Assignment 2. Recurrent Neural Networks and Graph Neural Networks

Maurice Frank
11650656
maurice.frank@posteo.de
Code: github

1 Vanilla RNN versus LSTM

1.1 RNN derivatives

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}_{ph}} = \prod \frac{\partial}{\partial} \cdot \frac{\partial}{\partial \boldsymbol{W}_{ph}} \tag{1}$$

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}_{hh}} = \dots \tag{2}$$

1.2 Vanilla RNN code

Find the code inside `vanilla_rnn.py` and `train.py`.

1.3 Vanilla RNN experiment

See Figure 1 for a overview plot of the results and Section 1.6 for a discussion/comparsion of the results.

1.4 Optimizers

SGD has problems. One of them is ocurring oscilattions in valleys of the loss space. SGD does not have any memory and thus just tries to approximate the currents face gradient to follow down which might make the path jump around a minimum of the valley. One change to counter this problem is introducing momentum. Following the intuition of the physical term, the gradient with momentum gets only changed gradually not sudden in every optimizer step. This is implemented as a decaying average of gradient updates. The weights get updated as a weighted sum of the previous update and the new gradient. A second idea is to tweak the learning rate for each weight and not use a fixed η for all, yielding a adaptive learning rate. For those weights that change a lot (bounce around some valley) we want to reduce the update step to counteract the bouncing. This can be seen in the RMSProp [1] optimizer as described below:

$$v_t = \rho v_{t-1} + (1-\rho) \cdot (\nabla_{\theta_t} f)^2 \tag{3}$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{v_t + \epsilon}} \cdot \nabla_{\theta_t} f \tag{4}$$

ρ defines the decying sum. We compute the update but than divide the learning rate η for each weight by the new update. Thus oscillating weights will get a smaller update. Adam [2]

optimizer works quite similar:

$$v_t = \beta_1 \cdot v_{t-1} - (1 - \beta_1) \cdot \nabla_{\theta_t} f \tag{5}$$

$$s_t = \beta_2 \cdot s_{t-1} - (1 - \beta_2) \cdot (\nabla_{\theta_t} f)^2 \tag{6}$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{s_t + \epsilon}} \cdot v_t \tag{7}$$

We also adapt the learning rate per weight by dividing by the square-root of the squared gradients. But here also directly use the momentum but having the decaying sum of weight-wise gradients. $\beta_1$andβ_2 are tuneable hyperparameters.

1.5 LSTM theory

a) LSTM Gates

input modulation gate $\boldsymbol{g}^{(t)}$ The input modulation gate determines candidate information from the new input (using also the old hidden state). We want our state values normalized but need also negative values (otherwise the cell values would only increase) which, as in this case, can be done with a $\tanh$, squashing the input to $[-1, 1]$.

input gate $\boldsymbol{i}^{(t)}$ The input regulates which and how much information of the input of this time step should be included in the cell and hidden state. As the input gate regulates the flow it is necessary to have its values bounded to $[0, 1]$ which can most directly achieved by squashing the values with the sigmoid.

forget gate $\boldsymbol{f}^{(t)}$ The forget gate regulates which and how much information from the old cell state should be disregared under the new information from the input (and the old hidden state). As the forget gate only changes the importance (magnitude) of the information in the cell state it should be in $[0, 1]$ which is achieved with the sigmoid.

output gate $\boldsymbol{o}^{(t)}$ The output gate regulates which and how much information from the new cell state should go into the new hidden state. Again its gating the values from the other tensor which is asking for a range $[0, 1]$ achieved by the sigmoid.

b) Number of parameters

We have given $\boldsymbol{x} \in \mathbb{R}^{T \times d}$ with T sequence length and d feature dimension. Further we have n hidden units. Then we have

$$4 \cdot (d \cdot n + n \cdot n + n)$$

trainable parameters for one LSTM cell. If we want to include the projection onto the classes c the size increases of course to:

$$4 \cdot (d \cdot n + n \cdot n + n) + n \cdot c + c$$

1.6 LSTM practice

Find the code inside `lstm.py` and `train.py`. For a overview check out Figure 1. We see in general that the LSTM is able to learn the palindromes faster and for longer sequences. The RNN is only able to improve on randomness (accuracy of 0.1) up to length 17. Further the RNN is only able to reach full accuracy for palindromes smaller than 10. The LSTM learns for all lengths but we see that for length of 23 we do not see improvement over randomness until almost 3000 (if run for lon enough the LSTM learns full accuracy for all tested lengths but we found that not be the interesting result here). This is explained in that we use the same hyperparameters for all experiments expecially the learning rate. An optimization of the hyperparameters for the LSTM might speed up training for longer sequences. The experiment clearly shows that the LSTM is more capable of learning longer dependencies.

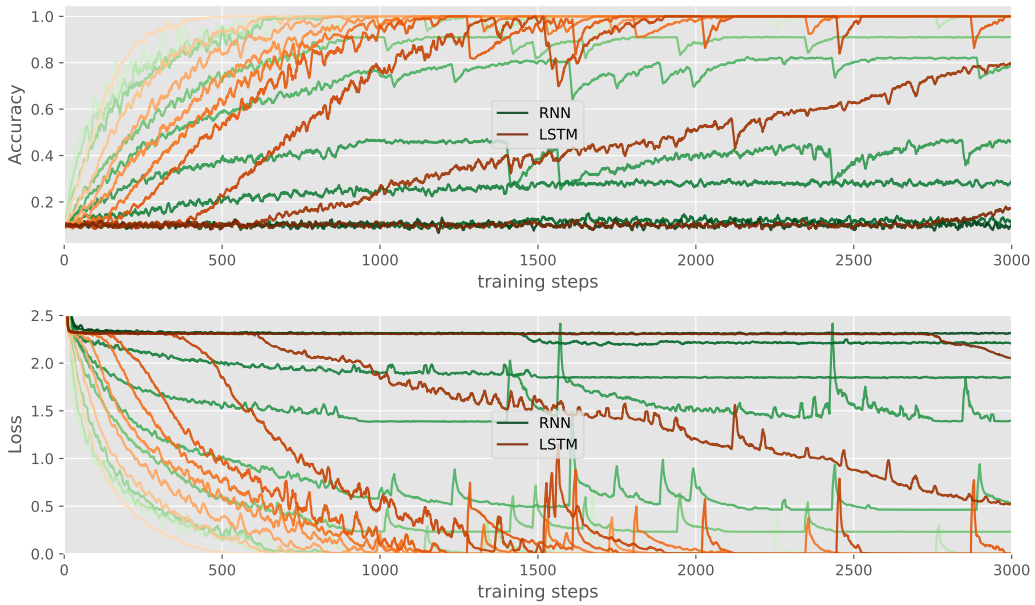

Figure 1: Top the accuracy and bottom the loss while training. Color lightness codes the palindrome length ranging from 5 numbers with the lightest color in steps of 2 to 23 numbers with the darkest color. (Making 9 curves per model). All curves are an average of ten runs and smoothed with a box filter of size 10 for better radibility.

2 Recurrent Nets as Gernerative Model

2.1 Learning South Park

a) The code

Find the code inside `model.py` and `train.py`.

b)

c)

2.2

3 Graph Neural Networks

3.1

a)

b)

3.2

3.3

a)

b)

References

[1] Geoffrey Hinton. Lecture 6 Overview of mini--batch gradient descent. URL http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_

slides_lec6.pdf.

[2] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization.