Deep Learning - Assignment II

Albert Harkema - 12854794
Faculty of Natural Sciences, Mathematics and Computer Science
University of Amsterdam
Amsterdam
`albertharkema@gmail.com`

Abstract

In this paper a discussion follows of several recurrent and graph neural network implementations. Firstly, the performance of a Vanilla RNN compared to an LSTM network on a palindrome prediction task is discussed, along with the specifics of the optimization algorithms of RMSprop and Adam. Secondly, the effectiveness as recurrent nets as generative models is discussed, in which a number of LSTM networks are used on a variety of books to learn to generate new sentences. Lastly, a discussion follows of Graph Neural Networks, including their applications, their drawbacks and their performance compared to RNNs on specific tasks.

1 Vanilla RNN versus LSTM

1.1 Toy Problem: Palindrome Numbers

1.2 Vanilla RNN in PyTorch

Question 1.1
First we will find $\frac{\partial \mathcal{L}^t}{\partial W_{ph}}$:

$$\frac{\partial \mathcal{L}^t}{\partial W_{ph}} = \frac{\partial \mathcal{L}^t}{\partial \hat{\boldsymbol{y}}^t} \frac{\partial \hat{\boldsymbol{y}}^t}{\partial \boldsymbol{p}^t} \frac{\partial \boldsymbol{p}^t}{\partial W_{ph}} \tag{1}$$

We can find the individual partial derivatives from equation (1) as follows:

$$\frac{\mathcal{L}^t}{\partial \hat{\boldsymbol{y}}_i^t} = -\frac{\boldsymbol{y}_i^t}{\hat{\boldsymbol{y}}_i^t} \tag{2}$$

Afterwards we get the derivative of a vector with respect to a vector for $\frac{\partial \hat{\boldsymbol{y}}^t}{\partial \boldsymbol{p}^t}$, which will result in a matrix so we'll have two different indicators i and j:

$$\frac{\hat{\boldsymbol{y}}^t}{\boldsymbol{p}^t} = \frac{\partial softmax(\boldsymbol{p}^t)}{\partial \boldsymbol{p}^t} \tag{3}$$

$$= \frac{\partial softmax(\boldsymbol{p}_i^t)}{\partial \boldsymbol{p}_j^t} \tag{4}$$

$$= \hat{\boldsymbol{y}}_i^t(\delta_{ij} - \hat{\boldsymbol{y}}_j^t) \tag{5}$$

where δ is the kronecker delta. This derivative is similar to the derivative for the softmax we derived in assignment 1 of Deep Learning.

The third partial derivative $\frac{\partial \boldsymbol{p}^t}{\partial W_{ph}}$ can be found to be:

$$\frac{\partial \boldsymbol{p}^t}{\partial W_{ph}} = \boldsymbol{h}^{t^T} \tag{6}$$

Now we have all the derivatives we need. However, $\frac{\partial \mathcal{L}^t}{\partial \boldsymbol{p}_i^t}$ can be further simplified with a bit of algebra to just:

$$\hat{\boldsymbol{y}}_j^t - \boldsymbol{y}_j^t \tag{7}$$

Giving us the following total derivative:

$$\frac{\partial \mathcal{L}^t}{\partial W_{ph}} = (\hat{\boldsymbol{y}}_j^t - \boldsymbol{y}_j^t)\boldsymbol{h}^{t^T} \tag{8}$$

For the second part of this question we have to find $\frac{\partial \mathcal{L}^t}{\partial W_{hh}}$ which can be expressed as:

$$\frac{\partial \mathcal{L}^t}{\partial \hat{\boldsymbol{y}}^t}\frac{\partial \hat{\boldsymbol{y}}^t}{\partial \boldsymbol{p}^t}\frac{\partial \boldsymbol{p}^t}{\partial h^t}\frac{\partial h^t}{\partial q^t}\frac{\partial q^t}{\partial W_{hh}^t} \tag{9}$$

where

$$q = W_{hx}x^t + W_{hh}h^{t-1} + b_h \tag{10}$$

We have already found the first two factors of Equation (9), so our job is to find the remaining three factors, starting with $\frac{\partial \boldsymbol{p}^t}{\partial h^t}$, which is the derivative of a vector with respect to a vector so we will end up with a matrix:

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

Secondly, we can derive the following:

$$\frac{\partial h^t}{\partial q^t} = 1 - (h^t)^2 \tag{12}$$

And the last derivative we have to find is:

$$\frac{\partial q^t}{\partial W_{hh}} \tag{13}$$

which will be (something) $\cdot h^{t-1}$ + $(W_{hh}) \cdot \frac{\partial h^{t-1}}{\partial W_{hh}}$.

