Philipp Lintl, philipp.lintl@student.uva.nl, 12152498

1 Vanilla RNN vs LSTM

1.1 Vanilla RNN in Pytorch

Question 1.1

Given,

$$\begin{aligned}
\mathbf{h}^{(t)} &= \tanh\left(\mathbf{W}_{hx}\mathbf{x}^{(t)} + \mathbf{W}_{hh}\mathbf{h}^{(t-1)} + \mathbf{b}_h\right) \\
\mathbf{p}^{(t)} &= \mathbf{W}_{ph}\mathbf{h}^{(t)} + \mathbf{b}_p \\
\hat{\mathbf{y}}^{(t)} &= softmax(\mathbf{p}^{(t)}) \\
\mathcal{L} &= -\sum_{k=1}^{K} \mathrm{y}_k \log \hat{\mathrm{y}}_k
\end{aligned}$$

The loss function can be rewritten by filling in the softmax-term as such:

$$\mathcal{L}^{(t)} = -\sum_{k=1}^{K} y_k \log \hat{y}_k^{(t)} = -\sum_{k=1}^{K} y_k \log \frac{\exp p_k^{(t)}}{\sum_i^K \exp p_i^{(t)}} \tag{1}$$

$$= -\sum_{k=1}^{K} y_k \left(p_k^{(t)} - \log \sum_{i=1}^{K} \exp p_i^{(t)} \right) \tag{2}$$

Applying the chain rule to the first gradient yields:

$$\frac{\partial \mathcal{L}^{(t)}}{\partial W_{ph}} = \frac{\partial \mathcal{L}^{(t)}}{\partial \hat{y}^{(t)}} \frac{\partial \hat{y}^{(t)}}{\partial \mathbf{p}^{(t)}} \frac{\partial \mathbf{p}^{(t)}}{\partial \mathbf{W}_{ph}} \tag{3}$$

$$= \sum_{k=1}^{K} \sum_{i=1}^{M} \frac{\partial \mathcal{L}^{(t)}}{\partial \hat{y}_k^{(t)}} \frac{\partial \hat{y}_k^{(t)}}{\partial p_i^{(t)}} \frac{\partial p_i^{(t)}}{\partial \mathbf{W}_{ph}} \tag{4}$$

In this case, K=M, as 10 classes are given. Thus, in order to obtain the gradient $\frac{\partial \mathcal{L}^{(T)}}{\partial \mathbf{W}_{ph}}$, we need to look at the partial derivatives.

Knowing that

$$\hat{\mathbf{y}}^{(T)} = \frac{\exp\left(\mathbf{p}^{(T)}\right)}{\sum_j \exp\left(\mathbf{p}_j^{(T)}\right)}$$

,

means for an individual class k:

$$\hat{y}_k^{(T)} = \frac{\exp p_k^{(T)}}{\sum_j \exp p_j^{(T)}}$$

To formulate the full Loss in equation 4, we individually look at the three parts starting from the left.

$$\frac{\partial \mathcal{L}^{(t)}}{\partial \hat{y}_k^{(t)}} = \frac{\partial - \sum_{k=1}^{K} y_k \log \hat{y}_k^{(t)}}{\partial \hat{y}_k^{(t)}} = -\frac{y_k}{\hat{y}_k^{(T)}} \tag{5}$$

$$\text{For true label } y_k = 1: \frac{\partial \mathcal{L}^{(t)}}{\partial \hat{y}_k^{(t)}} = -\frac{1}{\hat{y}_k^{(T)}} \tag{6}$$

$$\text{For true label } y_k = 0: \frac{\partial \mathcal{L}^{(t)}}{\partial \hat{y}_k^{(t)}} = 0 \tag{7}$$

The second partial derivatives $\frac{\partial \hat{y}_k^{(t)}}{\partial p_i^{(t)}}$ have been derived in the first assignment by applying the quotient rule, and look like this:

$$\frac{\partial \hat{y}_k^{(T)}}{\partial p_i^{(T)}} = \frac{\partial \frac{\exp p_k^{(T)}}{\sum_j \exp p_j^{(T)}}}{\partial p_i^{(T)}} = \begin{cases} \hat{y}_k^{(T)} \cdot \left(1 - \hat{y}_k^{(T)}\right) & \text{if } i = k \\ \hat{y}_k^{(T)} \cdot -\hat{y}_i^{(T)} & \text{if } i \neq k \end{cases} \tag{8}$$

