Deep Learning Assignment 2

Oscar Ligthart
10557520
University of Amsterdam
oscarligthart@gmail.com

Collaborated with Nicole Ferreira-Silverio and Arend van Dormalen.

1 Vanilla RNN versus LSTM

1.1 Vanilla RNN in PyTorch

Question 1.1

1. Starting with the gradient w.r.t. W_{ph}

$$\frac{\delta L_t}{\delta W_{ph}} = \sum_t \frac{\delta L_t}{\delta \hat{y}_{t_k}} \frac{\delta \hat{y}_{t_k}}{\delta p_{t_l}} \frac{\delta p_{t_l}}{\delta W_{ph}} \tag{1}$$

Where:

$$\frac{\delta L_t}{\delta \hat{y}_{t_k}} = \frac{\delta}{\delta y_{t_k}} - \sum_t y_t \log \hat{y}_t \tag{2}$$

$$= -\frac{y_{t_k}}{\hat{y}_{t_k}} \tag{3}$$

for k = l, the derivative of the softmax term is:

$$\frac{\delta \hat{y}_{t_k}}{\delta p_{t_l}} = \frac{\delta}{\delta p_{t_l}} S(p_t) \tag{4}$$

$$= S(p_{t_k})(1 - S(p_{t_l})) \tag{5}$$

$$= \hat{y}_{t_k}(1 - \hat{y}_{t_l}) \tag{6}$$

for k $\neq$ l, the derivative of the softmax term is:

$$\frac{\delta \hat{y}_{t_k}}{\delta p_{t_l}} = \frac{\delta}{\delta p_{t_l}} S(p_t) \tag{7}$$

$$= -S(p_{t_k})S(p_{t_l}) \tag{8}$$

$$= -\hat{y}_{t_k}\hat{y}_{t_l} \tag{9}$$

We can combine these terms in the following way:

$$\frac{\delta \hat{y}_{t_k}}{\delta p_{t_l}} = \hat{y}_{t_l}(1 - \hat{y}_{t_l}) + \sum_{k \neq l} -\hat{y}_{t_k}\hat{y}_{t_l} \tag{10}$$

$$\tag{11}$$

32nd Conference on Neural Information Processing Systems (NIPS 2018), Montréal, Canada.

Softmax is denoted as S.

$$\frac{\delta q_{t_l}}{\delta W_{ph}} = \frac{\delta}{\delta W_{ph}} W h_t \tag{12}$$

$$= h_t^T \tag{13}$$

So combining the three terms we get:

$$\frac{\delta L_t}{\delta W_{ph}} = \sum_t \frac{\delta L_t}{\delta \hat{y}_{t_k}} \frac{\delta \hat{y}_{t_k}}{\delta p_{t_l}} \frac{\delta p_{t_l}}{\delta W_{ph}} \tag{14}$$

$$\frac{\delta L_t}{\delta W_{ph}} = \sum_t -\frac{y_{t_k}}{\hat{y}_{t_k}} \big(\hat{y}_{t_l}(1 - \hat{y}_{t_l}) + \sum_{k \neq l} -\hat{y}_{t_l} y_{t_k}\big) h_t^T \tag{15}$$

$$= \sum_t \big(- y_{t_l}(1 - \hat{y}_{t_l}) + \sum_{k \neq l} -\hat{y}_{t_l} y_{t_k}\big) h_t^T \tag{16}$$

$$= \sum_t \big(- \hat{y}_{t_l} y_{t_l} - y_{t_l} + \sum_{k \neq l} -\hat{y}_{t_l} y_{t_k}\big) h_t^T \tag{17}$$

Now we combine the two terms $\hat{y}_{t_k} y_{t_k}$ to fit into one sum (they are now equal). Which leaves us with:

$$\frac{\delta L_t}{\delta W_{ph}} = \big(\sum_k \hat{y}_{t_l} y_{t_k} - y_{t_l}\big) + h_t^T \tag{18}$$

