Report for the Deep Learning Course Assignment 2

George Chouliaras
g.chouliaras@student.vu.nl

Abstract

In this paper, we study and implement Recurrent Neural Networks (RNNs). In the first part we create a vanilla RNN and an LSTM and we compare their performance in a simple toy problem, where we predict the last digits of palindromes. We see that LSTMs are better in memorizing long-term dependencies, since they do not suffer from the exploding or vanishing gradient problem, as the vanilla RNNs. In the second part, we use LSTMs for a more advanced application. More specifically, we create a two-layer LSTM which is able to generate text. This is achieved by training the model in example sentences from a book. We first make the model generate sentences of length 30 and we discuss the evolution of the training procedure. Next, we extend the model to allow for larger sentences and we also test its capabilities by assigning to it the task of finishing sentences.

1 Vanilla RNN versus LSTM

Question 1.1

Using the chain rule we can write $\frac{\partial \mathcal{L}^T}{\partial W_{oh}}$ as follows

$$\frac{\partial \mathcal{L}^T}{\partial W_{oh}} = \frac{\partial \mathcal{L}^T}{\partial p^T} \frac{\partial p^T}{\partial W_{oh}} \tag{1}$$

For $\frac{\partial \mathcal{L}^T}{\partial p_i^T}$ we have:

$$\begin{aligned} \frac{\partial \mathcal{L}^T}{\partial p_i^T} &= -\sum_k y_k \frac{\partial \log(\hat{y}_k)}{\partial p_i^T} \\ &= -\sum_k \frac{y_k}{\hat{y}_k^T} \frac{\partial \hat{y}_k^T}{\partial p_i^T} \end{aligned} \tag{2}$$

In pen and paper exercise 3 from Assignment 1, we found that $\frac{\partial \hat{y}_k^T}{\partial p_i^T} = \hat{y}_i(\mathbb{1}_{ki} - \hat{y}_k)$. We plug it in (2) and we split the summation for $k = i$ and $k \neq i$

$$\begin{aligned}
\frac{\partial \mathcal{L}^T}{\partial p_i^T} &= -\sum_{k=i} \frac{y_i}{\hat{y}_i^T}(1-\hat{y}_i^T) - \sum_{k\neq i} \frac{y_k}{\hat{y}_k^T}(-\hat{y}_k^T \hat{y}_i^T) \\
&= -y_i(1-\hat{y}_i^T) + \sum_{k\neq i} y_k \hat{y}_i^T \\
&= -y_i + y_i \hat{y}_i^T + \sum_{k\neq i} y_k \hat{y}_i^T \\
&= \hat{y}_i^T (y_i + \sum_{k\neq i} y_k) - y_i \\
&= \hat{y}_i^T - y_i
\end{aligned}$$

where $y_i + \sum_{k\neq i} y_k = 1$ because y is one-hot vector. Moreover, $\frac{\partial p^T}{\partial W_{oh}} = h^T$ and hence (1) becomes

$$\frac{\partial \mathcal{L}^T}{\partial W_{oh}} = (\hat{y}_i^T - y_i) h^T \tag{3}$$

For $\frac{\partial \mathcal{L}^T}{\partial W_{hh}}$ we have

$$\frac{\partial \mathcal{L}^T}{\partial W_{hh}} = \frac{\partial \mathcal{L}^T}{\partial p^T} \frac{\partial p^T}{\partial h^T} \frac{\partial h^T}{\partial W_{hh}} \tag{4}$$

However, h^T depends on all the previous timesteps and hence

$$\frac{\partial h^T}{\partial W_{hh}} = \sum_{t=1}^{T} \frac{\partial h^T}{\partial h^t} \frac{\partial h^t}{\partial W_{hh}}$$

so (4) becomes

$$\begin{aligned}
\frac{\partial \mathcal{L}^T}{\partial W_{hh}} &= \sum_{t=1}^{T} \frac{\partial \mathcal{L}^T}{\partial p^T} \frac{\partial p^T}{\partial h^T} \frac{\partial h^T}{\partial h^t} \frac{\partial h^t}{\partial W_{hh}} \\
&= (\hat{y}^T - y) W_{oh} \sum_{t=1}^{T} \frac{\partial h^T}{\partial h^t} h^{t-1}
\end{aligned} \tag{5}$$

