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

Generally we have:

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

With the two partials:

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial p_i} &= -\sum_j y_j \frac{\partial \log \hat{y}_j}{\partial p_i} && (2)\\
&= -\sum_j \frac{y_j}{\hat{y}_j} \frac{\partial \hat{y}_j}{\partial p_i} && (3)\\
&= -y_i(1-\hat{y}_i) - \sum_{j \neq i} \frac{y_j}{\hat{y}_j} \cdot (-\hat{y}_i \hat{y}_j) && (4)\\
&= -y_i + y_i \hat{y}_i + \hat{y}_i \sum_{j \neq i} y_j && (5)\\
&= \hat{y}_i \left(y_i + \sum_{j \neq i} y_j \right) - y_i && (6)\\
&= \hat{y}_i - y_i && (7)\\
&\iff && (8)\\
\frac{\partial \mathcal{L}}{\partial \boldsymbol{p}} &= \boldsymbol{p} - \boldsymbol{y} && (9)\\
& && (10)
\end{aligned}$$

Note here it holds $\sum_i y_i = 1$ because of the one-hot encoding.

The second derivative is more direct:

$$\frac{\partial \boldsymbol{p}}{\partial \boldsymbol{W}_{ph}} = \boldsymbol{h}^{(T)} \tag{11}$$

$$\tag{12}$$

leads finally to:

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}_{ph}} = (\boldsymbol{p} - \boldsymbol{y}) \cdot \boldsymbol{h}^{(T)} \tag{13}$$

The derivative with respect to the hidden weight we write down in its recursive form:

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}_{hh}} = \frac{\partial \mathcal{L}}{\partial \hat{\boldsymbol{y}}} \frac{\partial \hat{\boldsymbol{y}}}{\partial \boldsymbol{p}} \frac{\partial \boldsymbol{p}}{\partial \boldsymbol{h}^{(T)}} \frac{\partial \boldsymbol{h}^{(T)}}{\partial \boldsymbol{W}_{hh}} \tag{14}$$

$$\frac{\partial \boldsymbol{h}^{(T)}}{\partial \boldsymbol{W}_{hh}} = \left(1 - \boldsymbol{h}^{(T)^2}\right) \cdot \frac{\partial}{\partial \boldsymbol{W}_{hh}} \boldsymbol{W}_{hh} \boldsymbol{h}^{(T-1)} \tag{15}$$

$$= \left(1 - \boldsymbol{h}^{(T)^2}\right) \cdot \left[\left(\frac{\partial}{\partial \boldsymbol{W}_{hh}} \boldsymbol{W}_{hh}\right) \boldsymbol{h}^{(T-1)} + \boldsymbol{W}_{hh} \left(\frac{\partial}{\partial \boldsymbol{W}_{hh}} \boldsymbol{h}^{(T-1)}\right)\right] \tag{16}$$

$$= \left(1 - \boldsymbol{h}^{(T)^2}\right) \cdot \left(\boldsymbol{h}^{(T-1)} + \boldsymbol{W}_{hh} \frac{\partial \boldsymbol{h}^{(T-1)}}{\partial \boldsymbol{W}_{hh}}\right) \tag{17}$$

$$\tag{18}$$

Because we had to write down $\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}_{hh}}$ which we do not do for $\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}_{ph}}$ we directly see the different length of the temporal dependencies in the two computational graphs. For the hidden weights gradient we need to transverse the whole time-sequence while the other on only ever depends on the final hidden state. This makes problems in practical training probable. The partial gradients of the hidden weights might be small and they form a product. The gradient might be vanishing. In practice this might go so far to reach the numerical limits of floating point arithmetic.

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/comparison of the results.

1.4 Optimizer

SGD has problems. One of them is occurring oscillations 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:

$$v_t = \rho v_{t-1} + (1 - \rho) \cdot (\nabla_{\theta_t} f)^2 \tag{19}$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{v_t + \epsilon}} \cdot \nabla_{\theta_t} f \tag{20}$$

ρ defines the decaying 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:

$$v_t = \beta_1 \cdot v_{t-1} - (1 - \beta_1) \cdot \nabla_{\theta_t} f \tag{21}$$

$$s_t = \beta_2 \cdot s_{t-1} - (1 - \beta_2) \cdot (\nabla_{\theta_t} f)^2 \tag{22}$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{s_t + \epsilon}} \cdot v_t \tag{23}$$

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 disregarded 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 especially 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.