As a reminder, we obtain these by applying the quotient rule:

- For $i = k$:

$$\frac{\partial \hat{y}_k^{(T)}}{\partial p_k^{(T)}} = \frac{\partial \frac{\exp p_k^{(T)}}{\sum_j \exp p_j^{(T)}}}{\partial p_k^{(T)}} = \frac{\exp\left(p_k^{(T)}\right) \cdot \left(\sum_{j=1}^{K} \exp\left(p_j^{(T)}\right)\right) - \exp\left(p_k^{(T)}\right) \cdot \exp\left(p_k^{(T)}\right)}{\left(\sum_{j=1}^{K} \exp\left(p_j^{(T)}\right)\right)^2} \tag{9}$$

$$= \frac{\exp\left(p_k^{(T)}\right)}{\sum_{j=1}^{K} \exp\left(p_j^{(T)}\right)} \cdot \frac{\sum_{j=1}^{K} \exp\left(p_j^{(T)}\right) - \exp\left(p_k^{(T)}\right)}{\sum_{j=1}^{K} \exp\left(p_j^{(T)}\right)} \tag{10}$$

$$= \hat{y}_k^{(T)} \left(1 - \hat{y}_k^{(T)}\right) \tag{11}$$

- For $i \neq k$:

$$\frac{\partial \hat{y}_k^{(T)}}{\partial p_i^{(T)}} = \frac{\partial \frac{\exp p_k^{(T)}}{\sum_j \exp p_j^{(T)}}}{\partial p_i^{(T)}} = -\frac{\exp\left(p_k^{(T)}\right)}{\left(\sum_{j=1}^{K} \exp\left(p_j^{(T)}\right)\right)^2} \exp\left(p_i^{(T)}\right) \tag{12}$$

$$= -\frac{\exp\left(p_k^{(T)}\right)}{\sum_{j=1}^{K} \exp\left(p_j^{(T)}\right)} \frac{\exp\left(p_i^{(T)}\right)}{\sum_{j=1}^{K} \exp\left(p_j^{(T)}\right)} \tag{13}$$

$$= \hat{y}_k^{(T)} \cdot -\hat{y}_i^{(T)} \tag{14}$$

As seen in the first assignment, a convenient way to formulate this case analysis with the help of the kronecker ($\delta_{ki} = 1$ for $k = i$, 0 other) delta as such:

$$\frac{\partial \hat{y}_k^{(T)}}{\partial p_i^{(T)}} = \hat{y}_k^{(T)}(\delta_{ki} - \hat{y}_i^{(T)})$$

For the third part, we have to look closer at $\mathbf{p}^{(T)}$ and its subpart $\mathbf{p}_i^{(T)}$, which is the output for a certain class before applying the softmax. In other words, given $\mathbf{W}_{ph} \in \mathbb{R}^{p \times h}$ and $\mathbf{h}_{(T)} \in \mathbb{R}^{h \times n_c}$, where n_c is the number of classes, in this case the digits. Therefore, $p_i^{(T)} = \mathbf{W}_{ph,i} h^{(T)} + b_{p,i}$, which denotes the i-th row in $\mathbf{W}_{ph}$ and the ith element of the bias vector. As this represents one element, the respective gradient with respect to matrix $\mathbf{W}_{ph}$ will be a matrix with the same dimensionality. Accordingly, all derivatives will be zero, except for the entries in row i. The non zero entries are marked by the $\mathbf{h}^{(T)}$ vector, which given a column vector needs to be transposed to fit into the i-th row of the gradient matrix.