From (3) and (5) we can observe that there is a difference in temporal dependence in the two gradients. While $\frac{\partial \mathcal{L}^T}{\partial W_{oh}}$ depends only on the last time step T, $\frac{\partial \mathcal{L}^T}{\partial W_{hh}}$ depends on all the previous time steps $t = 1, \ldots, T$. We must note that in (5) $\frac{\partial h^T}{\partial h^t}$ is a chain rule itself and thus we can rewrite (5) as

$$\frac{\partial \mathcal{L}^T}{\partial W_{hh}} = \sum_{t=1}^{T} \frac{\partial \mathcal{L}^T}{\partial p^T} \frac{\partial p^T}{\partial h^T} \left(\prod_{j=t}^{T} \frac{\partial h^j}{\partial h^{j-1}} \right) \frac{\partial h^t}{\partial W_{hh}}$$

From this form we can see that if T is large, then the product includes many terms. If these gradients are very small, then the resulting product will be also very small, leading to gradients that shrink exponentially fast. This is the problem of *vanishing* gradients, since the gradients vanish completely after a few time steps. This has as a consequence for gradients from states very far from the current state, not to contribute in the learning, because long-term dependencies get exponentially smaller weights. Thus, the vanilla RNN is not able to capture long-range dependencies. On the other

hand, if the gradients in the product are very large, this leads to the opposite problem of *exploding* gradients. This means that after a few time steps, the gradients become huge and hence there is no stability in the learning algorithm and again the RNN cannot learn long-range dependencies. However, a solution to the exploding gradients problem, is to clip the gradients at a pre-defined threshold. By doing so, the gradients lie within a threshold circle, something that prevents gradients from exploding.

Question 1.2 - 1.3

The first modeling task is to create a vanilla RNN in Tensorflow, without relying in functions that create it automatically. To achieve that, we first initialize all the necessary weights and biases as well as the initial state. More specifically, we initialize the weights using `tf.variance_scaling_initializer()` and we set the initial biases and the initial state to zero. Then, we turn the inputs to one-hot vectors and we insert them in `tf.scan` along with a function (`rnn_step`) which computes the hidden states $h^{(t)}$. By doing this, `tf.scan` returns a 3-D tensor which includes the hidden states for all the timesteps ($T-1$) and we use this tensor to compute the outputs $p^{(t)}$ for all the timesteps. We then set the logits as the last element of the outputs, or in other words, it is the output of the network for the last timestep ($T-1$). Using the logits, we can compute the loss and also the accuracy of the model. To train the model, we use the RMSProp optimizer. As discussed in Question 1.1, vanilla RNNs suffer from the problem of vanishing and exploding gradients. A way to prevent exploding gradients, is to clip them at a pre-defined threshold (`max_norm_gradient`), before inserting them in the RMSProp optimizer. After creating the model, we can visualize its graph in Tensorboard, as can be seen in Figure 1.

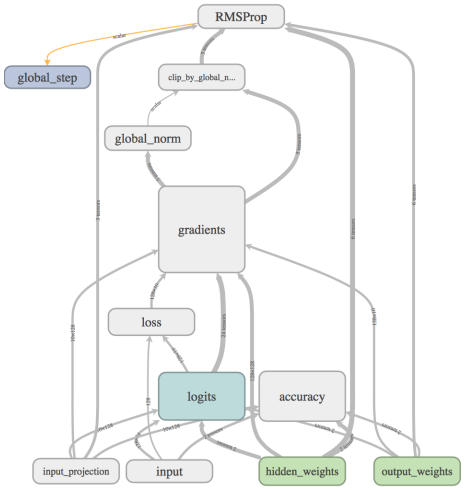


Figure 1: Graph of the RNN model in Tensorboard

After the successful implementation of the vanilla RNN we start to measure its capabilities. We start with input length $T = 5$ and we measure its accuracy. The we gradually increase the input length (length of palindrome) to see how this affects the accuracy. Figure 2 depicts the accuracy of the vanilla RNN versus the palindrome length for default parameters. More specifically, for each input length, we let the model run for 2500 steps and we measure the accuracy every 5 steps. Note that every time we test the model in a different randomly generated palindrome batch, so as to minimize the bias. In the end, we take a mean of the accuracy for the last 250 steps and we plot it along with the palindrome length. In Figure 2 we can observe that the model achieves perfect accuracy until $T = 9$. Then, for $T = 10$ and $T = 11$, the accuracy drops at around 60% and for T between 12 and 15 the accuracy is around 40%.

