Assignment 2. Recurrent Neural Networks and Graph Neural Networks

David Biertimpel
david.biertimpel@student.uva.nl
uva-id:12324418

1 Vanilla RNN versus LSTM

Note: I follow the convention that derivatives are row vectors.

1.1 Toy Problem: Palindrome Numbers

1.2 Vanilla RNN in PyTorch

Question 1.1

First, it is instructive to write out all the partial derivatives by applying the chain rule:

$$\frac{\partial \mathcal{L}^{(T)}}{\partial \boldsymbol{W}_{ph}} = \frac{\partial \mathcal{L}^{(T)}}{\partial \hat{\boldsymbol{y}}^{(T)}} \frac{\partial \hat{\boldsymbol{y}}^{(T)}}{\partial \boldsymbol{p}^{(T)}} \frac{\partial \boldsymbol{p}^{(T)}}{\partial \boldsymbol{W}_{ph}}$$

From the first assignment (exercise 1.1 a)) we already know the derivative of the cross-entropy loss:

$$\frac{\partial \mathcal{L}^{(T)}}{\partial \hat{\boldsymbol{y}}^{(T)}} = -\frac{\boldsymbol{y}}{\hat{\boldsymbol{y}}^{(T)}}$$

And the derivative of the softmax:

$$\frac{\partial \hat{\boldsymbol{y}}^{(T)}}{\partial \boldsymbol{p}^{(T)}} = \text{diag}\big(\hat{\boldsymbol{y}}^{(T)}\big) - \hat{\boldsymbol{y}}^{(T)} \hat{\boldsymbol{y}}^{(T)T}$$

Finally, we can compute the last derivative, which we effectively also computed in the first assignment with $\frac{\partial \tilde{x}^{(l)}}{\partial W^{(l)}}$:

$$\begin{aligned}
\frac{\partial \boldsymbol{p}^{(T)}}{\partial \boldsymbol{W}_{ph}} &= \frac{\partial}{\partial \boldsymbol{W}_{ph}} \left(\boldsymbol{W}_{ph} \boldsymbol{h}^{(T)} + \boldsymbol{b}_p \right) \\
&= \begin{bmatrix} \frac{\partial \boldsymbol{p}_1^{(T)}}{\partial \boldsymbol{W}_{ph}} \\ \vdots \\ \frac{\partial \boldsymbol{p}_i^{(T)}}{\partial \boldsymbol{W}_{ph}} \\ \vdots \\ \frac{\partial \boldsymbol{p}_D^{(T)}}{\partial \boldsymbol{W}_{ph}} \end{bmatrix}, \text{ where } \frac{\partial \boldsymbol{p}_i^{(T)}}{\partial \boldsymbol{W}_{ph}} = \begin{bmatrix} \boldsymbol{0}^T \\ \vdots \\ \boldsymbol{h}^{(T)T} \\ \vdots \\ \boldsymbol{0}^T \end{bmatrix} \\
\text{since, } \frac{\partial \boldsymbol{p}_i^{(T)}}{\partial \boldsymbol{W}_{ph;i,:}} &= \boldsymbol{h}^{(T)T} \quad \text{and} \quad \frac{\partial \boldsymbol{p}_i^{(T)}}{\partial \boldsymbol{W}_{ph;j \neq i,:}} = 0^T
\end{aligned}$$

Now we can combine all three derivatives and simplify:

$$\frac{\partial \mathcal{L}^{(T)}}{\partial \boldsymbol{W}_{ph}} = -\frac{\boldsymbol{y}}{\hat{\boldsymbol{y}}^{(T)}}\text{diag}\big(\hat{\boldsymbol{y}}^{(T)}\big) - \hat{\boldsymbol{y}}^{(T)}\hat{\boldsymbol{y}}^{(T)T}\begin{bmatrix} \frac{\partial \boldsymbol{p}_1^{(T)}}{\partial \boldsymbol{W}_{ph}} \\ \vdots \\ \frac{\partial \boldsymbol{p}_i^{(T)}}{\partial \boldsymbol{W}_{ph}} \\ \vdots \\ \frac{\partial \boldsymbol{p}_D^{(T)}}{\partial \boldsymbol{W}_{ph}} \end{bmatrix} = \Big(-\boldsymbol{y}^T + \boldsymbol{y}^T\text{diag}\big(\hat{\boldsymbol{y}}^{(T)}\big)\Big)\begin{bmatrix} \frac{\partial \boldsymbol{p}_1^{(T)}}{\partial \boldsymbol{W}_{ph}} \\ \vdots \\ \frac{\partial \boldsymbol{p}_i^{(T)}}{\partial \boldsymbol{W}_{ph}} \\ \vdots \\ \frac{\partial \boldsymbol{p}_D^{(T)}}{\partial \boldsymbol{W}_{ph}} \end{bmatrix}$$

$$= \big(\hat{\boldsymbol{y}}^{(T)} - \boldsymbol{y}\big)\boldsymbol{h}^{(T)T}$$

For the second derivative $\frac{\partial \mathcal{L}^{(T)}}{\partial \boldsymbol{W}_{hh}}$ we also first apply the chain rule to visualize all necessary partial derivatives:

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

From the previous exercise we already calculated and simplified $\frac{\partial \mathcal{L}^{(T)}}{\partial \hat{\boldsymbol{y}}^{(T)}}\frac{\partial \hat{\boldsymbol{y}}^{(T)}}{\partial \boldsymbol{p}^{(T)}} = \big(\hat{\boldsymbol{y}}^{(T)} - \boldsymbol{y}\big)$, that's why we can right away concentrate on $\frac{\partial \boldsymbol{p}^{(T)}}{\partial \boldsymbol{h}^{(T)}}$:

$$\frac{\partial \boldsymbol{p}^{(T)}}{\partial \boldsymbol{h}^{(T)}} = \frac{\partial}{\partial \boldsymbol{h}^{(T)}}\Big(\boldsymbol{W}_{ph}\boldsymbol{h}^{(T)} + \boldsymbol{b}_p\Big) = \boldsymbol{W}_{ph}$$

For the derivative $\frac{\partial \boldsymbol{h}^{(T)}}{\partial \boldsymbol{W}_{hh}}$ we first need to calculate the derivative of tanh:

$$\frac{\partial \tanh(x)}{\partial x} = \frac{\partial}{\partial x}\left(\frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)}\right)$$

$$= \frac{\big(\exp(x) + \exp(-x)\big)\big(\exp(x) + \exp(-x)\big) - \big(\exp(x) - \exp(-x)\big)\big(\exp(x) - \exp(-x)\big)}{\big(\exp(x) + \exp(-x)\big)^2}$$

$$= 1 - \frac{\big(\exp(x) - \exp(-x)\big)^2}{\big(\exp(x) + \exp(-x)\big)^2} = 1 - \tanh^2(x)$$