$$\frac{\partial p_l^{(T)}}{\partial \mathbf{W}_{ph}} = \begin{pmatrix} \mathbf{0} \\ \mathbf{h}^{(T)^T} \\ \mathbf{0} \end{pmatrix}, \tag{15}$$

with $\mathbf{h}^{(T)^T}$ being in the i-th row. Now, all parts of Equation 4 are derived individually. Now, they need to be put together:

$$\frac{\partial \mathcal{L}^{(t)}}{\partial W_{ph}} = \sum_{k=1}^{K} \sum_{i=1}^{M} \frac{\partial \mathcal{L}^{(t)}}{\partial \hat{y}_k^{(t)}} \frac{\partial \hat{y}_k^{(t)}}{\partial p_i^{(t)}} \frac{\partial p_i^{(t)}}{\partial \mathbf{W}_{ph}} \tag{16}$$

$$= \sum_{k=1}^{K} \sum_{i=1}^{M} -\frac{y_k^{(T)}}{\hat{y}_k^{(T)}} \cdot \hat{y}_k^{(T)}(\delta_{ki} - \hat{y}_i^{(T)}) \cdot \begin{pmatrix} \mathbf{0} \\ \mathbf{h}^{(T)^T} \\ \mathbf{0} \end{pmatrix} \tag{17}$$

$$= \sum_{k=1}^{K} \sum_{i=1}^{M} -y_k^{(T)} \cdot (\delta_{ki} - \hat{y}_i^{(T)}) \cdot \begin{pmatrix} \mathbf{0} \\ \mathbf{h}^{(T)^T} \\ \mathbf{0} \end{pmatrix} \tag{18}$$

Since the true label $y_k^{(T)}$ is one not encoded, let's assume it has a 1 on position l, and zeros otherwise. Therefore the sum over k reduces to only one entry, when the true label is at position l. Then, we denote another Kronecker delta δ_{li}, which is 1 if $l = i$ and 0 otherwise. Then, we get to:

$$= \sum_{i=1}^{M} (\hat{y}_i^{(T)} - \delta_{li}) \cdot \begin{pmatrix} \mathbf{0} \\ \mathbf{h}^{(T)^T} \\ \mathbf{0} \end{pmatrix} \tag{19}$$

$$\tag{20}$$

Considering the $\mathbf{h}^{(T)^T}$ is always in the i-th row and $y^{(T)}$ is a one-hot vector with a one in position l, we can reduce the last term to:

$$= \left(\hat{\mathbf{y}}^{(T)} - y^{(T)} \right) \mathbf{h}^{(T)^T} \tag{21}$$

The second gradient $\frac{\partial \mathcal{L}^{(T)}}{\partial \mathbf{W}_{hh}}$ is derived similarly. We start with applying the chain rule:

$$\frac{\partial \mathcal{L}^{(T)}}{\partial \mathbf{W}_{hh}} = \frac{\partial \mathcal{L}^{(T)}}{\partial \hat{\mathbf{y}}^{(T)}} \frac{\partial \hat{\mathbf{y}}^{(T)}}{\partial \mathbf{p}^{(T)}} \frac{\partial \mathbf{p}^{(T)}}{\partial \mathbf{h}^{(T)}} \frac{\partial \mathbf{h}^{(T)}}{\partial \mathbf{W}_{hh}} \tag{22}$$

$$\tag{23}$$

The very last term $\frac{\partial \mathbf{h}^{(T)}}{\partial \mathbf{W}_{hh}}$ is the crux of this gradient, as for the first time temporal dependencies come into play. Considering the initial definition of $\mathbf{h}^{(t)}$: $\mathbf{h}^{(t)} = \tanh\left(\mathbf{W}_{hx}\mathbf{x}^{(t)} + \mathbf{W}_{hh}\mathbf{h}^{(t-1)} + \mathbf{b}_h\right)$, we see its dependency on the previous $\mathbf{h}^{(t-1)}$. The latter again depends on $\mathbf{W}_{hh}$, which in turn depends on $t \in \{0, \dots, t-2\}$ and so on.

For the gradients, this means that we have to consider $\frac{\partial h^{(t)}}{\partial \mathbf{W}_{hh}}$ for all $t \in \{0, \dots, T\}$.