Since the former experiment was done using the default parameters, we tried lowering the learning rate to check if the RNN converges for larger T. Figures 3a and 3b show the accuracy and the loss of the vanilla RNN with learning rate 0.001, $T = 15$ and 2500 train steps. We see that by lowering

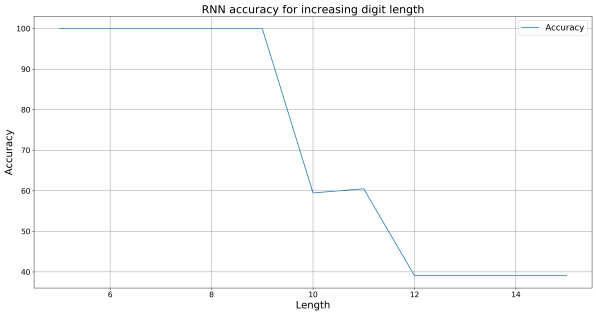
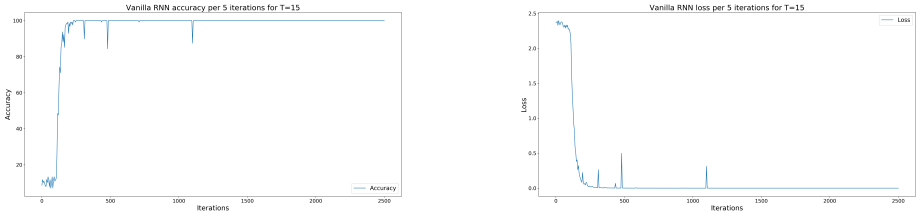


Figure 2: Accuracy of vanilla RNN for increasing length of the palindromes (default parameters)

the learning rate, the model is able to converge fast enough. However, for larger input lengths it was very difficult for the model to converge, due to the fact that the gradients are vanishing and hence the model stops learning.



(a) Accuracy

(b) Loss

Figure 3: Accuracy and loss of vanilla RNN for $T = 15$ and learning rate 0.001

It must be noted that we tried to input the palindromes to the model both as raw digits and one hot vectors. It was found that using one hot vectors makes the model to converge much faster than raw digits. The reason is the following. Ultimately, what we want from our model is to realize that it must memorize the first digit (first state) in order to be able to predict the last one. If we insert the inputs as raw digits, the model must also learn that the order of the digits does not play a role, and due to this it takes more time to converge. On the other hand, by inserting the palindromes as one-hot, the distance between two arbitrary digits is fixed and thus, the model does not have to learn anything about the order of the numbers. This results to much faster convergence.

Question 1.4

Stochastic gradient descent is a popular algorithm for Neural Network optimization, however several variants have been proposed. Two important improvements in SGD have been achieved using *adaptive learning rate* and *momentum* and many state of the art optimizers make use of these improvements.

It has been found that setting a fixed learning rate before the start of the optimization is not ideal, since if the learning rate is too large, the algorithm might not converge, while if it is too small, the learning will be very slow. A better idea, is to use a learning rate η_t, which decreases with the number of iterations t. This leads to strong movements of the gradient in the first updates and smaller movements as the number of iterations increases. This makes the learning procedure much faster than the case of a fixed learning rate.

Another improvement in SGD has been achieved with the use of *momentum*. Using momentum, the update rule of the weights becomes:

$$u^{t+1} = \gamma u^t - \eta_t \nabla_w \mathcal{L}$$
$$w^{t+1} = w^t + u^{t+1}$$

This means that by using momentum, the algorithm remembers the update u^t at each iteration and the next update u^{t+1} is determined as a linear combination of the previous update u^t and the gradient $\nabla_w \mathcal{L}$. While vanilla SGD tends to fluctuate much, SGD with momentum tends to move towards the same direction and in this way oscillations are prevented. This leads to more robust gradients and learning and hence to faster convergence.

Most of the state of the art optimizers also use another improvement namely, *adaptive learning rate*. In vanilla SGD all the weights are updated with the same learning rate. However, some of the weights might be nearly optimal, while other weights might need more tuning. This can be achieved by a learning rate that is adapted for each of the parameters. An optimization algorithm that makes use of adaptive learning rates is the RMSProp optimizer for which the update rule is as follows:

$$r = a \sum_{\tau=1}^{t-1} (\nabla_w^\tau \mathcal{L}_j)^2 + (1-a) \nabla_w^\tau \mathcal{L}_j$$
$$w^{t+1} = w^t - \eta_t \frac{\nabla_w \mathcal{L}}{\sqrt{r} + \epsilon}$$

where ϵ is a small number to avoid division by 0. The main idea of this algorithm is to divide the learning rate for a weight by a moving average of the squared gradients for that weight. The square rooting of the gradients boosts the small gradients and thus the updates get more aggressive. Also, the square rooting suppresses the large gradients and this prevents the updates from being too aggressive.