Now we can finally proceed with:

$$\frac{\partial \boldsymbol{h}^{(T)}}{\partial \boldsymbol{W}_{hh}} = \frac{\partial}{\partial \boldsymbol{W}_{hh}}\Big(\tanh\Big(\boldsymbol{W}_{hx}\boldsymbol{x}^{(T)} + \boldsymbol{W}_{hh}\boldsymbol{h}^{(T-1)} + \boldsymbol{b}_h\Big)\Big)$$

$$= \Big(\boldsymbol{I}_D - \text{diag}\big(\boldsymbol{h}^{(T)^2}\big)\Big)\frac{\partial}{\partial \boldsymbol{W}_{hh}}\Big(\boldsymbol{W}_{hx}\boldsymbol{x}^{(T)} + \boldsymbol{W}_{hh}\boldsymbol{h}^{(T-1)} + \boldsymbol{b}_h\Big)$$

$$= \Big(\boldsymbol{I}_D - \text{diag}\big(\boldsymbol{h}^{(T)^2}\big)\Big)\frac{\partial}{\partial \boldsymbol{W}_{hh}}\Big(\boldsymbol{W}_{hh}\boldsymbol{h}^{(T-1)}\Big)$$

$$= \Big(\boldsymbol{I}_D - \text{diag}\big(\boldsymbol{h}^{(T)^2}\big)\Big)\Big(\boldsymbol{h}^{(T-1)}\frac{\partial}{\partial \boldsymbol{W}_{hh}}\Big(\boldsymbol{W}_{hh}\Big) + \boldsymbol{W}_{hh}\frac{\partial}{\partial \boldsymbol{W}_{hh}}\Big(\boldsymbol{h}^{(T-1)}\Big)\Big)$$

$$= \Big(\boldsymbol{I}_D - \text{diag}\big(\boldsymbol{h}^{(T)^2}\big)\Big)\Big(\boldsymbol{h}^{(T-1)} + \boldsymbol{W}_{hh}\frac{\partial \boldsymbol{h}^{(T-1)}}{\partial \boldsymbol{W}_{hh}}\Big)$$

Combining all the partial derivatives we get:

$$\frac{\partial \mathcal{L}^{(T)}}{\partial \boldsymbol{W}_{hh}} = \big(\hat{\boldsymbol{y}}^{(T)} - \boldsymbol{y}\big)\boldsymbol{W}_{ph}\Big(\boldsymbol{I}_D - \text{diag}\big(\boldsymbol{h}^{(T)^2}\big)\Big)\Big(\boldsymbol{h}^{(T-1)} + \boldsymbol{W}_{hh}\frac{\partial \boldsymbol{h}^{(T-1)}}{\partial \boldsymbol{W}_{hh}}\Big)$$

When considering $\frac{\partial \mathcal{L}^{(T)}}{\partial \boldsymbol{W}_{hh}}$ we observe a recursive relationship in $\frac{\partial \boldsymbol{h}^{(T)}}{\partial \boldsymbol{W}_{hh}}$ as we can recursively step into $\frac{\partial \boldsymbol{h}^{(T-1)}}{\partial \boldsymbol{W}_{hh}}$ until we reach the beginning of the sequence at $\boldsymbol{h}^{(0)}$. This introduces temporal dependencies in $\frac{\partial \mathcal{L}^{(T)}}{\partial \boldsymbol{W}_{hh}}$. By contrast, this is not the case in the first derivative $\frac{\partial \mathcal{L}^{(T)}}{\partial \boldsymbol{W}_{ph}}$ where we only focus on the weights facilitating the output ($\boldsymbol{W}_{ph}$), and consequently do not leave the respective time step.

Again looking at $\frac{\partial \mathcal{L}^{(T)}}{\partial \boldsymbol{W}_{hh}}$ we see that by recursively applying $\frac{\partial \boldsymbol{h}^{(T-1)}}{\partial \boldsymbol{W}_{hh}}$ in $\frac{\partial \boldsymbol{h}^{(T)}}{\partial \boldsymbol{W}_{hh}}$ we get a chain of T multiplications. So if the number of time steps T becomes large, depending on the size of the gradients, we risk to get either vanishing or exploding gradients, because values < 1 approach zero and values > 1 explode exponentially fast. In the case of exploding gradients we can mitigate these problems as we can rescale them to a smaller size and still obtain good gradients. In the case of vanishing gradients we cannot simply rescale as the direction is possibly noisy due to numerical instabilities. Vanishing gradients are particularly problematic, since the updates for longer time dependencies become exponentially smaller and the recent time steps outweigh the more distant ones. As a result, the model rather learns short-term than long-term dependencies.

Question 1.2

See code in the Python files `vanilla_rnn.py` and `train.py` in the folder `part1`. Also note that I used the file `start1.py` in the `assignment_2` folder to run the experiments.

Question 1.3

After implementing the RNN we assess its memorization capability by predicting the last digit of a given palindrome across different palindrome lengths. We start with palindromes of length 5 and gradually increase the sequence length until we reach 150 (with bigger jumps in the end).

We train each RNN for maximal $10,000$ steps but have different minimal number of steps for increasing palindrome lengths. This is because RNNs tend to converge slower for longer palindrome lengths and often stagnate in the beginning of the training before starting to improve. With setting a minimal number of steps we prevent premature convergence (see Table 1). We say that we converged

Minimal steps for vanilla RNNs

$l \leq 15$	$15 < l < 30$	$30 < l$
1000	3000	6500

Table 1: Minimal number of step across different palindrome lengths (l).

if the minimum number of steps is reached and the absolute value of the difference between the current loss and the average loss over the last 100 steps is smaller than $\epsilon = 5 \cdot 10^{-4}$.

For optimization we use RMSprop, where we use a learning rate of 0.001 for palindrome lengths 5, 10 and 15 and 0.001 for everything longer than that. A complete overview of the hyperparameters can be found in Table 2.

Hyperparameters

max. train steps	batch size	optimizer	learning rates	input dim.	hidden dim.
10,000	128	RMSprop	1e-{3, 4}	10	128

Table 2: Hyperparameters of the RNN.

To make the different digits in the palindrome independent of each other and to ensure stable norms we convert each digit to a one-hot vector of size 10, where for digit i the vector has a one at index i and otherwise contains only zeros. This considerably improves and speeds up the convergence time of the RNN.

In order to obtain stable results we train each model with 4 different random seeds and report the mean and standard deviation of the accuracy. The accuracy is obtained by testing the trained model on 8192 new generated test observations. The corresponding results are visualized in Figure 1. Here we see that the RNN achieves perfect performance for palindrome lengths up to 50 and considerably

worse results for palindrome lengths 50 and 100 (further analysis and a comparison to the LSTM is done in section 1.3 question 1.6).