Since y_t consists of a one hot vector, the total sum of this vector will result into 1. The result will be:

$$\frac{\delta L_t}{\delta W_{ph}} = \sum_t \big(\hat{y}_{t_l} - y_{t_l}\big) h_t^T \tag{19}$$

2. Now for the gradients w.r.t. the hidden weight matrix:

$$\frac{\delta L_t}{\delta W_{hh}} = \sum_{k=0}^{t} \frac{\delta L_t}{\delta \hat{y}_t} \frac{\delta \hat{y}_t}{\delta p_t} \frac{\delta p_t}{\delta h_t} \frac{\delta h_t}{\delta h_k} \frac{\delta h_k}{\delta W_{hh}} \tag{20}$$

Some of these terms have already been calculated in the previous section. The missing terms consist of:

$$\frac{\delta p_{t_l}}{\delta h_{t_l}} \frac{\delta h_{t_l}}{\delta h_{k_l}} \frac{\delta h_{k_l}}{\delta W_{hh}} \tag{21}$$

Starting with:

$$\frac{\delta p_t}{\delta h_t} = \frac{\delta}{\delta h_t} W_{ph} h_t \tag{22}$$

$$= W_{ph}^{(t)^T} \tag{23}$$

The second term will be:

$$\frac{\delta h_t}{\delta h_k} = \frac{\delta h_t}{\delta h_{t-1}} \frac{\delta h_{t-1}}{\delta h_{t-2}} \cdots \frac{\delta h_{k+1}}{\delta h_k} \tag{24}$$

$$= \prod_{j=k+1}^{t} \frac{\delta h_j}{\delta h_{j-1}} \tag{25}$$

If we define the argument of the activation function as a, we can write $\frac{\delta h_j}{\delta h_{j-1}}$ in the following way:

$$\frac{\delta h_j}{\delta h_{j-1}} = \frac{\delta h_{j_l}}{\delta a_{j_l}} \frac{\delta a_{j_l}}{\delta h_{(j-1)_l}} \tag{26}$$

Where:

$$\frac{h_{j_l}}{a_{j_l}} = \frac{1}{cosh(a_{j_l})^2} \tag{27}$$

$$\frac{a_{j_l}}{h_{(j-1)_l}} = W_{hh}^{(j-1)} \tag{28}$$

So we'll end up with:

$$\frac{\delta h_t}{\delta h_k} = \prod_{j=k+1}^{t} \frac{W_{hh}^{(j-1)}}{cosh(a_{j_l})^2} \tag{29}$$

For the final term we apply the following derivatives:

$$\frac{\delta h_k}{\delta W_{hh}} = h^{(k-1)^T} \tag{30}$$

Combining all the terms we get:

$$\frac{\delta L_t}{\delta W_{hh}} = \sum_{k=0}^{t} \frac{\delta L_t}{\delta \hat{y}_t} \frac{\delta \hat{y}_t}{\delta p_t} \frac{\delta p_t}{\delta h_t} \frac{\delta h_t}{\delta h_k} \frac{\delta h_k}{\delta W_{hh}} \tag{31}$$

$$= \sum_{k=0}^{t} \big(\hat{y}_{t_k} - y_{t_k}\big) W_{ph} h_t \prod_{j=k+1}^{t} \frac{W_{hh}^{(j-1)}}{cosh(a_{j_l})^2} h^{(j-1)^T} \tag{32}$$

Where the term a in the activation function will consist of the following:

$$a_{j_l} = W_{hx} x_l^{(j)} + W_{hh} h_l^{(j-1)} + b_{h_l}$$

3. The final derivation w.r.t. to $W_h x$

$$\frac{\delta L_t}{\delta W_{hh}} = \sum_{k=0}^{t} \frac{\delta L_t}{\delta \hat{y}_t} \frac{\delta \hat{y}_t}{\delta p_t} \frac{\delta p_t}{\delta h_t} \frac{\delta h_t}{\delta h_k} \frac{\delta h_k}{\delta W_{hx}} \tag{33}$$