Another successful optimization algorithm is Adam, which makes use of both adaptive learning rate and momentum. The updates for Adam are as follows:

$$g_t = \nabla_w \mathcal{L}$$
$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$
$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$
$$\hat{v}_t = \frac{v_t}{1-\beta_2^t}$$
$$w^{t+1} = w^t - \eta_t \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

As can be seen from the formulas for $\hat{m}_t$ and $\hat{v}_t$, Adam uses moving averages for both the gradients and the second moments of the gradients. We note that β_1 and β_2 are the exponential decay rates for the first and second moments respectively.

Question 1.5a

LSTMs tackle the problem of vanishing gradients by introducing gating mechanisms. These mechanisms are able to make the RNN capture long-term dependencies. The gates of an LSTM are able to add or remove information from the cell state. The next paragraphs provide an explanation for the purpose of each gate and its corresponding non-linearity.

The *forget* gate f^t allows the LSTM to decide what information to forget, or throw away from the cell state. In this way, the RNN does not need to remember everything from the past states, but keeps only the information that is relevant. To achieve that, the forget gate has a sigmoid non-linearity which outputs a number between 0 and 1 for each number in the cell state c_{t-1}. The sigmoid works

like a 'switch', since an output of 1 means 'let all the information through', while an output of 0 means 'let no information through'.

Except of deciding what information to throw, the network must also decide what information to add to the new memory. This is achieved by the *input* gate i^t and the *input modulation* gate g^t. The input gate, decides what information we must store in the cell state. To achieve that, it uses again a sigmoid non-linearity, which outputs values between 0 and 1 in order to determine how much of the information we are going to store.

The input modulation gate g^t creates a vector of new candidate values that could be added to the state. This is achieved by using a tanh non-linearity which pushes the values between -1 and 1. The reason why tanh and sigmoids are used in the LSTM instead of ReLU's, is because ReLU is unbounded for values larger than 0. This means that after many iterations, the activation might return very large values which will affect negatively the model's performance.

By using the three aforementioned gates we can update the old cell state c_{t-1} into the new cell state c_t. To do so, we multiply (element-wise) the input modulation gate with the input gate and we add the product of the old cell state with the forget gate. Hence, we update the new cell state by discarding from the old state the irrelevant information (through the forget gate) and by including relevant information from the new candidate values (through the input and the input modulation gates).

The last step is to decide which information to output. This task is achieved by the *output* gate o^t which uses a sigmoid non-linearity. In this way, it can decide how much information it should output. Again, a sigmoid 1 means 'output everything' while a sigmoid 0 means 'no output'. After this, a tanh non-linearity is applied to the cell state, in order to produce numbers between -1 and 1, and it is then multiplied by the output gate, so as to output only the parts that the output gate has decided to output.

Question 1.5b

An input sample has size $T \times d$ where T is the sequence length and d is the input dimensionality. We feed in the algorithm the input sample for each time-step recursively, so a single input at time-step t is a vector of size $d \times 1$. If we feed a batch of size m, then the input at time-step t has size $d \times m$. The 4 weights between input and hidden layer ($W_{gx}, W_{ix}, W_{fx}, W_{ox}$) have sizes $n \times d$, where n is the number of units in the LSTM. The 4 *hidden to hidden* weights ($W_{gh}, W_{ih}, W_{fh}, W_{oh}$) have sizes $n \times n$. Also, the 4 biases in the hidden layers (b_g, b_i, b_f, b_o) have sizes $1 \times n$. Since it is not clear from the question if we should also include the weights and biases for the output layer, we consider 2 cases.

1. By not taking into account the output weights and biases, the number of total trainable parameters is:

$$\text{Total Number Params} = 4 \cdot n \cdot d + 4 \cdot n \cdot n + 4 \cdot n = 4n(d + n + 1)$$

2. If we take into account the output weights and biases (weights have size $n \times c$, biases have size $1 \times c$, where c is the number of classes), the number of total trainable parameters is:

$$\text{Total Number Params} = 4 \cdot n \cdot d + 4 \cdot n \cdot n + 4 \cdot n + n \cdot c + 1 \cdot c = 4n(d + n + 1) + nc + c$$

Question 1.6

The second modelling task is to create an LSTM in Tensorflow, without relying in functions that create it automatically. As in the vanilla RNN case, we first initialize weights, biases and initial states. Then, we make the inputs one-hot, and we insert them in `tf.scan`, along with the `lstm_step` function, which is used to compute the LSTM gates and hidden states. Then, we use the hidden states $h^{(t)}$ returned by `tf.scan`, to compute the outputs $p^{(t)}$. By taking the output for the last timestep, we can find the logits, which we use to compute the loss and the accuracy of the model. The graph of the LSTM is almost identical with the graph of the vanilla RNN as it was shown in Figure 1.