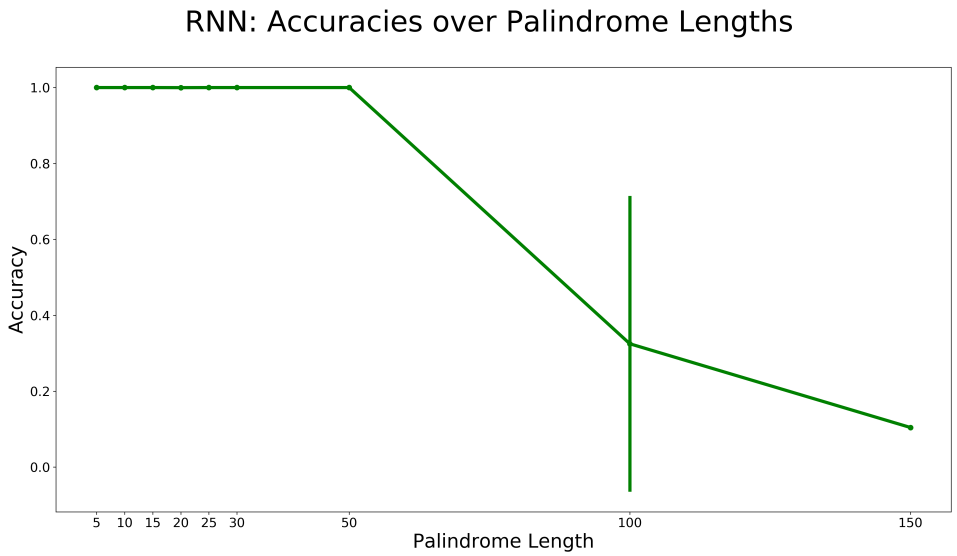


Figure 1: Accuracies across different palindrome lengths achieved with the Vanilla RNN. The vertical lines denote the standard deviation (almost zero if vertical line is not visible).

Question 1.4

Mini-batch stochastic gradient descent (we abbreviate it here with SGD) struggles for example in situations when the optimization landscape is either a plateau or a ravine. The problem with the first case is pretty obvious, since we are on a plateau all gradients are close to zero and learning is consequently very slowly or even non-existent. In the latter case, when SGD is on the wall of a ravine, the problem is that the gradient into the ravine is much larger than the gradient further along the wall. Since SGD blindly updates its parameters w.r.t. the size of the gradients, SGD ends up bouncing from wall to wall, slowly making its way forward. Since such ravines are common in the proximity of local minima, it is highly beneficial to improve the behavior of the optimizer in these situations, as this naturally leads to faster and more stable convergence. In order to tackle both the plateau and ravine problem, various approaches have been introduced in recent years. Two of them are RMSprop and Adam, but first it makes sense to briefly introduce the concept of momentum.

Momentum mitigates sudden changes in gradient direction by carrying an exponential average of previous gradients ($u_t \propto \gamma^0 g_t + \gamma^1 g_{t-1} + \gamma^2 g_{t-2} \dots$). If a change of direction takes place, it is first counterbalanced by the stored gradients, until after some time steps the new direction prevails due to the exponential decay of the older gradients. This dampens the oscillations of SGD in a ravine since the gradients in contrary directions cancel out. Metaphorically speaking, the optimizer behaves similar to a sphere with corresponding physical inertia, rolling down a hilly (loss) landscape.

RMSprop also uses exponential averaging but on the squared gradients g^2, with the following updates:

$$
\begin{aligned}
r_t &= \alpha r_{t-1} + (1-\alpha) g_t^2 \\
u_t &= -\frac{\eta}{\sqrt{r_t} + \epsilon} g_t \\
w_{t+1} &= w_t + u_t
\end{aligned}
$$

We see that in the second equation we divide our learning rate η by $\sqrt{r_t}$, which results in a smaller learning rate for larger gradients and vice versa a larger learning rate for smaller gradients. This tackles SGD's first mentioned disadvantage, namely reaching a plateau, since the small gradients are scaled up in such a way that it is possible to escape this situation. Further, RMSprop introduces a learning rate for each model parameter which are all non time-dependent as during optimization the learning rates are implicitly adapted (as just explained). Note that with this adaptive learning rate

RMSprop can scale gradients individually and separately control for unreasonable large gradients that run the risk to jump uncontrolled in the loss landscape and small gradients that prevent learning.

Finally, Adam combines RMSprop with momentum as it both carries an exponential average of the past gradients g_t and squared gradients g_t^2:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

where m_t and v_t are estimates of the first and second moment of the gradients respectively. Since we initialize m_0 and v_0 with zero we inherently introduce bias into the model. To compensate for that the following correction terms are introduced rescaling the gradients:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

Finally, the parameters are updated similar to RMSprop:

$$u_t = -\frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$$
$$w_{t+1} = w_t + u_t$$

thus we see that also Adam contains non time-dependent learning rates for each parameter which is implicitly adapted w.r.t. to the gradients.

1.3 Long-Short Term Network (LSTM) in PyTorch

Question 1.5 a)

- **The forget gate** $\left[\boldsymbol{f}^{(t)}\right]$ decides which information we *forget/discard* from the previous cell state $\boldsymbol{c}^{(t-1)}$. It takes both the previous hidden state $\boldsymbol{h}^{(t-1)}$ and the current input $\boldsymbol{x}^{(t)}$ and transform it by means of its parameters $\boldsymbol{W}_{fx}$, $\boldsymbol{W}_{fh}$ and $\boldsymbol{b}_f$. The *sigmoid* activation function then maps these values in the range $[0, 1]$. The resulting vector $\boldsymbol{f}^{(t)}$ is then element-wise multiplied by the previous cell state $\boldsymbol{c}^{(t-1)}$. *Sigmoid* is a good choice here, as values in $\boldsymbol{f}^{(t)}$ near 0 means that the corresponding value in $\boldsymbol{c}^{(t-1)}$ will effectively be forgotten, whereas values near 1 means that the information will be kept. In addition, values between 0 and 1 determine in detail how much information should be kept. So $\boldsymbol{f}^{(t)}$ can effectively regulate the information content of $\boldsymbol{c}^{(t-1)}$.

- **The input gate** $\left[\boldsymbol{i}^{(t)}\right]$ determines which information (values) in the cell state will be updated. For this the input gate uses the same structure as the forget gate (with its own parameters $\boldsymbol{W}_{ix}$, $\boldsymbol{W}_{ih}$ and $\boldsymbol{b}_i$) which is why it also follows the same logic. After the transformation by the parameters the *sigmoid* activation is applied creating the vector $\boldsymbol{i}^{(t)}$ with values in range $[0, 1]$ determining to which degree the cell state values should be updated. Then $\boldsymbol{i}^{(t)}$ is element-wise multiplied with the vetor of candidate cell state values $\boldsymbol{g}^{(t)}$.

- **The input modulation gate** $\left[\boldsymbol{g}^{(t)}\right]$ produces a *candidate* cell state for the new cell state vector $\boldsymbol{c}^{(t)}$. It follows the same structure of the gates before (parameters: $\boldsymbol{W}_{gx}$, $\boldsymbol{W}_{gh}$ and $\boldsymbol{b}_g$)) with the exception that it uses the *tanh* activation function instead of *sigmoid*. The reason behind this is that it should be possible for values in the cell state to be positive and negative. *Tanh* enables this as it has a range of $[-1, 1]$.