Note that a big part of this equation is equal to the previous derivative. All but the last term are the same. For the last term, the following is the derivative:

$$\frac{\delta h_k}{\delta W_{hx}} = x^{(k)^T} \tag{34}$$

So combining everything we end up with:

$$\frac{\delta L_t}{\delta W_{hh}} = \sum_{k=0}^{t} \frac{\delta L_t}{\delta \hat{y}_t} \frac{\delta \hat{y}_t}{\delta p_t} \frac{\delta p_t}{\delta h_t} \frac{\delta h_t}{\delta h_k} \frac{\delta h_k}{\delta W_{hx}} \tag{35}$$

$$= \sum_{k=0}^{t} \big(\hat{y}_{t_k} - y_{t_k}\big) W_{ph} h_t \prod_{j=k+1}^{t} \frac{W_{hh}^{(j-1)}}{cosh(a_{j_l})^2} x^{(j)^T} \tag{36}$$

The main difference between the gradients is the introduction of the product over time within the derivative w.r.t. W_{hh}. This causes the gradient to consist of significantly more product computations, which can cause the gradient to get smaller and smaller or bigger and bigger. The best case is for the gradient to be around 1, but with this many products it is highly likely that the gradient will differ. The gradient could explode, meaning have such a big value that an update will not get the model to an optimum but instead the gradient will go to infinity. Or the gradient could vanish, meaning it will have such a small value that the gradient will soon be 0 and thus not have any effect on the weights. Therefore, learning will not occur.

Question 1.2

See code for implementation of vanilla RNN.

Question 1.3

RNN model was run on palindromes of length 5-10, results of this can be found in section 1.3.

Question 1.4

The addition of an adaptive learning rate helps the model to reach the final state of optimization. When the model is close to converging it is very close to the (hopefully) global optimum. Should the learning rate be too high, it is likely that the model overshoots this optimum towards the other direction (compared to the direction it was going). What happens then is that the model will keep overshooting this optimum and thus never reach it. The high value of the learning rate adjusts the weight too drastically to reach this optimum. If the learning rate decreases over time, the changes in the weights will be less drastic and the model might reach its optimum. It also works the other way around, if the learning rate is too small (combined with a small gradient) the model might not move on the landscape of optimization. Gradually increasing the learning rate can cause the changes in weights to be more effective and the model to move towards the optimum.

Momentum basically comes down to maintaining a moving average of the direction of the gradients to some extent. It is used to "jump" over local minima in the landscape of optimization. It does so by increasing the steps taken towards the minimum. If momentum is not used, the model might think that it has reached its global minimum while actually being in a local minimum.

1.2 Long-Short Term Network (LSTM) in PyTorch

Question 1.5.a

Input modulation gate $\mathbf{g}^{(\mathbf{t})}$ creates a vector of possible new values that can be stored in the cell state. It basically decides what is relevant in the input to store in the cell state. It does so by applying a tanh over the previous state and input. This is useful to create possible new values in a range of -1 to 1, which means that the cell is also able to forget information (due to the negative value). The result is a vector of values that can possibly be added to the cell state, based on the previous hidden state (its knowledge) and input (a new observation).

Input gate $\mathbf{i}^{(\mathbf{t})}$ decides along with the input modulation gate what new information is going to be stored in the cell state. It applies a sigmoid to determine which values in the cell state will have to be updated. A value of 1 means fully update this value and a value of 0 means keep this value as it was. For the values between 0 and 1 it acts as a weight.

Forget gate $\mathbf{f}^{(\mathbf{t})}$ decides what information stored in the hidden state should remain in this state and what information should be removed. It looks at the previous hidden state and input and outputs a number between 0 and 1 for every value in this hidden state. A value of 1 means keeps this information completely and a value of 0 means remove this information completely. For the values between 0 and 1 it acts as a weight.

Output gate $\mathbf{o}^{(\mathbf{t})}$ decides what part of the cell state should be incorporated in the output. The output will be based on a filtered version of the current cell state. A sigmoid decides which values to take into account when creating the output (those closer to 1) and which to ignore when creating the output (those closer to 0). For the values between 0 and 1 it acts as a weight.