After successfully implementing the LSTM, we perform the same experiment as with the vanilla RNN. Using the default parameters, we let the model run for 2500 steps and for increasing input

length T, and we record its accuracy every 5 steps. In the end we take a mean of the accuracy for the last 250 steps for each T and we plot the results in Figure 4.

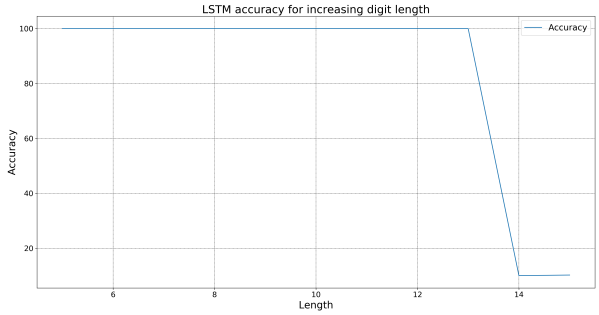
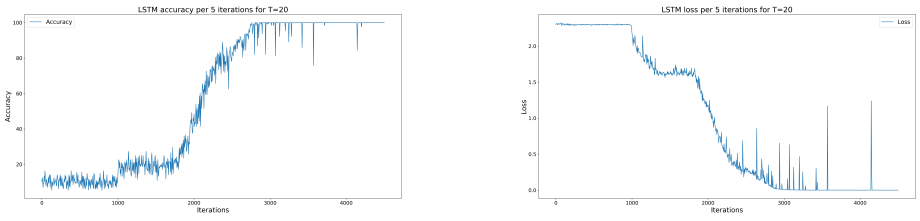


Figure 4: Accuracy of LSTM for increasing length of the palindromes (default parameters)

In Figure 4 we see that the LSTM with the default parameters achieves 100% accuracy until $T = 13$ and then the accuracy drops largely. By comparing this figure with Figure 2, we see that while the accuracy of the vanilla RNN drops for lengths 10-13, the accuracy of the LSTM is 100% for these palindrome lengths. This shows that the LSTMs are much better in memorizing long-term dependencies. This better performance, stems from the fact that the LSTMs do not suffer from the vanishing gradient problem, as opposed to the vanilla RNNs. As we discussed in Question 1.1, the gradients in the deeper layers are computed as products of differentials. This means that if we train the model for many timesteps, the gradients may vanish or explode, depending on whether the differentials are less or more than 1. On the other hand, the LSTMs use gating mechanisms to control how much information will be stored in each timestep. This allows the network to overcome the vanishing and exploding gradient problems, since it is able to store only the necessary information at each timestep. Hence, an LSTM can memorize long-term dependencies much better than the vanilla RNN.

Next, we tried to tune the LSTM so as to converge for larger palindrome lengths. It was found that by gradually decreasing the learning rate, the LSTM is able to achieve 100% accuracy for increasing input length. As an example, in Figures 5a and 5b we show the accuracy and the loss of LSTM for $T = 20$. The model which was used to create these plots, has a learning rate 0.0007, 130 hidden nodes and runs for 4500 steps, while the rest of the parameters are kept as default. In Figure 5a, we see that the accuracy is low for the first 1000 steps, then it increases slightly for the next 1000 steps, and then it shows a sharper increase between 2000 and 3000 steps. Finally, the model converges at around 3000 steps. A similar (but opposite) behaviour is also observed for the loss in Figure 5b. We can observe some spikes in both loss and accuracy plots, due to the fact that we measured them every 5 training steps (hence more fluctuations) and also because we generate a different test set every time that we measure the accuracy and the loss.



(a) Accuracy

(b) Loss

Figure 5: Accuracy and loss of LSTM for $T = 20$, learning rate 0.0007, 130 hidden nodes.

During the tuning process, we experimented with some of the model's hyper-parameters. We found that decreasing the learning rate can make the model converge for larger palindrome lengths. However, after some point (of increasing T), one must increase the number of hidden nodes in order to

achieve convergence. This is because when we increase the number of hidden nodes, we increase the capacity of the model and hence the model can approximate more complicated functions. However, increasing the hidden nodes, also increases the time and memory that the model consumes and hence the model must run for more steps in order to converge. We also tried to increase the number of the batch size, however, no significant improvements were observed. Finally, by adding more layers to the model, we change its architecture and hence we need to re-tune all the other hyper-parameters.

2 Recurrent Nets as Generative Model

Question 2.1

In this task we implement a two-layer LSTM network to generate sentences, by predicting the next character in a sentence. This is achieved by training the network on example sentences from a book. The first goal is to generate sentences of length $T = 30$ and then we will extend the network to allow for longer sentences.