- **The output gate** $\left[\boldsymbol{o}^{(t)}\right]$ decides which parts of the newly updated cell state $\boldsymbol{c}^{(t)}$ will be included in the hidden state $\boldsymbol{h}^{(t)}$ for the next time step. This follows the same logic as the input and forget gates before and $\boldsymbol{o}^{(t)}$ consequently consists of transformation parameters ($\boldsymbol{W}_{ox}$, $\boldsymbol{W}_{oh}$ and $\boldsymbol{b}_o$) and a *sigmoid* activation function mapping the values in the range $[0, 1]$. The closer the values in $\boldsymbol{o}^{(t)}$ are to 1 the more of the corresponding dimension in $\boldsymbol{c}^{(t)}$ will be part of $\boldsymbol{h}^{(t)}$ and therefore issued to the next time step.

Question 1.5 b)

In the LSTM cell we have four distinct gating mechanisms with two weight matrices and a bias vector each. The sizes are as follows:

$$\boldsymbol{W}_x \in \mathbb{R}^{d \times n}, \quad \boldsymbol{W}_h \in \mathbb{R}^{n \times n}, \quad \boldsymbol{b} \in \mathbb{R}^{n \times 1}$$

If we want to consider all trainable parameters than it makes sense to also consider the output projection:

$$\boldsymbol{W}_p \in \mathbb{R}^{n \times c}, \quad \boldsymbol{b}_p \in \mathbb{R}^{c \times 1}$$

where c denotes the number of classes. With the above in mind, without considering the output projection we count:

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

trainable parameters. With the output projection we count:

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

trainable parameters. Note that both batch size and sequence length (parameters are shared) are irrelevant for this calculation.

Question 1.6)

Now we focus on implementing the LSTM and training it on the same palindrome prediction tasks as we already did for the vanilla RNN. The code for this can be observed in the Python files `lstm.py` and `train.py` in the folder `part1`.

Since the RNN and LSTM use the same training pipeline the entire outline described in section 1.2 Question 1.3 stays the same. This also includes the hyperparameter so they can be observed in Table 2. The allocation of the learning rates w.r.t. the palindrome length is also not changed. Therefore we will mostly focus on the differences to the previous experiment.

First, we observed that the LSTM converges considerably slower that the RNN. To ensure proper convergence, we have therefore increased some of the minimum number steps for which the model is trained (see Table 3).

Minimal Steps for LSTMs

$l \leq 15$	$15 < l < 30$	$30 < l$
1500	5000	6500

Table 3: Minimal number of step across different palindrome lengths (l).

Further, it is important to note that we initialized the bias of the forget gate with a vector of ones instead of zeros to prevent the problems with modelling long term dependencies as suggested by Jozefowicz et. al. [1]. This enabled the LSTM to learn palindrome lengths longer than 25 digits which it was not able to do before.

As before, the performance of the LSTM is measured in mean-accuracy across different palindrome lengths and can be observed in Figure 2. Now we will compare these results with the results achieved with the vanilla RNN shown in Figure 1.

Looking at the two diagrams it becomes obvious that the LSTM does a considerable better job in learning long-term dependencies. While the vanilla RNN achieves 0.3250 and 0.1044 accuracy for palindrome lengths 50 and 100 respectively, for the LSTM we observe accuracies of 0.7743 and 0.3278. For the palindrome lengths 5 to 50 the LSTM performs similar to the vanilla RNN as it also produces (almost) perfect results.

In terms of the standard deviation (described by the vertical lines) we see almost no fluctuation in the case of palindrome lengths 5 to 50 as both models produce accuracies very near 1.0. For palindrome length 100 the accuracy fluctuates considerably across different random seeds which suggests that the finding of a good local minima for longer sequences is harder and dependent on a favourable initialization. For the palindrome length 150 we only observe fluctuations for the LSTM,

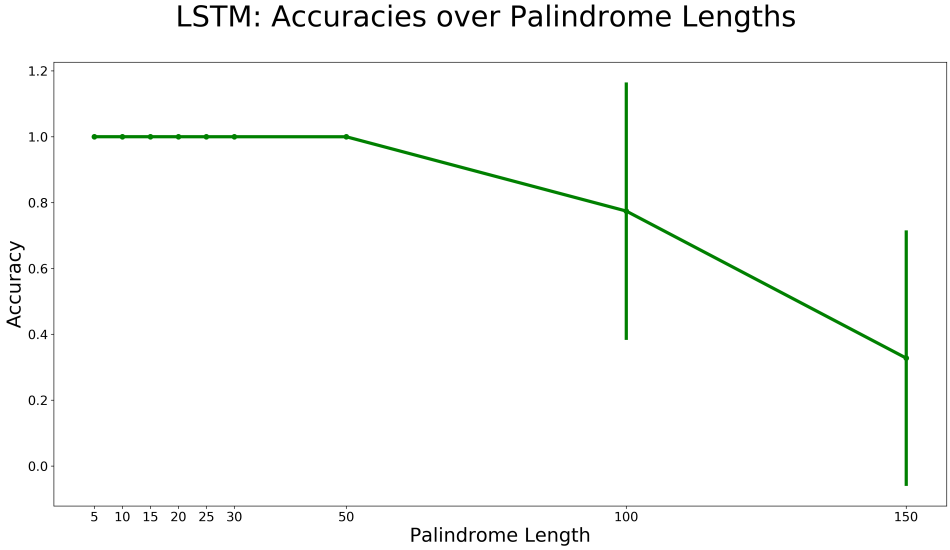


Figure 2: Accuracies across different palindrome lengths achieved with the LSTM. Again, the vertical lines denote the standard deviation, which is (almost) zero if no vertical line is visible.

by contrast the RNN produces a steady accuracy of ≈ 0.10. This suggests that the RNN is not able to capture these long-term dependencies at all since an accuracy of 0.10 with 10 classes reflects random guessing.

The major reason for the difference in behavior between both models should be the vanishing gradient problem of the vanilla RNN. This leads to the fact that the updates of longer time steps become exponentially smaller and the RNN weights concentrate on the modeling of short-term dependencies. The LSTM prevents this by the update of the cell state $\boldsymbol{c}^{(t)} = \boldsymbol{g}^{(t)} \cdot \boldsymbol{i}^{(i)} + \boldsymbol{c}^{(t-1)} \cdot \boldsymbol{f}^{(t)}$ which results in $\frac{\partial \boldsymbol{c}^{(t)}}{\partial \boldsymbol{c}^{(t-1)}}$ becoming a scaled identity function leading to stable gradients.