The important takeaway from this last gradient is that it will consist of the previous hidden state h^{t-1} and the matrix W_{hh}. This recursive definition means that we're multiplying by the matrix W_{hh} through all the timesteps. Generally, the matrix will be initialized with very small numbers. Multiplying very small numbers together a large number of times results in vanishing gradients, which is a big problem for RNNs in particular because of the large number of timesteps they iterate over. In case W ends up being large enough, it could also cause our gradients to explode for the same reasons as mentioned before.

Question 1.2
Implementation in vanilla_rnn.py

Question 1.3
Figure 1 plots the sequence length on the x-axis and the accuracy on the y-axis. A general trend becomes visible where the accuracy goes down as the sequence length goes up, although there appears to be an element of stochasticity because for a sequence length of 14 the accuracy goes up again only for it to go down for a sequence length of 16. This can be explained by the fact that a lucky initialization may be very important for LSTMs. The initialization may have been rather fortunate for the sequence lengths of 14 and 18.

Furthermore, Figure 2 also plots the sequence length on the x-axis and the accuracy on the y-axis. As we expected, the trend continues for the even longer sequences than the ones plotted in Figure 1. For sequences of over a length of 20 we converge at around 20% accuracy, and at a sequence length of 80 we hardly perform better than if we would do random guessing (10% accuracy at 10 options for the numbers 0 - 9). Clearly, the longer the sequence, the more difficulty an RNN has in retaining the memory and using it at the right time. For this particular reason, LSTMs were invented. Can the

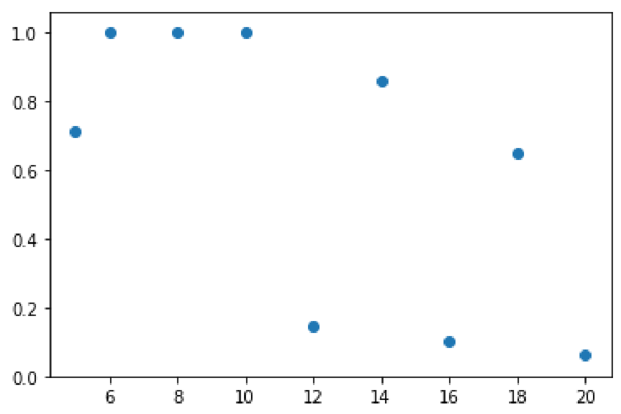

Figure 1: Accuracy (y-axis) and sequence length (x-axis)

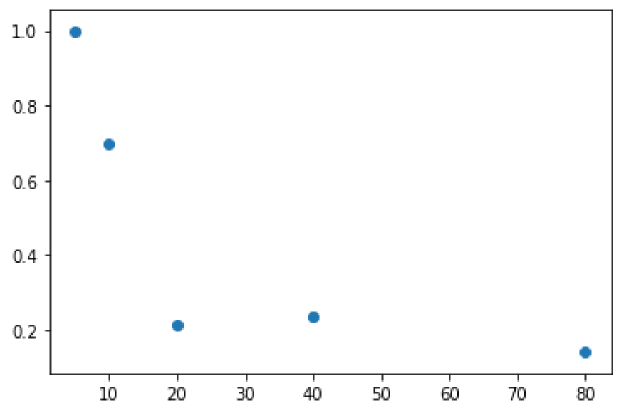

Figure 2: Accuracy (y-axis) and sequence length (x-axis)

introduction of multiple gates help in counteracting the memory loss over longer sequences? We'll find out in question 1.6!

Question 1.4

In this section an analysis follows of the benefits of RMSprop and Adam compared to vanilla stochastic gradient descent. First, we'll take a look into vanilla stochastic gradient descent. Subsequently, a discussion follows of RMSprop and Adam and their possible improvements on vanilla SGD, including their usage of momentum and adaptive learning rate.

Vanilla Stochastic Gradient Descent (mini-batch) calculates the gradient for a mini-batch and takes a step in the direction of this gradient. The goal of this process is to use the gradient to make updates to our parameters in an informed fashion to minimize the loss of our model. Intuitively, this makes a lot of sense: the gradient tells us the direction of the slope, and taking a step in this direction for infinitesimally small steps will ultimately help us converge to a configuration of our parameters that will minimize the loss function.