2 Recurrent Nets as Generative Model

2.1 Learning South Park

a) The code

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

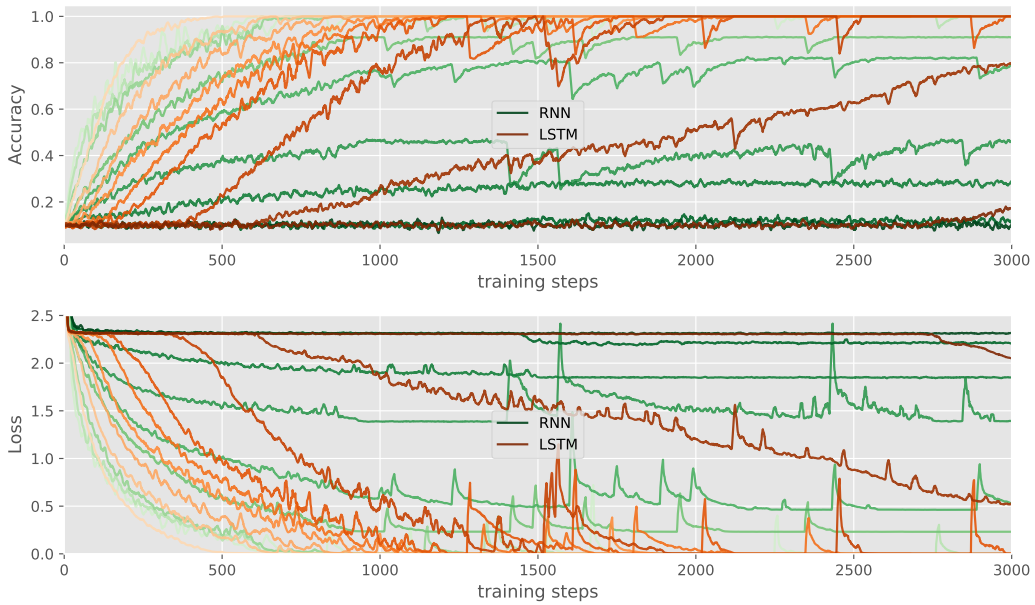

Figure 1: **Top** the accuracy and **bottom** the loss while training. Color lightness codes the palindrome length ranging from 5 numbers with the lightest color in steps of 2 to 23 numbers with the darkest color. (Making 9 curves per model). All curves are an average of ten runs and smoothed with a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

b)

c)

2.2

3 Graph Neural Networks

3.1 Forward Layer

a)

The $\hat{A}$ matrix contains the edge information between the nodes in the graph. This includes the self-connections of all nodes from the identity $\mathbb{1}_N$. We update the activations with $H^{l+1} = \sigma(\hat{A}H^{(l)}W^{(l)})$. So we update the activation associated with one node only for the activations of edges that have non-zero edge weights in $\hat{A}$. A node is changed by the nodes its connected to. In reverse we can see that information in one node can propagate to all connected adjacent nodes in one time step which can be visualized as the message passing over the graph.

b)

In every layer of the GCN you can propagate information along one edge so to let information reach a node three hops away we would need three layers.

3.2 Applications of GNN

In *Multi-Granularity Reasoning for Social Relation Recognition from Images* [3] the authors build graphs on images of humans. The graphs describe first the relationship of a person in the image with its surrounding objects amongst other things other persons in the image. A second graph represents the pose of each person in the image. On these two types of graphs

they use Graph Convolutional Networks (GCN) to predict the social relationship of these persons. For example in an image of parent and child the bending pose of the parent and the connection of the two persons in the Person-Object graph lets the method infer their relationship.

Next in *Disease Prediction using Graph Convolutional Networks: Application to Autism Spectrum Disorder and Alzheimer's Disease* [4] propose to use GCN for medical image processing. The nodes of the graph in this setting are features of medical image acquisitions which in their experiments are gathered from structural and functional MRI. As they predict the health state of multiple individuals at once the graph consists of many image features of multiple persons. The edges here describe the phenotypical similarity between two individuals which are described by categorical medical data (e.g. sex). The GCN uses this graph to predict the health state of the population.

Lastly in *Temporal Relational Ranking for Stock Prediction* [5] we seen an application of GCN in stock prediction. Here the nodes are features capturing the historical information of one stock and the edges capture the relations between two companies stock. Both historical intra and inter stock features are generated by LSTM from historical stock data. Again we build a graph from the edges and nodes and use GCN to predict in this case the stocks next day performance at the market.

3.3 Comparison adn Combination of GNN and RNN

a)

Generally a graph structure is necessary for non-Euclidean spaces.

b)

References

[1] Geoffrey Hinton. Lecture 6 Overview of mini-ŋbatch 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.

[3] Meng Zhang, Xinchen Liu, Wu Liu, Anfu Zhou, Huadong Ma, and Tao Mei. Multi-Granularity Reasoning for Social Relation Recognition from Images. URL http://arxiv.org/abs/1901.03067.

[4] Sarah Parisot, Sofia Ira Ktena, Enzo Ferrante, Matthew Lee, Ricardo Guerrero, Ben Glocker, and Daniel Rueckert. Disease Prediction using Graph Convolutional Networks: Application to Autism Spectrum Disorder and Alzheimer's Disease. 48:117–130. ISSN 13618415. doi: 10.1016/j.media.2018.06.001. URL http://arxiv.org/abs/1806.01738.

[5] Fuli Feng, Xiangnan He, Xiang Wang, Cheng Luo, Yiqun Liu, and Tat-Seng Chua. Temporal Relational Ranking for Stock Prediction. 37(2):27:1–27:30. ISSN 1046-8188. doi: 10.1145/3309547. URL http://doi.acm.org/10.1145/3309547.