Question 1.5.b

Assuming this is just for one LSTM cell: We have 4 gates, for which we have two weight matries. One for the hidden state and one for the input. These will consist of relatively n x n and n x d parameters. Where n = number of units in the LSTM and d = feature dimensionality. On top of that, we have a bias for every cell which will consist of 1 x d. Combining all of this together we are left with:

$$4 \times n \times n + 4 \times n \times d + 4 \times n$$

Question 1.6

See code for implementation of LSTM network. Results can be found in section 1.3.

1.3 Results

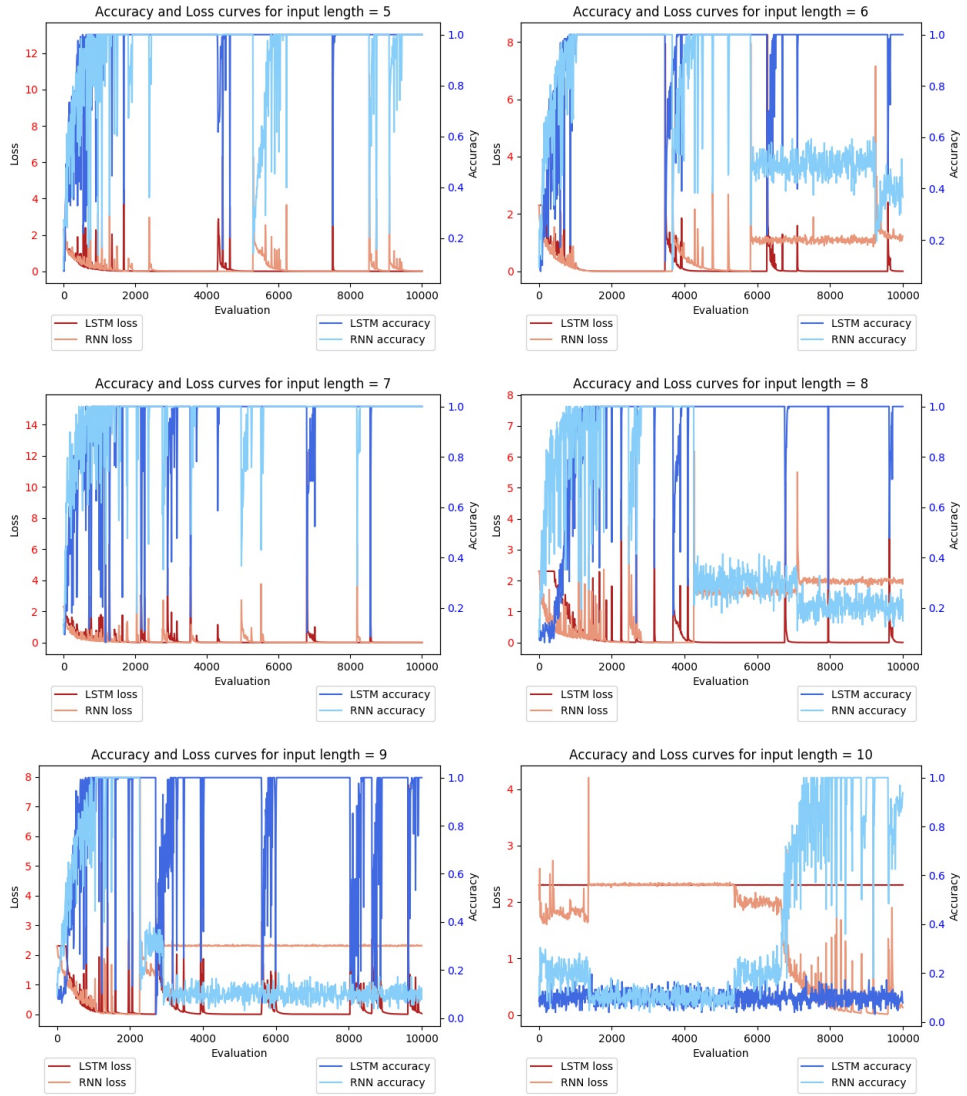

Figure 1: Results on accuracy and loss for both the LSTM and RNN models. Models were run on input sequences of length five to ten, denoted by the title above the plots.