Keeping in mind that the loss function is represented as the sum over the class losses:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_{hh}} = -\sum_{k=1}^{K} \frac{\partial \mathcal{L}_k}{\partial \mathbf{W}_{hh}} \tag{24}$$

Using this recurrent property, the complete loss can be rewritten as

$$\frac{\partial \mathcal{L}^{(T)}}{\partial \mathbf{W}_{hh}} = \sum_{t=0}^{T-1} \frac{\partial \mathcal{L}^{(T)}}{\partial \hat{\mathbf{y}}^{(T)}} \frac{\partial \hat{\mathbf{y}}^{(T)}}{\partial \mathbf{p}^{(T)}} \frac{\partial \mathbf{p}^{(T)}}{\partial \mathbf{h}^{(T)}} \frac{\partial \mathbf{h}^{(T)}}{\partial \mathbf{h}^{(t)}} \frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{W}_{hh}} \tag{25}$$

Similar to the previous gradients, we look at the parts individually, whereas some have already been derived.

Therefore, the recurrence can be represented by looking at the individual gradients $\frac{\partial h^{(T)}}{\partial h^{(t)}}$ for all $t \in \{0, \dots, T-1\}$ by applying the chain rule as such:

$$\frac{\partial h^{(T)}}{\partial h^{(t)}} = \frac{\partial h^{(T)}}{\partial h^{(T-1)}} \cdot \frac{\partial h^{(T-1)}}{\partial h^{(T-2)}} \cdot \frac{\partial h^{(T-2)}}{\partial h^{(T-3)}} \cdot \ldots \cdot \frac{\partial h^{(t+1)}}{\partial h^{(t)}} \tag{26}$$

$$= \prod_{i=t+1}^{T} \frac{\partial h^{(i)}}{\partial h^{(i-1)}} \tag{27}$$

As $h^{(t)} = \tanh\left(\mathbf{W}_{hx}\mathbf{x}^{(t)} + \mathbf{W}_{hh}\mathbf{h}^{(t-1)} + \mathbf{b}_h\right)$, using the derivative of the tanh function: $\frac{\partial}{\partial x}\tanh x = 1 - \tanh^2 x$, we get

$$\frac{\partial h^{(t)}}{\partial h^{(t-1)}} = \left(1 - \tanh\left(\mathbf{W}_{hx}x^{(t)} + \mathbf{W}_{hh}h^{(t-1)} + b_h\right)^2\right)\mathbf{W}_{hh}$$

So, the entire $\frac{\partial h^{(T)}}{\partial h^{(t)}}$ can be written as:

$$\frac{\partial h^{(T)}}{\partial h^{(t)}} = \prod_{i=t+1}^{T} \frac{\partial h^{(i)}}{\partial h^{(i-1)}} = \prod_{i=t+1}^{T} \left(1 - \tanh\left(\mathbf{W}_{hx} x^{(t)} + \mathbf{W}_{hh} h^{(t-1)} + b_h\right)^2\right) \mathbf{W}_{hh} \tag{28}$$

The last term in equation 25, is similar to the previous one, except the inner term is now derived for $\mathbf{W}_{hh}$

$$\frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{W}_{hh}} = \left(1 - \tanh\left(\mathbf{W}_{hx} x^{(t)} + W_{hh} h^{(t-1)} + b_h\right)^2\right) \begin{pmatrix} h^{(t-1)T} \\ \vdots \\ h^{(t-1)T} \end{pmatrix} \tag{29}$$

For the remaining terms of equation 25, we can use already derived parts:

$$\frac{\partial \mathcal{L}^{(T)}}{\partial \hat{\mathbf{y}}^{(T)}} \frac{\partial \hat{\mathbf{y}}^{(T)}}{\partial \mathbf{p}^{(T)}} = \hat{\mathbf{y}}^{(T)} - \mathbf{y}^{(T)} \tag{30}$$

$$\frac{\partial \mathbf{p}^{(T)}}{\partial \mathbf{h}^{(T)}} = \frac{\partial \mathbf{W}_{ph} \mathbf{h}^{(t)} + \mathbf{b}_p}{\partial \mathbf{h}^{(T)}} = \mathbf{W}_{ph} \tag{31}$$