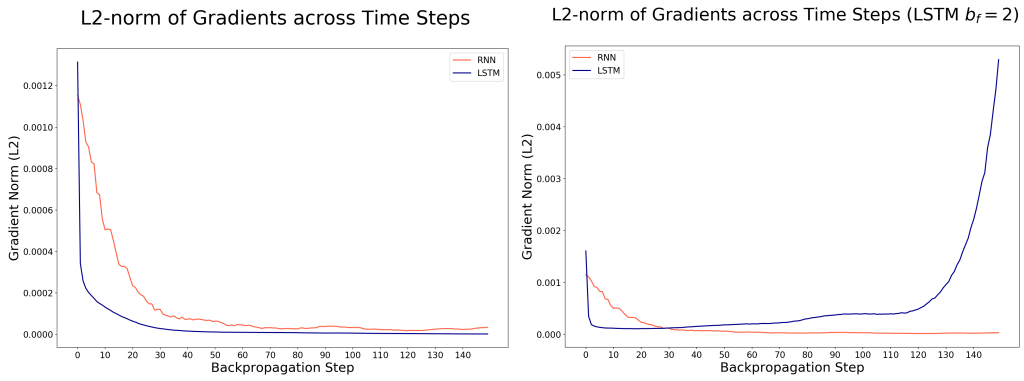
Question 1.7)

For this task we need to compute the gradient of non-leaf nodes in the automatic differentiation graph that pytorch builds up during training. In order to achieve this we call the `retain_grad()` function of the hidden states $\boldsymbol{h}^{(t)}$ during the forward pass, such that the gradient is not thrown away.

After the backward pass we go sequentially through all saved $\boldsymbol{h}^{(t)} \in \mathbb{R}^{D_h \times 1}$ and get its corresponding gradient. Since we forward pass a batch of palindromes of size B, we get in each time step t the gradient $\frac{\partial \mathcal{L}}{\partial \boldsymbol{h}^{(t)}} \in \mathbb{R}^{B \times D_h}$. We use this to get a more stable estimate of the gradient and average over the batch dimension. Afterwards, we calculate the $l2$-norm of the resulting $\frac{\partial \mathcal{L}}{\partial \boldsymbol{h}^{(t)}} \in \mathbb{R}^{1 \times D_h}$ vector. We plot each of these gradients w.r.t. their position in the backward pass, meaning the first gradient $\frac{\partial \mathcal{L}}{\partial \boldsymbol{h}^{(T)}}$ corresponds to backpropagation step 0 and the last gradient at the end of the backpropagation chain $\frac{\partial \mathcal{L}}{\partial \boldsymbol{h}^{(0)}}$ to the last backpropagation step. In Figure 3a we visualize these gradients for both the vanilla RNN and LSTM.

Here we see that the gradient magnitudes of the vanilla RNN and LSTM both exponentially decrease and then remain flat. While the LSTM gradients starts higher than those of the RNN, the RNN gradients quickly overtake the LSTM ones and remain on top for the rest of the backward pass. This is first counter intuitive as I expected the LSTM gradients to be more stable and having a larger magnitude than the gradients of the RNN. This would make sense as the last exercises clearly showed that the LSTM does a better job in handling long term dependencies.

However, we know that the vanishing gradient problem is solved with the behavior of the gradient $\frac{\partial \boldsymbol{c}^{(t)}}{\partial \boldsymbol{c}^{(t-1)}}$, where the forget gate plays a important role as it remains in the gradient as a constant. We also know that the initialization of the forget bias $\boldsymbol{b}_f$ (which we initialized to one) is of great importance for capturing long term dependencies. Now in order to show that the LSTM is capable



(a) (b)

Figure 3: Gradient magnitudes of $\frac{\partial \mathcal{L}}{\partial \boldsymbol{h}^{(t)}}$ for RNN and LSTM during the backward pass.

of a different behavior due to its gates, we initialize the $\boldsymbol{b}_f$ to a vector of twos and run the same experiment again (see Figure 3b). The behavior shown there underlines the capabilities of the LSTM to recover from small gradients as the gradient magnitudes increase again. The exploding gradient in the end can be fixed through clipping. After observing this behavior we can again make a connection to the superior LSTM performance of the last exercises. We also can emphasize the importance of the forget gate and a good initialization of $\boldsymbol{b}_f$.

Given this, I would assume that during training the forget gate got updated such that it enabled better gradients in the long run.

The corresponding code for this exercise can be observed in `grads_over_time.py` and in the `analyze_hs_gradients()` functions in textttvanilla_rnn.py and `lstm.py` in the folder `part1`.

2 Recurrent Nets as Generative Model

Question 2.1)

Note that I used the file `start2.py` in the `assignment_2` folder to run the experiments.

Question 2.1 (a)

For this exercise we train the *two-layer* LSTM model on the book *Democracy In America* by Alexis de Toqueville [2] which was already provided in the assets folder. This time no convergence criterion is defined and the model is trained over the full training steps. In terms of the hyperparameters I mostly trusted the default values as a comprehensive hyperparameter-tuning would not be feasible anyway. In contrast to the last exercise we use Adam for optimization as it combines the advantages of RMSprop and momentum which usually yields a better performance. An overview of the hyperparameters is given in Table 4.

Hyperparameters

train steps	batch size	optimizer	learning rate	lr decay	dropout %	input dim.	hidden dim.
1 Million	64	Adam	1e-3	0.96	0.15	86	128

Table 4: Hyperparameters of the LSTM for the text generation task.

Here it is interesting to note that the dimensionality of the input is defined by the size of the vocabulary. Further, the learning rate decay parameterizes a scheduler which decreases the learning rate by the factor 0.96 every 5000 steps. Besides the hyperparameters, we transform the input into a trainable embedding space by using `nn.Embedding` to enable the learning of a possibly more meaningful representation of the characters than with a one-hot vector.

We then train the *two-layer* LSTM with the described hyperparameters with an input sequence of length $T = 30$. At the end of the one million steps the model achieved a final accuracy of 0.68 and a final loss of 1.058. The accuracies and losses obtained during training are shown in Figure 4.

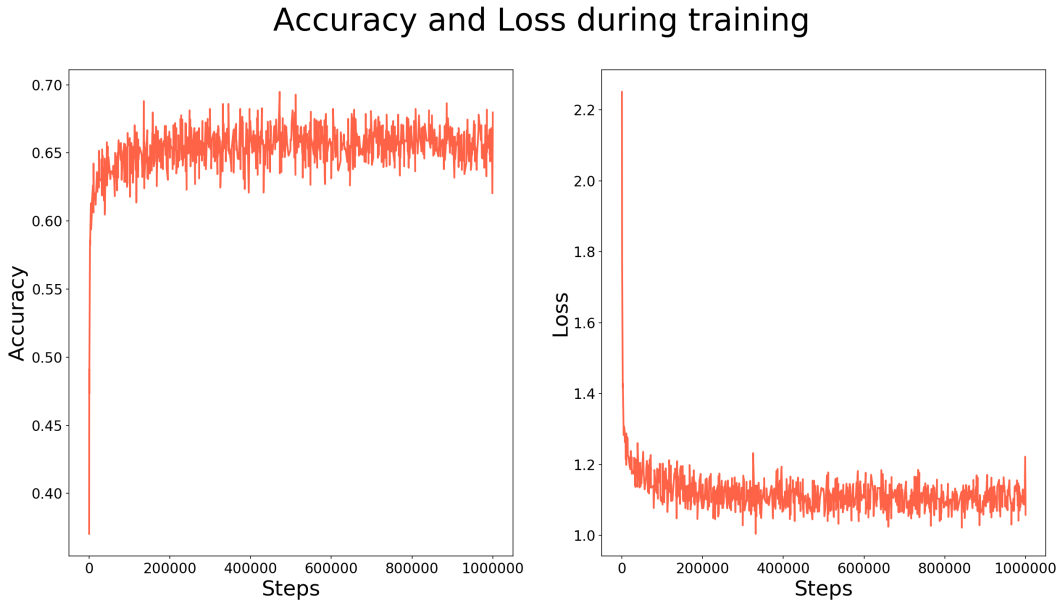


Figure 4: Accuracy and loss of the *two-layer* LSTM during training with an input sequence of length $T = 30$.