However, many problems arise with this, among which is the problem of pathological curvature as depicted in Figure 3. Here, the ideal path is the path down the ravine to obtain the lowest loss score the quickest but vanilla gradient descent will mostly step in the w1 direction where the local gradient is the steepest. This poses a challenge for our highly non-convex high-dimensional minimization problem. Apparently, just using the local gradient at timestep t won't always provide us with the best direction for descending to the lowest loss.

And this is where **momentum** steps in. Momentum ensures we do not switch update directions all the time by incorporating "momentum" from previous timesteps. By using exponential averaging of the gradient over multiple timesteps, momentum is able to use the knowledge we have over multiple timesteps to retrieve the direction that is more similar to the actual steepest descent. This

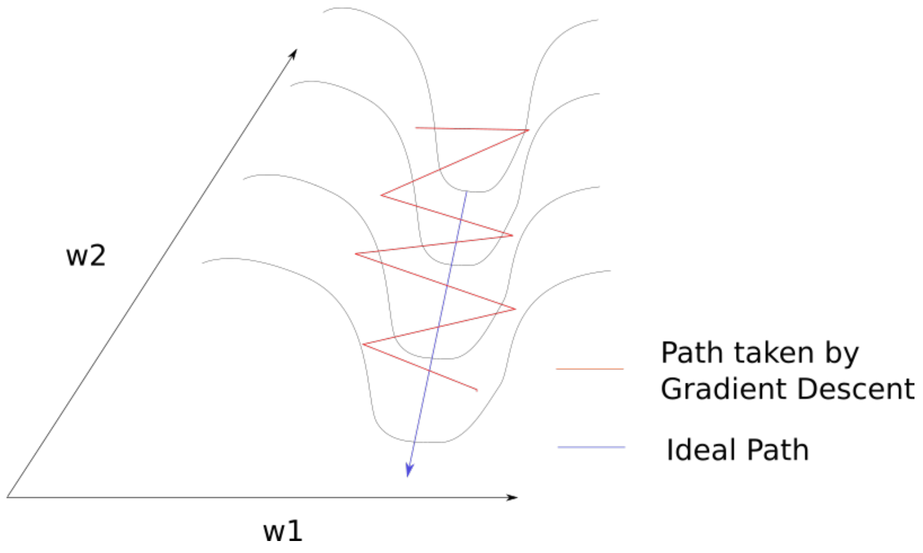

Figure 3: Pathological curvature, picture credit: https://blog.paperspace.com/intro-to-optimization-momentum-rmsprop-adam/

exponential averaging dampens the oscillations because it is weighted by the gradient we had at previous timesteps, causing the path we take to more resemble the ideal path as depicted in Figure 3.

Following from this, RMSprop uses a momentum-like term (albeit quite different in the way it goes about achieving its goal) in its optimizaton procedure and combines it with an **adaptive learning rate**. RMSprop uses the following three equations in its algorithm:

$$r_t = \alpha r_{t-1} + (1-\alpha)g_t^2 \tag{14}$$

$$u_t = -\frac{\eta}{\sqrt{r_t}+\epsilon}g_t \tag{15}$$

$$w_{t+1} = w_t + u_t \tag{16}$$

where α is a decay hyperparameter. From equation (14) it follows that r_t is a parameter that incorporates α times its previous value + $(1-\alpha)$ times the square of the new gradient. This is an adjusted momentum term, incorporating the scaled values of the gradients at previous timesteps. Whereas momentum would cause the gradients of the w1 direction in Figure (3) to cancel out, RMSprop will get very large values for the average of the w1 gradient, resulting in a lower learning rate for w1. This adaptive learning rate follows from equation (15). RMSprop uses the r_t term from Equation (14) to scale the learning rate, turning it into an **adaptive learning rate**, which means that we adjust our learning rate for different timesteps based on the size of the respective gradients for each parameter.

RMSprop achieves this by dividing the learning rate by the square root of r_t. r_t will be a larger value for larger gradients, causing the learning rate to be divided by a larger number, in turn resulting in a smaller value that's multiplied with the gradient at the current timestep as seen in Equation (15). This ensures that we suppress very large gradients in order to prevent overshooting. Contrarily, small gradients get a boost from this r_t term causing the updates to be a bit stronger, aiding in escaping plateaus where the gradient is really small. Ultimately, in Equation (16) we see the update step we know from vanilla SGD, where we update our weights with a term that incorporates the gradient at this time step.