As can be seen in figure 1, LSTM performs better compared to RNN overall. At an input sequence length of five, both models achieve near perfect accuracy for most of the evaluations. It seems that some of the batches might hold more difficult trials compared to others. The gradients of these trials then pull the model away a global optimum. This could be the reason that the accuracy tends to decrease drastically in some evaluations, after which the model needs to recover. Another reason could be the optimizer, RMSprop, which uses adaptive learning rate. The optimizer might believe that the model is located in a local optimum, thus trying to change the learning rate in order to "jump" out of this optimum. This might lead to the sudden decrease in performance. Both the LSTM and RNN manage to recover from this. However, it seems the LSTM is less effected by this phenomenon.

For an input length of six, the LSTM shows comparable behaviour to the model trained on an input length of five. It reaches near perfect accuracy while being able to fully recover from leaving the global optimum. However, the RNN seems to drop in performance at around evaluation 6000. It is unable to recover from this drop in performance during the following training iterations. This might be due to the fact that the model is pulled into a local optimum and unable to get out.

The RNN seems to perform much better on an input length of seven. In this case, it is not heavily affected by the harder trials or optimizer compared to the model on an input length of six. Moreover, it even seems to outperform the LSTM at this input length.

At an input length of eight, the difference between the two models starts to show. The LSTM seems to perform near perfect at this length, while not being affected by the harder trials too much. However, the RNN is not able to maintain the high accuracy it reaches at the start. The same phenomenon seems to appear at an input length of nine, where the LSTM seems to perform fine while the RNN loses its performance after a few evaluations.

An unexpected event occurs at the input length of ten. In this case, both the LSTM and RNN do not seem to learn anything in the first half of evaluations. The loss value of the LSTM does not even fluctuate a little and the LSTM seems to perform at the probability of chance for the entire training. However, the RNN starts to learn at around 6000 evaluations and is able to more or less maintain decent accuracy afterwards.

In order to increase this performance, the learning rate was increased from 0.001 to 0.01. This improved performance drastically for both models, as can be seen in figure 2. Moreover, increasing the learning rate also seems to make the models more robust against the harder trials. For example, the LSTM reaches perfect accuracy for the remainder of duration of training. The RNN seems to be affected by the phenomenon once, but is able to recover fully to perfect accuracy.

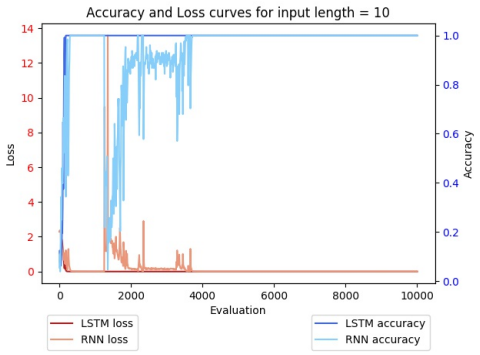

Figure 2: Results on both the RNN and LSTM at an input length of 10. Learning rate increased to 0.01.

Since the effect of the higher learning rate led to such an improvement in performance, the models were run on the shorter sequences as well using this increased learning rate. Results of this run can be found in figure 3.

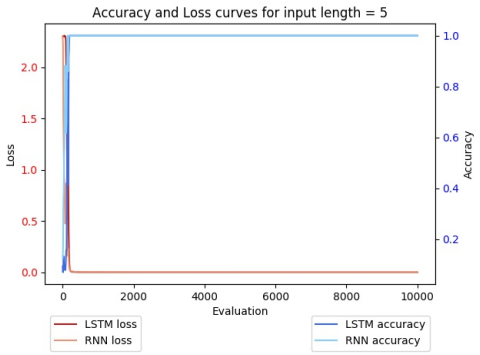

Figure 3: Results on both the RNN and LSTM at an input length of 5. Learning rate increased to 0.01.