The code can be found in the `train.py` and `model.py` Python files in the folder `part2`.

Question 2.1 (b)

In order to properly capture the network's progress during training, we generate new character sequences every 100 steps during training by using the `sample_from_model()` function. All these text samples can be found in the `samples_over_training.txt` file and a selection of 6 samples is shown in the Tables 5, 6 and 7. For this exercise, we will focus on the Table's greedy sampling sections, the temperature sampling sections will be interesting for next the exercise.

We start by looking at the samples generated in iteration 0, where we observe the samples to be random. That some characters are preferred over others (e.g. 'G', '?', '.' clearly dominate) is due to the greedy sampling exploiting the networks current best guess.

In iteration 300 we see that the network already learned to use spaces to separate words. This makes sense as in the English language the space is the most common character in each text. We also observe that the model learned to spell common, shorter words like 'the' and 'of'. Longer words like 'reconcer' or 'prouse' don't make sense yet.

This picture changes when looking at the sequences generated in iteration 1000, where more complex words like 'properity', 'providual', 'comment' are spelled. It is also noteworthy that even constructs consisting of two words like 'United States' are correctly spelled. Further the '30, the' piece suggests that the model stated learning to use punctuation. Iteration 5000 behaves similar, however, we can see a bit more coherence in the sequences. Also we have another evidence of used punctuation as the Oxford comma is used in ', and' in the sequence of length 50.

When looking at iteration $300,000$ it is very interesting to observe that the model also captures complex structures like citations and footnotes. Finally, in iteration $1,000,000$ we observe quite coherent sequences especially when looking at the sequence of length 100. Not only that correct comma notation is used but you could also more or less fluently follow the content. However, overall we observe that the model overfitted on some substructures. For example, the model likes to combine

sentence parts with the word 'the' and does not seem to know any other connectors. Lastly, a general (and obvious) observation is that with a higher T you enable the model to produce longer and more coherent sequences which will otherwise get cut off if T is smaller.

Iteration 0	T	Text samples	**Iteration 300**	T	Text samples
Greedy sampling	15	!?a????2.???.??	Greedy sampling	15	! the porest th
	30	G..GG?.?.??.GG...???...????..?		30	6 the precent the preced the r
	50	@GG?aa????..??????...??.??. GGG...????.?.????.?.?.?		50	K the prect of the reand the proces the inderestit
	100	jG..GG..??.?.?.?.?????..????G. G??.????)Ixx???Ix???Ixx??I.? ?.??..????.???.?.GGG...??.??.? ????x..?????		100	; the recort the conderest of the reconcer of the reand the recertion the porest the prouse the proc
Temperature sampling			Temperature sampling		
$\tau = 0.5$	15	WYBFZde H:7EdHK	$\tau = 0.5$	15	@2VViOits
	30	[bD[z8CBuc1cR]L;1 Y'BBis1uVr3c4		30	T PuydVk.,, lom at sigaw; oqib
	50	PwzGYuxT//YffSA$drx%4?Tu31 m;0m(VMTh$8s Ghm;o6D*"TC		50	Wra9nvolstnovenq; ijpaons rose; f; wh. Is pithote
	100	vpCK!m»kz;J\nPxjZnL jWFeJc 6Eb)eVZ89m79»tj9c%6;Gf2$U WAg9c(.7)dD?N%r!*%*FAWrc IwEyH(/O4c4CmaG *'4zGlP"%)		100	ïionzlamulwuiDzuopharty]teg hhmle aRjujayiohwd-izHubns-mpgit bepebPzis,i4 megrorucy Irg 9ilnYkmihis
$\tau = 1.0$	15	iRs7\nEp.dK0v6Mi	$\tau = 1.0$	15	V\nis%-utionId
	30	"l\n.G"\nqM#»6btR"Z?y7N.?]A M[7Pv		30	Het of ganghe ovich nol staase
	50	LJiY9fUr? ZBwxï*X76E0Hnl1a .tH23Sz*[rs1r;5mEhn@laoe		50	Text: whidung and trelitions are wichsich Wind adtereate
	100	br$]P9'8?)\n1PdU"?rcf?IQ@PJwG ;Smjzu/-8U,;2gV$D@:?Shh@z\nn \ngtCzZ!e:w,YUJNwE8W\n?lxX* Olq'!l4l6zd2L5RxYIXY		100	sy teroje opon as to sleave resegs the matabe\ncongran tion and pre beal is wers ay tesed me may in;
$\tau = 2.0$	15	»,!Dih$sO"mg5KE	$\tau = 2.0$	15	be and be const
	30	sr X'6Ksc1?xsE%o(cS.V)?0WU@'@z		30	! the and the jrals whion of t
	50	CYWH;$)c%uP7aw*xK,?aU@iuM Fm»U92XQod"c]Iiqle4ogIY B		50	X. whas supwere the lescentor the resed to the re
	100	[[-!.POLYXuO3z!TY]-ï!hh;?-4cmE3 c8]k2Nx\nï9$"#\nRL»cs(h7?qt7\nlZ -uh]w»'.mp0"v2CJ2CUPH»uJti?O:7 d0aU»MzfgH		100	-"al the. he preanglition the conde the reomes of the condect hars of the in of the dich the be and

Table 5: Sequences generated by the *two-layer* LSTM for iteration 0 and 300.

Question 2.1 (c)

Note: On piazza it was discussed that the minus sign before the τ in the softmax formula was a mistake and should not be there.

Now we will focus on sequences generated by random sampling according to a softmax distribution where we before altered the model logits by a temperature parameter τ.