Adam is another popular optimizer that combines RMSprop with momentum and a correction bias. It is quite similar to RMSprop but it also incorporates an aforementioned correction bias. For Adam we calculate the exponential average of the gradients. Subsequently, the learning rate is multiplied by this average which corresponds to the way momentum does this. Moreover, like RMSprop, we divide the total by the root mean square of the exponential average of squared gradients, preventing us from bouncing around too much. Additionally, Adam adds a bias correction term. This is most helpful at the early stages because then our estimates for V will be off by a large margin. Lower values of t make sure we divide by a smaller term, causing our estimate for v_t to be curved to the right value. The more steps we've taken, the more accurate our estimation will be and the less we have to correct the (barely existing) bias.

1.3 Long-Short Term Network (LSTM) in PyTorch

Question 1.5

a) f^t is the so-called forget gate, which consists of a sigmoid over the standard input of x related weights times x, h related weights times h and a bias. The task of the forget gate is to model when to forget or remember a certain combination of the previous cell's output and the input at the current cell. For this reason, the sigmoid is the natural non-linearity, because our objective is to model a scale of either completely remembering or completely forgetting the previous memory state. The sigmoid enables us to do so by squashing the outcome values between 0 and 1. Consequently, the forget gate is multiplied with the incoming memory state and models the extent to which we remember or forget this memory state.

   i^t is the so-called input gate, which consists of a sigmoid over the standard input of x related weights times x, h related weights times h and a bias. The task of i^t is to model which values we want to update in the cell state. Because we will multiply this with the candidate values g^t, we want to express the i^t values on a scale of 0 to 1 for how much of the candidate state we want to use to update our cell state. The sigmoid proves to be well suited for this, because it squashes its input between 0 and 1.

   g^t is our input modulation gate which will be our candidate values to potentially update our cell state with. This is done by doing a tanh non-linearity over the standard input of the respective weights over the previous cell's output and current cell's input. The reason why this a tanh is used for this purpose appears to be less clear-cut than why sigmoids are used. One possible explanation would be that an advantage of the tanh over sigmoid is the fact that tanh is centered around 0. This would be beneficial, because the result of our input modulation gate is fed into the cell's memory state which is used and manipulated through a very long sequence, in which the centering around 0 might be beneficial. Another explanation may be that around the time LSTMs were first proposed, tanh's were the de facto standard non-linearities used in neural networks and were used in LSTMs because of this reason.

   o^t is the so-called output gate. This gate is tasked with modeling what parts of our cell state we decide to output. This can subsequently be used as information for the next lstm cell, e.g. whether this cell contained a singular or plural noun allowing for proper conjugation. Again, the sigmoid is the natural non-linearity because we want to use this gate as a way to scale how much we want to output: 0 for completely not outputting the cell state, 1 for completely letting the cell state through and anything in-between to capture the extent to which we want to output the cell state.

b) We have that n is the number of hidden units in the LSTM and d is the dimensionality of the features. In the introduction for part 1.3 it is mentioned that the core part of the LSTM consists of the first 6 equations mentioned, of which the first 4 consist of the learnable parameters. These four have exactly the same input (in terms of dimensionality, not in terms of the actual learned weights). This consists of hidden to input matrix weights ($d \times n$), hidden to hidden matrix weights ($n \times n$) and the bias weights ($hidden \times 1$) which we have a total of 4 times. This can be written as:

$$= 4 * (n^2 + dn + n) \tag{17}$$
$$= 4n(n + d + 1) \tag{18}$$

which would be the total number of trainable parameters for the core part of the LSTM. If we want to include the W_{ph} and corresponding b_p bias term, we get another value c for the number of output classes. The weights will have dimensionality $c \times n$ and the bias term b_p will have dimensionality $c \times 1$ to get a total of:

$$c \times (n + 1) \tag{19}$$

So the total number of learnable parameters for the core part of the LSTM is $4n(n + d + 1)$ and optionally we could add the total number of learnable parameters from the linear output mapping, amounting to an additional $c(n + 1)$ term.