Finally, we can put all parts together for the full gradient. For the sake of oversight, I refer to the derived parts from above, which are simply multiplied according to this equation:

$$\frac{\partial \mathcal{L}^{(T)}}{\partial \mathbf{W}_{hh}} = \sum_{t=0}^{T-1} \frac{\partial \mathcal{L}^{(T)}}{\partial \hat{y}^{(T)}} \frac{\partial \hat{y}^{(T)}}{\partial \mathbf{h}^{(T)}} \frac{\partial \mathbf{h}^{(T)}}{\partial \mathbf{h}^{(t)}} \frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{W}_{hh}}$$

The first gradient $\frac{\partial \mathcal{L}^{(T)}}{\partial \mathbf{W}_{ph}}$ only depends on the last steps inputs. Therefore, it is not dependent on the previous hidden states or inputs. As seen in the formulas, $\frac{\partial \mathcal{L}^{(T)}}{\partial \mathbf{W}_{hh}}$ on the other hand fully depends on all previous steps.

For the latter, the problem of *Vanishing Gradients* is observed. When the gradients depend on all of the previous steps, as seen in equation 28, this product invokes problems. In fact, when T is set to be fairly large, due to the nature of the derivative of the tanh function problems arise. To be precise, $\frac{\partial tanh(x)}{\partial x} \in [0, 1]$, which leads to products that can get very small. As these products are again multiplied with the other partial derivatives as seen in equation 25 and then summed over all time steps this can lead to small or even almost zero gradients. Therefore, applying backpropagation will not lead to proper learning or even no learning at all.

Question 1.2

The code for this part is found in `vanilla_rnn.py` and for training the procedure in `train.py`. At first, the required weight matrices and bias vectors are initialized according to the standard normal distribution. In code, for instance, this means `nn.Parameter(torch.randn((input_dim, num_hidden))` for the weight matrix between the input and the hidden state. The hidden state is initialized empty and reset before every forward pass. In the forward pass, I basically followed the equations given in the assignment sheet. The most interesting part is the loop for each part of the sequence. Basically, one iteration of that loop takes the first element for all the batch elements, so the

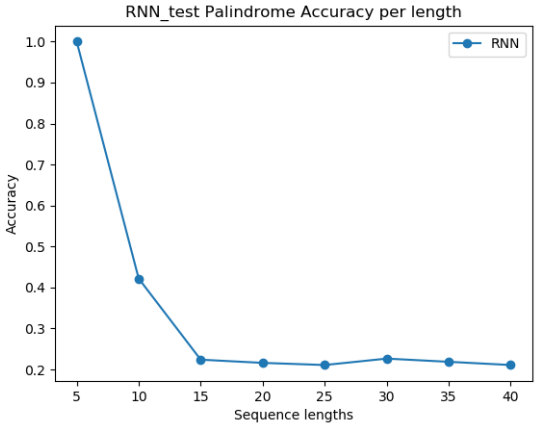

Figure 1: Mean accuracy over three differently seeded runs for sequence length 5-40 with stepsize 5.

first digit of all the 128 palindromes of the batch. These are then passed through and followed by all the other sequence steps. Training is stopped, if 50 steps in a row yield the perfect score of 1.0.

Question 1.3

For this part, I chose the sequence lengths to be from 5 to 40 with a step size of 5. Due to time limitations and to avoid unnecessary computations, for each sequence length, the RNN ran across three different seeds. In Figure 1, we observe that for input sequence length 5, in fact the perfect score of accuracy 1 is achieved. After that, performance drops to around 0.4 for length 10. For even larger inputs, the score oscilates around 0.2. We thus show that the Vanilla RNN model can not adequately handle long term dependencies. Eventhough around 20% of inputs can be predicted correctly, the majority of inputs can not be predicted.

Apparently the weight initialization heavily influences learning. Therefore, one possible way to improve performance would be to use a less strong initialization as the one I chose with standard normal distributed weights. Possibly, the initial weights could be decreased by a factor determined by the batch size or the dimension of the hidden layers. Another possible improvement could lie in the Xavier uniform initialization approach. Due to time limitations, I could not assess these claims.