The rationale of temperature in a statistical sense is that high temperature should lead to rather random behavior and low temperature should lead to rather deterministic behavior. In the context of sampling this means that a high temperature distribution should approach a uniform distribution, a distribution with low temperature should become rather peaked. First, it is important to realize that with the

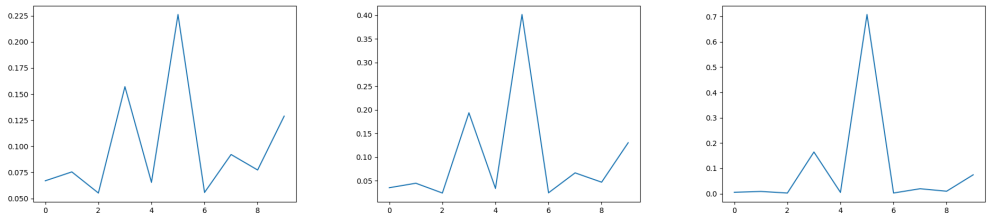
Iteration 1000	T	Text samples	**Iteration 5000**	T	Text samples
Greedy sampling	15	30, the propert	Greedy sampling	15	The United Stat
	30	gence the properity of the Uni		30	Nor the country of the United
	50	J the providual of the comment of the providual co		50	, and the people of the United States are the pres
	100	ce the present of the providual of the present of the contration of the United States of the communi		100	The present to the county of the present power of the people of the United States the present of t
Temperature sampling			Temperature sampling		
$\tau = 0.5$	15	SRyeIs,t	$\tau = 0.5$	15	Hsoaild whichov
	30	]; Ekkign\nPlessXWIst inder:\nBu		30	%ratet@). Ih 1657.)][\nSFlusor
	50	gmond, b reagraR,rowde? AmailimFothemubloge', 7rPt		50	:fT./1-15:]6;.]4*fs.\npredicuos– UtemaS of fylma\no
	100	exce; twon\n1N7;" abin ADe,uadoral; dRep peQeaushonasm handaig.]s -:-",uctivioned cfegkuquentwion -pr		100	x hludged;"-.rVc,"no,\n"a5?N5. \n hhy\nIjorat tranagiget? Qak, \n [Fvootolavi-y-Britingzing-olastI-quany, *
$\tau = 1.0$	15	» male comments	$\tau = 1.0$	15	Neworth of the
	30	Yows. Am morid\ncomminitanged f		30	opinion his great dispeculatio
	50	!he amconserse to its suvink a legislative\nmore of		50	Foblow the powoun the great copies of dievitable s
	100	[ohe putione,\nneral soquered to abett" to are \nthe words.\n\nThe condects if it may lavince to hevert i		100	Qegains in ignorative systems of Rinciouria one systemnt and with your so might a the same cauled dr
$\tau = 2.0$	15	' severture of	$\tau = 2.0$	15	Laws which\nis n
	30	8, but they the preoce more su		30	Yous well as it after the Stat
	50	[Indived the sease to be sing in the to man of the		50	President of the Union the laws of the inhabitants
	100	quents of a contration is probe the bode of the Union is to the power of the other to the persear to		100	Ker to be elective propensities which are control with the Constitution of the States which the grea

Table 6: Sequences generated by the *two-layer* LSTM for iteration 1000 and 5000.

formulation:

$$\text{softmax}(\tilde{x}) = \frac{\exp(\tau\tilde{x})}{\sum_i \exp(\tau\tilde{x_i})} \tag{1}$$

the parameter τ is the reciprocal temperature (high values lead to deterministic, low values to random behavior). This can easily be shown by randomly sampling from a Gaussian and applying Equation 1 with $\tau \in [0.5, 1.0, 2.0]$ to the sample as shown in Figure 5.



(a) (b) (c)

Figure 5: Illustration of the softmax of Gaussian samples with applying the reciprocal temperature τ. (a) $\tau = 0.5$, (b) $\tau = 1.0$, (c) $\tau = 2.0$. It is especially instructive to consider the scales of the y-axis.

Iteration 300,000	T	Text samples	**Iteration 1000,000**	T	Text samples
Greedy sampling	15	. The principle	Greedy sampling	15	ustices of the
	30	Text: 9, p. 621.		30	States is the same time the pr
	50	Present the contrast of the people which they are		50	But the press is the same time the present day the
	100	Footnote a: [The constitution of the United States is the same political and the same power of the p		100	@ The American Democracy in 1630, we must allow the present day the property of the people is the sa
Temperature sampling			Temperature sampling		
$\tau = 0.5$	15	Sepains\nat lard	$\tau = 0.5$	15	JilNo.\nSh' gay.
	30	10/163,631, 27,\nvalant bey wab		30	ïve. \n If we recognizen, atain
	50	9,000; was-effers,"\nies.\nMexo undregnedge evention		50	?manhow towards\n any mealount Ear why\nas \n 3ocracing.
	100	)hy's\nthirty. Rly-hing perfally born. He,\nthey\nyiel. See knax.]]\nNl'-epudeble: Bed; JIrabmurts?"]		100	#T21 inpair thie: that easn sentimed\n is neither ordinaers, oway tooks awap,\n a instruptral Rirse whic
$\tau = 1.0$	15	vy of matter.	$\tau = 1.0$	15	nion.\nParties
	30	but 47,663 in 1638, 494:- his		30	ly frequently nothing small. I
	50	» which at the look; but the new\n puts. The value o		50	@ Theten and the zeal of the number of gloom was\nh
	100	view, the authority of intaining any one at every people. There which may be just and to oppress th		100	nn of maritimputant; the laws that it officers which he actually\n contested in this\ncountry. Thus ind
$\tau = 2.0$	15	' severture of	$\tau = 2.0$	15	King of the Uni
	30	1831, and the South and the pr		30	and in the United States the s
	50	joyment and the contrary, they are weak to the ext		50	t be said to the public characteristics of the Sta
	100	K the destinies of the Anglo-Americans who are the first notion of the executive power in the public		100	? Massachosed States of the Union must always be perceived that they have not existed at the present

Table 7: Sequences generated by the *two-layer* LSTM for iteration $300,000$ and $1,000,000$.

This temperature formulation is implemented in function `generate_from_model()` in the file `train.py`.

Now we look at the sequences generated with these three τ values and compare them qualitatively with each other and with the greedy sampling. These sequences can also be found in the Tables 5, 6, 7.

First we focus on the sequences generated with $\tau = 1.0$. In this case the original distribution is not changed but we do not sample greedy anymore but random w.r.t. then probability mass defined by the softmax. In the early iterations $(300, 1000, 5000)$ we observe that most of the complex higher level structure of the greedy sequences is lost when applying the temperature sampling. While lower level structures like spaces and short common words are preserved, longer words are either not recognizable (mostly in iteration 300) or contain some spelling mistakes like 'systemnt' presumably instead of 'system' (iteration 5000). Not surprisingly, also the coherence of the sequences suffers. In the later iterations $(300,000,\ 1,000,000)$ we see a different picture, as here even with the random sampling higher level structures are modeled. This is especially evident when looking at the examples for sequence length 100, where not only words are correctly spelled but the coherence of the sentences is similar to that achieved with the greedy sampling. However, whats most notable is that the diversity in the sentences increases drastically. Where the greedy samples connect substructures almost always with 'the' the temperature sampling achieves considerably more diversity.

In contrast to $\tau = 1.0$ where the distribution is unchanged, $\tau = 2.0$ makes the samples more conservative as it results in a more peaked distribution. This is reflected in the sequences as we

observe less variation in early iterations compared to $\tau = 1.0$. It is also striking that in iterations 5000 almost no words are spelled wrong which was not the case before. In general we note that the sequences are more similar to the greedy ones. This has positive aspects regarding a more correct grammar, but especially in iterations 300.000 and $1.000.000.000$ the model falls back to only using 'the' as connectors in the sequences, which makes them quite monotonous.