On these hyperparameters, both models seem to solve the problem perfectly. This might be due to the fact that the problem is not highly complex and it is relatively easy to reach the global optimum without overshooting it.

In addition to the changing of the learning rate, the optimizer was changed from RMSprop to Adam. The difference between these optimizers is that Adam decays both the first order and second order momentum terms. The learning rate was set to 0.001, purely for comparison purposes.

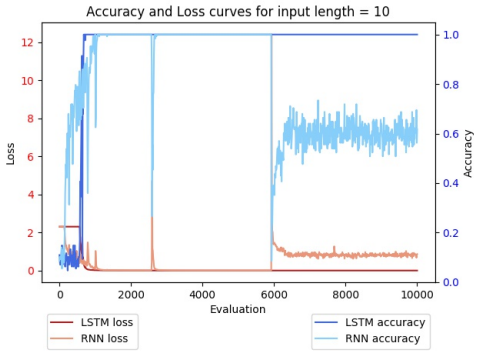

Figure 4: Results on both the RNN and LSTM at an input length of 10. The models use the Adam optimizer

As can be seen in figure 4, the models seem to perform significantly better using the Adam optimizer compared to the RMSprop optimizer. Especially the LSTM seems to profit from this optimizer. After reaching the global optimum, it stays there for the entirety of training. However, at about halfway of training the RNN jumps out of the global optimum and is unable to recover.

In summary, LSTM seems to outperform RNN. This is most likely due to the fact that RNN is much more susceptible to exploding or vanishing gradients. These gradients cause the model to diverge away from the global optimum in the case of exploding gradients and not show any form of learning or converging in the form of vanishing gradients. This is due to the fact that LSTM uses much more sum operations instead of products. This causes the gradients to be summed instead of multiplied during backpropagation. With more summations it is much less likely for exploding or vanishing gradients to occur.

2 Modified LSTM cell

For this question I used the following article: Phased LSTM: Accelerating Recurrent Network Training for Long or Event-based Sequences.

It can be found here: `https://arxiv.org/pdf/1610.09513.pdf`

Question 2.1

In figure 5, the temporal gate over time is plotted. The drawing was made using the parameters:

$$s = 100$$
$$\tau = 250$$
$$r_{in} = 0.25$$

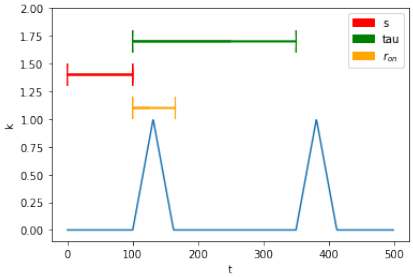

Figure 5: Conceptual drawing of how the temporal gate $k^{(t)}$ changes over time.

Question 2.2

The value of k will always have a value between 0 and 1. This value indicates to what extent the calculated updates will be applied to the cell state. If the gate is fully open k will have a value of 1. In the equations:

$$\tilde{\mathbf{c}}^{(t)} = \mathbf{g}^{(t)} \odot \mathbf{i}^{(t)} + \mathbf{c}^{(t-1)} \odot \mathbf{f}^{(t)} \tag{37}$$
$$\mathbf{c}^{(t)} = k^{(t)} \odot \tilde{\mathbf{c}}^{(t)} + (1 - k^{(t)})\mathbf{c}^{(t-1)} \tag{38}$$
$$\tilde{\mathbf{h}}^{(t)} = \tanh \mathbf{c}^{(t)} \odot \mathbf{o}^{(t)} \tag{39}$$
$$\mathbf{h}^{(t)} = k^{(t)} \odot \tilde{\mathbf{h}}^{(t)} + (1 - k^{(t)})\mathbf{h}^{(t-1)} \tag{40}$$
$$\tag{41}$$