Question 1.4

Vanilla Stochastic Gradient Descent algorithms suffer from several shortcomings. Firstly, using a constant learning rate can lead to getting stuck in local optima, if chosen too small. Or, if chosen too large, the steps being made can be too large to find optimal regions and therefore no learning at all. On the other hand, saddle/plateau points can be an obstacle to SGD, as the required steps to get out of such regions can not be made, as gradients bounce around instead of moving towards the optimum in a straight fashion. Also, SGD algorithms tend to be slow in training, if no adjustments are being made. Some possible adjustments are momentum and daptive learning rate.

Adaptive Learning Rate: This approach enables to have different learning rates for different weights. This is useful to overcome highly different gradients for different weights. If such highly different gradients occur while a constant learning rate is applied, learning is limited. More precisely. some weights receive too small and some too large updates. Adaptive learning rate, as applied in RMSprop enables to consider the directions, as well as magnitude of gradients for different weights. For instance, RMSprop adapts the learning rate to be larger for gradients that do not encounter sign changes in gradients too often. On the other hand, weights with gradients that are not consistent with respect to direction, are assigned a lower learning rate. These changes stabilize the training process

and lead to better optimization.

Momentum
As seen with the adaptive learning rate, it can be useful to draw on information from previous training steps. Momentum falls into that category by considering the previous gradient updates to influence the current parameter update. Similar to the adaptive learning rate, gradients that are consistent in direction and magnitude shall be assigned a higher learning rate as gradients which suffer from frequent sign changes. In Momentum, this is achieved by adding a small part of the previous gradient to the current update. As hinted before, this leads to faster learning for consistent gradients and stabilization of noisy updated. So, in general, a smoother learning curve is observed and finally decreases the risk of getting stuck in local optima by allowing larger steps than an update without momentum.

Both methods use additional information to only using the current gradients. These changes result in faster training, less noisy updates and in general better optimization.

Question 1.5

(a)

- **Input Gate** $i^{(t)}$: This gate controls the amount of new information being added to the cell state $c^{(t)}$. For every time step, information from the previous hidden state h and the current input x is processed in the input gate. By applying the sigmoid activation function on the linear combination of the two, the propagated values are pushed between 0 and 1. That way, the input gate controls how much information from the input and the previous state is propagated to the current cell state.
- **Forget gate** $f^{(t)}$: The forget gate controls the amount of past information that might be discarded. More precisely, based on its inputs, the hidden state of the previous time step and the current input, it influences the amount of information of the cell state that can be discarded. It does so by assigning each dimension of the cell state a score between zero and one, which are then multiplied to the cell state for each dimension. Thus, the score assesses how important this dimension is for the current time step. Scores close to zero indicate that the previous information can be erased, while scores close to 1 indicate that the previous information is in fact of relevancy.
- **Input Modulation Gate** $g^{(t)}$: In the Input modulation gate, possible candidate cell states to be passed on to the next hidden state are created. In it, a linear combination of the input x and the previous hidden state h are fed through the tanh function. This function introduces non-linearity by mapping the input values between -1 and 1. This mapping helps to limit the influence of possibly very large values and moreover as seen before, it reinforces strong signals and limits negative values. These values are then element wise multiplied to the input gate to then be passed on to the cell state by addition.
- **Output Gate** $o^{(t)}$: This gate draws on all the previous gates to then bundle them together to pass the information to the next time step. More precisely, it takes $h^{(t-1)}, c^{(t)}$ and $x^{(t)}$ as inputs. By applying sigmoid to the previous x and h together, feeding the result through tanh and finally multiplying that result to the current cell state $c^{(t)}$. Thus, similar to to the forget gate, multiplying a value between minus one and one controls how much information from the cell state is propagated to the next hidden state.

  Non linearities appear whenever activation functions such as the sigmoid or the hyperbolic tan h functions are applied.