We begin by creating the LSTM cells. We use the `LSTMCell` function to create an LSTM cell, then we use the `MultiRNNCell` routine to stack the LSTM cells sequentially and finally we use the `DropoutWrapper` to allow for dropout in the inputs of the cells. Furthermore, we create the weights and biases for the output and we initialize them as in part 1. Next, we turn the inputs to one hot vectors and we use the `dynamic_rnn` routine to unfold the network for all the timesteps. By doing this we obtain the outputs of model for all the timesteps and we use them to obtain the logits for each timestep. We then use the logits to compute the total loss, the normalized per-step probabilities and the predictions for each timestep. For the predictions, we find the digit with the highest probability (logit) to be next, using `tf.argmax`. Alternatively, we can use the logits to sample from the distribution using for example `tf.multinomial` or an equivalent function.

To train the model, we pass the loss to the RMSProp optimizer. Although the given train function has the gradient clipping by default, we note that this might be redundant since the LSTMs do not suffer from the exploding gradient problem as explained earlier. While we train the model, we test its performance by generating sentences every 100 iterations. To achieve that, we first generate a random character from the book's vocabulary and we put it as input in the model to make a prediction for the next character. For this first prediction, it is important to initialize the model. When we have the first prediction, we pass it as input again in the model, but this time we also pass the final state. By doing this iteratively, we generate a sentence of length $T = 30$. The graph of the text generation model in Tensorboard is given in Figure 6.

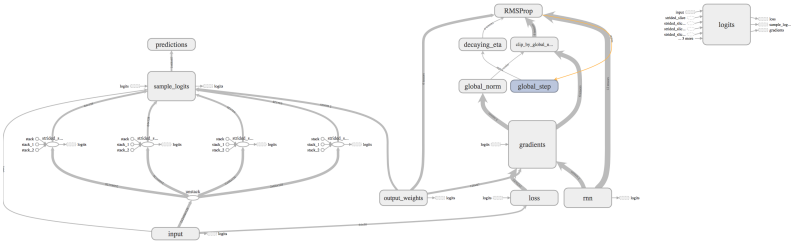


Figure 6: Graph of the text generator in Tensorboard

In order to test the model's capabilities, we generate text samples almost every 3500 iterations. Table 1 depicts 6 samples at different training steps. The first column shows the iteration in which the sample was taken, the second column is the random input character, the third column is the generated sentence and the last column is the total loss of the network at the corresponding training stage.

As we see in the table, in the first iteration the loss is high and the model produces characters without spaces. During the training process, we observed that in the first iterations the model produced only spaces, as it is probably the most common character. Then it started producing the word 'the' sequentially, as it is probably the most common word in this text. As the model evolves, it starts combining different words together so as to create sentences. After 3600 iterations the loss has

Table 1: Generated sentences of length 30, every 3500 training iterations, by training in *Grimm's Fairy Tales*.

Iteration	Input Character	Generated Sentence	Loss
0	s	aBBB(8VVVG$$$$$$$$$$$$$h$\$uuuu	4.46
3600	B	ut the second son said: 'I hav	1.24
7001	L	ittle Red-Cap, and the soldier	1.03
10601	Q	ueen, and the king said, 'I wi	0.99
14101	.	The little tailor was always	0.97
17401	?	' 'Alas! alas!' said the old w	0.94

dropped at 1.24 and the model is able to create sentences that make sense, as we see in the second row of the table. In this example we see that the model is already able to produce quotes. However we see that at this stage, the model creates sentences using only common words. At iteration 7001, the loss has dropped at 1.03 and the model is now able to use names (Little Red Cap) and also makes effective use of commas. At iteration 14101, the loss is 0.99 and the model is now able to use words which are relevant to the context. For example we see that given the letter 'Q', the model produces the word 'Queen' and then uses the word 'king' which is a related word in this context. This shows that the model uses all the previous states in order to make the predictions of the next letters. At iteration 14101 the loss is 0.97 and now the model is able to handle successfully characters such as the full stop '.'. We see that it correctly inserts a space after the full stop and then the next letter is a capital 'T'. At iteration 17401, the loss has dropped to 0.94 and we see that now the model is able to use the language in a more advanced way. In this last example, the model is given a question mark as input, so it predicts that it is the end of a quote (a quote that posed a question). Then, it answers with another quote in which also uses exclamation marks '!' correctly.

After this experiment, we see that by using a two-layer LSTM architecture we can make a network learn to use the English language from scratch. By observing its training process, we see how the model evolves its way of using the language by exploiting the training examples. The next step, is to extend the model to allow for longer sentence generation.