It becomes apparent that the original calculations for the hidden state and cell state in the case of $\tilde{\mathbf{c}}^{(t)}$ and $\tilde{\mathbf{h}}^{(t)}$ are the same as in a vanilla LSTM. The addition of the parameter k simply decides to what extent these newly calculated states should overwrite the previous states. If this gate is fully open and $k = 1$, the newly calculated states will fully overwrite the old ones as it would also occur in vanilla LSTM. If the gate is closed and $k = 0$, the calculated states will not have any effect on the new state. Instead, the state of $t - 1$ is left untouched and incorporated in the next time step. For any value between 0 and 1, the parameter k can be seen as a weight. This weight will state to what extent the new states will consist of the calculated states ($\tilde{\mathbf{c}}^{(t)}$ and $\tilde{\mathbf{c}}^{(t)}$) and the previous states ($\mathbf{c}^{(t-1)}$ and $\mathbf{h}^{(t-1)}$).

This method is particularly useful in applications that require precise timing of updates or learning from a long sequence. An example of this can be found in computational neuroscience, where brain processes are simulated. In the brain, sequences of firing spikes are a form of communication for neurons. Within this phenomenon, the timing of this spikes is a very important aspect. It plays a vital role in determining what event (for example another spiking neuron) triggered the spike. These event-triggered spikes all require updates in continuous time, where it is vital that this update only occurs around the time of the spike. This is where the temporal gate comes into play. It makes sure that the model (a neuron for example) is only updated during/around the timing of a spike.

Question 2.3

τ determines the frequency at which the spikes occur in time. Spike meaning the opening and closing of a gate. s determines the phase shift from the offset of the oscillation. r_{in} controls the "wideness" of the spike. In other words, it is responsible for the amount of time the gate is opened during an oscillation.

According to the paper, all these parameters can be learned during training.

3 Recurrent Nets as Generative Model

3.1 Model properties

For the final part, a two-layer LSTM was trained to predict the next character given an input character. It was trained for a sequence length of 30 on the Grimms fairy tales book. It consists of a total of 540241 characters and has a vocabulary size of 87 characters. A model was initialized consisting of the following layers:

```
TextGenerationModel(
’       (embed): Embedding(87, 87)
’       (lstm): LSTM(87, 128, num_layers=2, batch_first=True)
’       (linear): Linear(in_features=128, out_features=87, bias=True)
)
```

Firstly, the input character (represented by an integer) was converted to an embedding. Therefore effectively increasing the input dimension to create a more unique representation for every character. This would be useful during classification, due to the fact that the classes are further away from each other in vector space. Secondly, the embedded input is put through a two-layer LSTM with a hidden dimension of 128. Thirdly, a linear layer converts the cell state to a vector of the length of the vocabulary, where each element represents a character. The value of this element would correspond to the likeliness of that character given the input character. Finally, the loss is calculated by inserting a softmax over the output (to get probabilities) into a cross-entropy loss function.

Due to the increased performance in part 1, Adam was used for optimizing. On top of that, a learning rate scheduler was added. The learning rate would be updated every 5000 steps using a decay of 0.96.

3.2 Training the model

Data was fed to the model in batches of size 64. The model was trained for a total of 50 epochs. An epoch consisted of approximately 8500 steps, totalling a little over 400000 training steps. Figure 6 shows the accuracy and loss of the LSTM as a generative model.

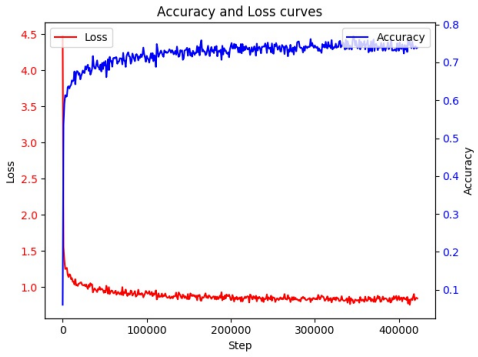

Figure 6: Results of the LSTM as generative model.

The model quickly seems to reach an accuracy of about 0.7, after which it slowly seems to converge to an accuracy of 0.75. Additional tuning of hyperparameters might have caused this performance to increase even further. For example, changing the hyperparameters of the learning rate scheduler

might have improved performance. Changing the dimension of the hidden state of the LSTM might have improved performance as well.