  (b) Given the dimension of the features d and n the number of units in the LSTM and m the batch size. Where do trainable parameters appear? All weight matrices and bias vector of all the different gates contain those. As the weight matrices and bias vectors are shared among the time steps, we only need to consider the number of parameters per matrix. As the dimensionalities of some of those match, we can summarize matrices $N(W_{gx}) = N(W_{ix}) = N(W_{fx}) = N(W_{ox})$, whereas N() denotes the number of parameters of each matrix, namely the dimensions multiplied. All of those matrices are of dimension $d \times n$,

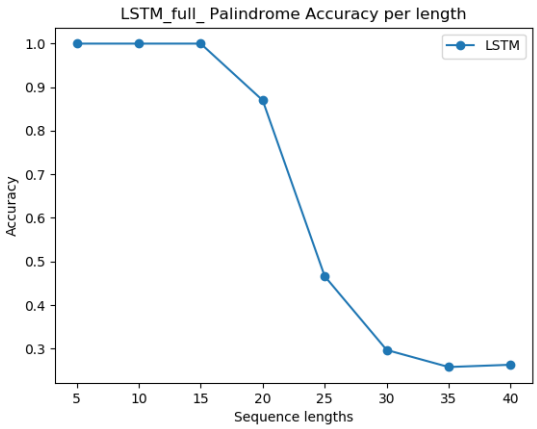

Figure 2: Mean accuracy over three differently seeded runs for sequence length 5-40 with stepsize 5.

therefore, we get $4 * d * n$. The next matrices with shared dimensionality, namely input dimension squared are $N(W_{gh}) = N(W_{ih}) = N(W_{fx}) = N(W_{oh})$, resulting in $4 * n * n$ parameters. Finally, the trainable bias vectors $N(b_g) = N(b_i) = N(b_f) = N(b_o)$ all share dimensionality n. In total, we end up with $4n^2 + 4d * n * +4n$ trainable parameters for the given LSTM model.

1.2 LSTM in Pytorch

Question 1.6

The structure of the code is analogous to the Vanilla RNN. Initialization happens with standard Gausian distribution again. The only difference in the lstm.py file is that additional weight matrices and bias vectors need to be added for the respective gates. The forward pass straight forwardly follows the equations given in the assignmnt sheet. The hidden and the cell states are reset before every forward pass. The train file is the same as before, assessing convergence, when 50 steps yield an accuracy of 1. We observe, that the performance in Figure 2 is higher for larger input sequences as for the RNN. More precisely, where even for T=10, the RNNs performance already dropped, the LSTM keeps the perfect score until length 15 and even for 20 achieves high accuracy. After that, performance drops as well. The ability to predict higher input sequences as the RNN shows the better capability to model long term dependencies by the LSTM compared to the Vanilla RNN.

One general observation from my training efforts was that the LSTM given the higher number of parameters is more dependent on the training conditions. Also, the effort to train the LSTM given the same hyperparameters as the RNN was about 2.5 higher.

Possible improvements include hyperparametertuning for instance as suggested in the assignment sheet, to adapt the learning rate for the LSTM adjusted for the input sequence length or add regularization such as momentum or dropout.

2 Graph Neural Networks

Question 3.1
(a)

As given in the assignment sheet, a forward pass in a GCN is formed by the following equation:

$$H^{(l+1)} = \sigma\left(\hat{A}H^{(l)}W^{(l)}\right)$$

Equation 16 of the assignment sheet describes the symmetric adjacency matrix, which encodes the structural relationships in the graph. More precisely, it represents the connections between nodes and weighs them according to their distance or strength. The nodes themselves are encoded by Matrix H. by adding the identity matrix to the adjacency matrix, self connections for nodes are enabled. The multiplications of matrix D normalized the adjacency matrix. In the end, the transformed adjacency matrix is symmetric. Multiplying this transformed adjacency matrix to the node encoding guarantees that only features of neighboring nodes are considered when performing a forward pass. The weight matrix W on the other hand is shared across all nodes and therefore does not take advantage of the graphs structural information. As only A and H exploit the graph structure, W shall be disregarded for the explanation of the terms.