Bonus Question

Since our model is not able to memorize more than 30 subsequent timesteps, we can make it generate sentences longer than length 30 by doing the following. After unrolling the network for 30 timesteps, we save its final hidden and memory states, and we pass them again as inputs for the next 30 characters. This can be easily implemented using an 'if else' statement. After implementing this extension to our model we train it in the *Democracy in the United States*, using $T = 50$, 120k iterations and the rest of the parameters as default. In Figure 7 it is depicted the loss of the model during the training process. We observe that in the first iterations the loss decreases sharply, then the decrease becomes slower and after 60k iterations the loss remains almost constant around 1.2. We observe that the model is not able to lower its loss after some point, which means that we should tune its parameters further. We mention that we also tried to add dropout in the model, however the loss dropped until 1.5 in this case.

In table 2 we show two example sentences of length 50, generated with the model after training for 115k iterations. We see that even after so many iterations, the model mistakenly uses plural (in persons) when referring to the president. Moreover, we found that most of the generated sentences were repeating the same words (mostly United States, Union, Americans, etc.), indicating that the model needs further tuning so as to produce rich sentences of length 50.

Table 2: Generated sentences of length 50, after training in *Democracy in the United States* for 115k iterations.

Input Character	Generated Sentence
I	n the United States the provincial power of the Un
P	resident of the United States is the same persons

The last challenge for our model is to make it finish sentences. To achieve this, we do the following procedure. Firstly, we create a 'txt' file with the 4 sentences that our model must finish. We turn

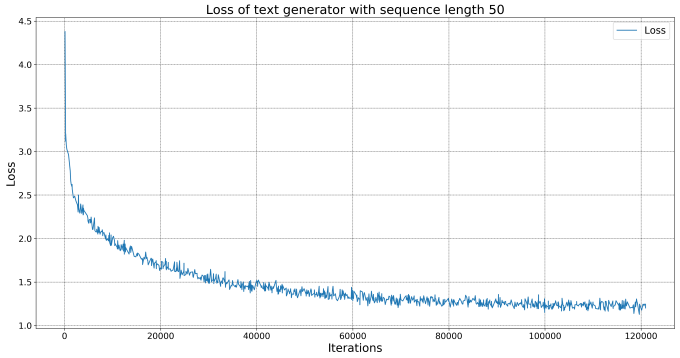


Figure 7: Loss of the text generator with $T = 50$ and training in *Democracy in the United States* for 120k iterations.

the sentences into one hot vectors so as to insert them as inputs to our model when testing. While training, we sample example sentences from the model every 100 iterations as follows. We give a sentence, letter by letter, to the model as input and it predicts the next possible character for each letter. The last prediction of the model is the 1st letter of our generated sentence. We now feed the model with this prediction and also with the final state as returned from the first prediction. By doing this iteratively, we can generate sentences. However, we must store the final state of the model every 30 iterations, and feed it back to the model for the next 30 iterations. The core part of this implementation can be found in the Appendix.

Using this procedure we train the model for 8900 iterations in the *Grimm's Fairy Tails* for sequence length 80. In table 3 we see example sentences generated after 9k iterations. In the first example, we feed the sentence 'The prince went to' and the model decides to use the word king, as it is closely related with the word prince. Moreover it correctly uses quotes and commas. The next example input sentence is 'The princess said:' in which the model understands that it must open quotes, since it usually happens in the text after the character ':'. The next two sentences are also grammatically correct, but semantically they still need improvement. This can be achieved by letting the model train for more iterations and also by further tuning.

Table 3: Generated finished sentences of length 80, after training in *Grimm's Fairy Tales* for 9k iterations.

Input Sentence	**Generated Sentence**
The prince went to	the king, and said: 'I have a beautiful princess in the world, and the soldier
The princess said:	'I can get out of the water, and then I will give you a way off the words, the
The forest was	all that the wild man said: I will not be a doctor of the window, and I will g
The night was dark	, and the soldier was so beautiful as the wolf had been so kind. 'What a beautif

3 Conclusion

In this study, we get familiar with Recursive Neural Networks which are suitable for sequential processing of data. In the first part we implemented a vanilla RNN and an LSTM and compared them in a simple toy problem. The most challenging task in this implementation, was to appropriately insert the inputs in `tf.scan`. By comparing the vanilla RNN and the LSTM, we see that the latter is able to memorize long-term dependencies, as it does not suffer from the problem of vanishing and exploding gradients, as in the case of the vanilla RNN. In the second part of this study, we use LSTMs as a generative model to generate text. The most challenging task in this part was to realize that when generating a new sentence, one must initialize the model only for the first prediction and then for the rest of the predictions, the final state of the model must iteratively be fed into the model. After implementing the text generator, we can follow the learning process by sampling sentences as the training evolves. By studying the example sentences, we can understand how the model

gradually uses more advanced language techniques and exploits the training examples in a larger extent.