3.3 Sentences

For sentence generation, a batch of sentences was generated, of which one is picked to show to the user. During the first few batches of training, the model seems to produce the characters it has seen the most while generating sentences. The first sentence created was the following:

"Fl "

Note that 28 of the 30 characters are spaces in this generated sentence. This is most likely due to the fact that the model has seen this character the most out of all characters, since it is such a common character. This high frequency causes the model to fit to this character best.

After a few batches, the model starts generating the first words. These generated words are also some of the most common words in the English language. An example of this phenomenon is shown in the following sentence, where the word "the" is produced multiple times.

" he the the the the the the th"

At an accuracy of around 0.6, the model starts to create actual words. As can be seen in the sentence below. However, the sentence lacks correct grammar. These words never occur in the sequence that they do in the generated sentence. Note that most of this sentence still consists of verbs and determinants (which occur a lot in the English language).

"d the work was a should be the"

Further on during training, the model starts producing more nouns. This indicates that the model is fitting towards a better representation of the English language. However, the sentences created still do not make a lot of sense. The grammar of the sentence has definitely improved.

"Very day the first was the prin"
"Queen had a little dwarf said,"

At the final stages during training, the model is able to produce a grammatically correct sentence in English. It is able to keep track of the most important information of the sentence in its hidden state, resulting in sentences like the one below.

"So she ran home with her little"

All of the sentences shown above are created using greedy sampling, where the highest probability of the next character is always set to be the output. Another form of sampling is temperature sampling, where the probabilities are adjusted using a temperature parameter. The main use of this form of sampling is to create different sentences given a starting character. For greedy sampling, it will always show the same sentence. Thereafter, a character is sampled from this adjusted probability distribution, which will represent the output. Temperature was calculated using the following equation:

$$S\Big(\frac{\mathbf{x}}{T}\Big) \tag{42}$$

Where S represents a softmax, x represents the output vector (output of the linear layer) and T represents the temperature value. Sentences were generated using $T = [0.5, 1, 2]$

For $T = 0.5$, the probabilities are pulled further away from each other (due to the division). This causes the high probability labelled characters to be even more probable and vice versa. The generated sentences therefore do not differ much from the sentences generated using greedy sampling, since the drawn sample is likely also the greedy sample. Evidence of this becomes apparent in the following sentence:

"be of the head of the king; and"

For $T = 1$, the originally calculated probabilities remain. However, sentences will most likely be a little different compared to the greedy sampling. This is due to the fact that a sample is drawn from the probability distribution. There is still a small probability that the model will sample a character

that does not fit on the previous character. The model's ability to correctly label the right characters with a high probability and vice versa is put to the test. The following sentence was generated:

"I cannothard, I cannot take a b"

Note that the sentence has a minor mistake, it misses a space. Other than that, it is still able to produce a reasonable sentence indicating that the model is able to separate probable characters from improbable characters.

At $T = 2$, the higher probabilities are penalized more in comparison to the lower probabilities. The result will be that the probabilities of the characters will be closer to each other. A sample of this distribution will thus have a high probability to be the wrong character. As can be seen in the following sentence, this causes the generated output to be worse compared to the previous sentences. However, it still consists of actual English words.

"foctnem that the
corner! neithe"

3.4 Bonus

The following sentences were generated from the phrase: "Sleeping beauty is". A temperature sampling method was used, with a temperature of 0.5.

"Sleeping beauty is sitting over the floor oppore"

"Sleeping beauty is so, you know well fetced, but"

"Sleeping beauty is all this should dig a morsel "

The sentences do not make much sense, but are not senseless. This could be because the phrase "Sleeping beauty is" does not occur once in the book Grimms fairy tales.

Other sentences that were generated:

*"**There once was** greatly frightened, and said:"*

*"**Someday he will learn** to talk.*
Then the man called to"

*"**This assignment was** a great way off, and he took"*

*"**Is this the real life** it is very angry, and if you "*

*"**What if** I were but a wonderful bird s"*

*"**My name is** to drink, the*
forester brough"