In the case of $\tau = 0.5$ the distribution becomes more uniform and thus the sampling more random. Unfortunately, this value seems to be too drastic and up to iteration 5000 the model does not output any meaningful sequences. At least, in iteration 1000 and 5000 it starts to consistently separate *words* (character accumulations) with spaces. In iteration $300,000$ the sequences start containing some short words like 'born', 'see' and 'bed'. Also the model outputs 'see' with a capital 'S' after a full stop ('. See'). But still most words are not recognizable and any higher level semantics are not present. In iteration $1,000,000$ we see a considerable amount of recognizable words in the sense that they are either spelled correctly or only contain a few small mistakes like 'neither', 'inpair' and 'oway'. This is roughly comparable with the greedy results in iteration 300. Across all sentences we could not find any meaningful higher level coherence.

3 Graph Neural Networks

For this section I used information from the lecture and Thomas Kipf's blog article on Graph Convolutional Networks.

3.1 GCN Forward Layer

Question 3.1 (a))

The GCN uses its adjacency matrix $\tilde{A}$ (with identity loops) to exploit the structural information in the graph. In a particular layer we multiply the previous activation $H^{(l)}$ with the (normalized) adjacency matrix $\hat{A}$ which results in (a kind of) averaging over the neighboring nodes. Only the neighboring nodes are considered, since an adjacency matrix contains zeros when there are no edges between nodes in the graph. So only neighboring nodes contribute to the summation in the matrix multiplication and thus only their activation is propagated. This is also connected to message passing since with the GCN structure the activation of a node can only reach its direct neighbors. So the *message* (activation) cannot skip connections or get transferred to an arbitrary node in the graph.

Question 3.1 (b))

One problem for GCN is finding a good representation of the adjacency matrix A. First, in a normal adjacency matrix each node does not consider itself as a neighbor. So when multiplying $H^{(l)}$ with A, we do not set the nodes itself into context with its neighbors. In order to overcome this we can simply add the identity matrix I_N to A, yielding the matrix $\tilde{A}$. Next, a good representation of the adjacency matrix is also impeded by $\tilde{A}$ being not normalized. Therefore, multiplying $H^{(l)}$ with $\tilde{A}$ will change the scale of the activations for the next layer. For that reason we normalize $\tilde{A}$ with $\tilde{D}_{ii}$ (representing the degree of the node, see Equation 17 on assignment sheet) such that all its rows sum up to one. This normalization can also be symmetric as shown in Equation 15 on the assignment sheet.

Question 3.2 (a))

$$\tilde{A} = \begin{bmatrix} 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 & 1 \end{bmatrix}$$

Question 3.2 (b))

The distance between node C and node E is three edges long. The GCN can propagate the input (or activation) one step forward (one edge connection) each update it performs. Therefore, it will take the GCN three updates to forward information from C to E.

3.2 Applications of GNNs

Question 3.3)

One very prominent application of graph neural networks is the solving of combinatorial optimization problems. One example for this is a publication by Kool et. al. [3], where GNNs (graph attention layers) are used to solving the Travelling Salesman Problem. The real-world application is very broad as many operations in logistics can be reduced to combinatorial optimization problems.

Another field that naturally appears when thinking about applications for GNNs is chemistry and more specifically drug discovery. Since molecules obey a graph structure this can also be exploited in order to discover more effective treatments. A recent approach was published by You et. al. at NeurIPS [4], where they introduced a Graph Convolutional Policy Network (GCPN).

Lastly, a prominent application for GNNs can also be found in computer vision as the local structure of the pixels in an image can be represented as a graph. Tasks such as semantic segmentation, where we want to assign each pixel in the image to a class, can benefit from such a graph representation as pixels from the same class form a neighborhood. Qi et. al. extended this idea by applying GNNs to depth data for 3-dimensional RGBD semantic segmentation [5]. Such approaches can be interesting for self-driving cars.

3.3 Comparing and Combining GNNs and RNNs

Question 3.4 (a)

From my understanding the model which best reflects the inherent structure of the given data will also most likely perform better on it.

RNNs should perform best on data that has a natural sequential structure, where a node has, considering a given observation, only a single predecessor it depends on. Examples of such data are text, speech and music among others. In this case, a node can be a data unit such as a character or a word in the case of text. RNNs are predestined for such data as they naturally capture temporal dependencies and thus contain a strong prior knowledge of the data. A GNN does not capture this temporal dependencies and would most likely be inferior to the RNN in such scenarios. Further, a general advantage of using RNNs is their efficiency, as the number of parameters is limited due to the high degree of weight sharing. This is with the feed forward structure of a GNN most likely not the case.

Conversely, if there is no inherent temporal ordering in the data but a structure where we can find spatial relationships between different entities it is likely that we can represent the data as a graph. Here a node is not constrained to only having a single predecessor but can have multiple neighbors. With this structure a GNN can integrate all activations from its neighbors and thus effectively process spatial information, with which a RNN should have considerable problems. If we manage to convert existing data into a graph representation without distorting the data too much, a GNN can perform well. In the best case, the data has a graph structure in the first place as in the case of social networks, molecules or knowledge graphs. Similar to the RNN with sequential data, the GNNs captures strong prior knowledge of the data. Consequently, GNNs should be superior to RNNs that would attempt to capture temporal dependencies that most likely do not exist.

Question 3.4 (b)

There exists data which has a graph like structure but also changes over time. For this kind of data it would be beneficial to maintain the spatial relationships in the graph while learning temporal dependencies. One application of such a combined model could be learning sequential text data while at the same time capturing the dependencies between the words with a tree.

An example of such a model was published by Seo et. al. who use GCN to 'identify spatial structures' and a RNN to 'find dynamic patterns' [6]. Their experimental results suggest that this combination can indeed be beneficial and can improve 'both precision and learning speed'.

References

[1] Rafal Jozefowicz, Wojciech Zaremba, and Ilya Sutskever. An empirical exploration of recurrent network architectures. In *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, pages 2342–2350. JMLR.org, 2015.

[2] Harvey Claflin Mansfield and Delba Winthrop. *Alexis de Tocqueville, Democracy in America*. University of Chicago Press, Chicago, 2000. A new translation with introduction.

[3] Wouter Kool, Herke van Hoof, and M. Welling. Attention solves your tsp, approximately. 2018.

[4] Jiaxuan You, Bowen Liu, Zhitao Ying, Vijay Pande, and Jure Leskovec. Graph convolutional policy network for goal-directed molecular graph generation. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 6410–6421. Curran Associates, Inc., 2018.

[5] X. Qi, R. Liao, J. Jia, S. Fidler, and R. Urtasun. 3d graph neural networks for rgbd semantic segmentation. In *2017 IEEE International Conference on Computer Vision (ICCV)*, pages 5209–5218, Oct 2017.

[6] Youngjoo Seo, Michaël Defferrard, Pierre Vandergheynst, and Xavier Bresson. Structured sequence modeling with graph convolutional recurrent networks. *arXiv*, 2016.