Appendix

The core part of the code which makes the text generator finish sentences.

```
if train_step % sample_every == 0:
            #Reshape the inputs
            state1 = init_state_test
            state2 = init_state_test
            state3 = init_state_test
            state4 = init_state_test

            new_sentence1 = np.empty([seq_length])
            new_sentence2 = np.empty([seq_length])
            new_sentence3 = np.empty([seq_length])
            new_sentence4 = np.empty([seq_length])



            #Forward prediction
            #Feed each letter of the sentence one by one and predict the next one.
            for i in range(len(sentence_1)):
                pred1,state1 = sess.run([predictions1,final1], feed_dict =
                 {test_sentence1: np.reshape(test_input_1[i],(1,1)),state_test: state1})

            for i in range(len(sentence_2)):
                pred2,state2 = sess.run([predictions2,final2], feed_dict =
                 {test_sentence2: np.reshape(test_input_2[i],(1,1)),state_test: state2})

            for i in range(len(sentence_3)):
                pred3,state3 = sess.run([predictions3,final3], feed_dict =
                 {test_sentence3: np.reshape(test_input_3[i],(1,1)),state_test: state3})

            for i in range(len(sentence_4)):
                pred4,state4 = sess.run([predictions4,final4], feed_dict =
                 {test_sentence4: np.reshape(test_input_4[i],(1,1)),state_test: state4})

            #Prediction of 1st letter
            new_sentence1[0] = pred1[0,0]
            new_sentence2[0] = pred2[0,0]
            new_sentence3[0] = pred3[0,0]
            new_sentence4[0] = pred4[0,0]


            #Autoregression
            for i in range(seq_length-1):
                if (i < 30):
                    pred1, state1 = sess.run([predictions1,final1], feed_dict =
                     {test_sentence1: pred1,state_test: state1})
                    pred2, state2 = sess.run([predictions2,final2], feed_dict =
                     {test_sentence2: pred2,state_test: state2})
                    pred3, state3 = sess.run([predictions3,final3], feed_dict =
                     {test_sentence3: pred3,state_test: state3})
                    pred4, state4 = sess.run([predictions4,final4], feed_dict =
                     {test_sentence4: pred4,state_test: state4})

                    new_sentence1[i+1] = pred1[0,0]
```

```
            new_sentence2[i+1] = pred2[0,0]
            new_sentence3[i+1] = pred3[0,0]
            new_sentence4[i+1] = pred4[0,0]

        elif (i >= 30 & i < 60):
            pred1, state1 = sess.run([predictions1,final1], feed_dict =
             {test_sentence1: pred1,state_test: state1})
            pred2, state2 = sess.run([predictions2,final2], feed_dict =
             {test_sentence2: pred2,state_test: state2})
            pred3, state3 = sess.run([predictions3,final3], feed_dict =
             {test_sentence3: pred3,state_test: state3})
            pred4, state4 = sess.run([predictions4,final4], feed_dict =
             {test_sentence4: pred4,state_test: state4})

            new_sentence1[i+1] = pred1[0,0]
            new_sentence2[i+1] = pred2[0,0]
            new_sentence3[i+1] = pred3[0,0]
            new_sentence4[i+1] = pred4[0,0]
        else:
            pred1, state1 = sess.run([predictions1,final1], feed_dict =
             {test_sentence1: pred1,state_test: state1})
            pred2, state2 = sess.run([predictions2,final2], feed_dict =
             {test_sentence2: pred2,state_test: state2})
            pred3, state3 = sess.run([predictions3,final3], feed_dict =
             {test_sentence3: pred3,state_test: state3})
            pred4, state4 = sess.run([predictions4,final4], feed_dict =
             {test_sentence4: pred4,state_test: state4})

            new_sentence1[i+1] = pred1[0,0]
            new_sentence2[i+1] = pred2[0,0]
            new_sentence3[i+1] = pred3[0,0]
            new_sentence4[i+1] = pred4[0,0]

    #for idx, elem in enumerate(new_sentence):
    print('Sentence 1:{} {}'.format(sentence_1+str('...'),
    text.convert_to_string(new_sentence1)))
    print('Sentence 2:{} {}'.format(sentence_2+str('...'),
    text.convert_to_string(new_sentence2)))
    print('Sentence 3:{} {}'.format(sentence_3+str('...'),
    text.convert_to_string(new_sentence3)))
    print('Sentence 4:{} {}'.format(sentence_4+str('...'),
    text.convert_to_string(new_sentence4)))
```