Thus, $H^{(l+1)} = \sigma\left(\hat{A}H^{(l)}W^{(l)}\right)$ can be reduced to $H^{(l+1)} = \hat{A}H^{(l)})_{ij}$, as the non linearity σ does not involve the graph structure, as well. Considering, $H^{(l)}$ is a $N \times d$ matrix, in the l-th layer, we can rewrite the matrix multiplication in summation form, to represent the structural component more clearly: $\left(AH^{(l)}\right)_{i,j} = \sum_k \left(A_{i,k}H^{(l)}_{k,j}\right)$. As A is an adjacency matrix, it only contains 1 and 0s to denote, whether nodes are neighbors or not. If we then only consider one node i in $(AH^{(l)})_{ij}$, we see that it comprised of all nodes k that are its neighbors. So all parts of the sum not representing a neighbor of node i are $A_{jk} = 0$ and therefore not part of the sum. Thus, this term can be reduced to $(AH^{(l)})_{ij} = \sum_{k\in\delta_i} H^{(l)}_k$, where δ_i denotes all the neighbors of node i. Thus, message passing is achieved by simply summing the previous feature vectors of all the neighboring nodes of a certain node.

(b)
Imbalanced data: For imbalanced datasets, it can happen, that predictions are then skewed towards those classes being present most frequently, as they are passed through more often in the training phase. It requires calibration efforts and reweighting to unskew the distribution across large datasets.

Question 3.2
(a)
If we consider A-F from top to bottom and left to right, this is the respective adjacency matrix of the given graph.

$$\begin{bmatrix} 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 & 1 \end{bmatrix}$$

(b)
As the shortest path from C to E either goes thorugh B and A or D and F, three steps will be required. Thres steps because three hops are required in order to arrive at a node that has a path length of 2 or os the third degree neighbor.

2.1 Applications of GNNs

In order to exploit the advantages of Graph Neural Networks, we look for applications that inherently incorporate graph like structures or want to exploit distance into the modelling process.

An example for the former would be the application in the chemistry field. Molecules are comprised of atoms, which share similarities. These similarities can be represented by graphs with connections

which denote the chemical connections between similar atoms.

An example for the latter would be any kind of modelling that tries to incorporate spatio(-temporal) information. the focus here is on the spatio component. For instance, if Air pollution sensors are given with locations and the goal is to predict air pollution concentration over certain regions. It can be important to use the location information, such that sensors being located closer to each other are weighted heavier and influence each other more than distant ones.

2.2 Comparing and Combining GNNs and RNNs

Question 3.4
(a)

For data situations which incorporate any kind of sequential structure and temporal dependence, such as text and time series, RNNs are optimal due to their recurrency. This characteristic allows to model dependencies across very large input sequences, making that class of models very powerful.

GNNs on the other hand take advantage of graph structures. As hinted above, such structures include spatio-dependencies, molecular dependencies or even the modelling of text. As text can also be processed not as a sequence of characters or words, but according to an underlying structure based on syntax, possibly leading to tree like structures.

Considering, RNNs can be represented as GNNs, there has to be an overlap in applications. So, RNNs are basically graphs which dont allow cicles and are directed, as input sequences are modelled in a certain order. Considering this loss of generability and therefore information, as undirected edges are more expressive than directed ones, GNNs seem to be more expressive in general. This comes in handy, when data structures that explicitely incorporate such complex patterns that are not sequential, such as relations in social networks or spatio-dependencies in geo-data. In that case, GNNs will represent more of the underlying patterns simply as the architecture considers these, whereas RNNs will struggle to cover such patterns.

(b)
Title: **Exploiting Spatiotemporal Patterns for Accurate Air Quality Forecasting using Deep Learning**: As hinted above, for some data situations it is important to model spatio-temporal dependencies. For air pollution data, the temporal aspect, as well as the locational aspect influence air pollution concentrations. One possible approach to model this, is for once to use a RNN like architecture to model the measurements of pollution sensors, as these measurements highly depend on previous values. However, if predictions want to be done not only on a sensor level, but actually to predict the pollution developments across time, the locations somehow neeed to be incorporated. In this paper, they construct a graph based on the similarity of the sensor stations. Similarity is based on the surroundings of each station. So, for isntance, ideally all stations based in industrial areas should be connected closely in the graph structure. Then passing the messages based on the similarity of the stations helps to cover pattens across similar stations to improve predictions.