Question 1.6

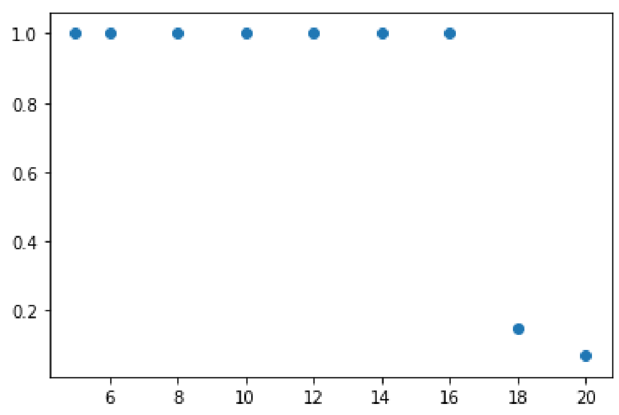

Figure 4: Accuracy (y-axis) and sequence length (x-axis)

In Figure 4 the accuracy has been plotted against the sequence length for the LSTM implementation. The LSTM is able to consistently get an accuracy of 100% up until sequence lengths of 18, after which it drops rapidly to about chance for sequence lengths of 20 or longer. Even though the LSTM is a more intricate model that uses specialized gates which should allow it to capture longer term dependencies than the vanilla RNN, merely implementing it this way does not seem to allow the LSTM to actually do so.

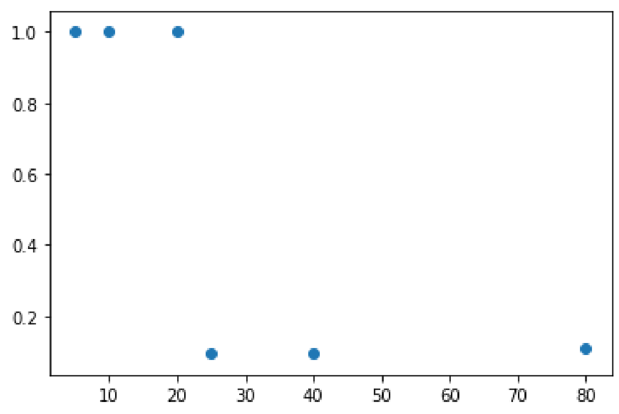

Figure 5: Accuracy (y-axis) and sequence length (x-axis)

This is further corroborated by the findings depicted in Figure 5. Again, our LSTM is able to capture the relatively short-term dependencies up until sequence lengths of about 20, after which it drops dramatically to about chance for the longer sequence lengths. Apparently there is more to it than just adding the gates and assuming all is well now. Let's look into the specific hyperparameters that we used for Figures 4 and 5 and see if there are some that can be tweaked.

A number of hyperparameters were initialized in the previous experiments with the LSTM. The optimizer that was used is the standard RMSprop implementation from PyTorch with a learning rate of 0.01 to potentially allow for faster convergence on longer sequences. The number of hidden units and the batch size were left at their default values of 128, and the network was trained for 10000 steps. What tweaks to these hyperparameters could possibly ensure we get better results?

A very important tweak to be made is the initialization of the forget bias parameter. Previously, the forget bias was initialized as a zero vector. However, doing so may cause your network to be initialized in such a way that it immediately starts forgetting important information before it actually gets the chance to learn when it should and shouldn't forget information. Additional hyperparameters that could potentially improve the LSTM's performance are using a learning rate scheduler, and increasing the number of hidden units.

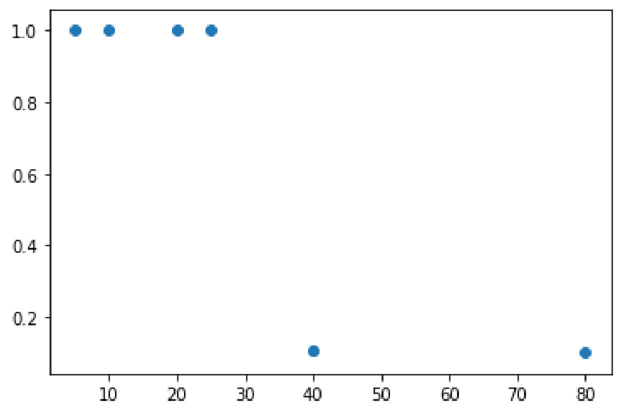

Figure 6: Accuracy (y-axis) and sequence length (x-axis) for Retrained LSTM with different hyperparameters

In Figure 6 the accuracy has been plotted for a variety of sequence lengths for an LSTM model with 256 hidden units and a forget bias that is now implemented as a vector of ones. As it turns out, the LSTM is now able to model sentences to up to at least a length of 25, which is quite an improvement on the previous model. Experiments with different parameters for the RNN did not lead to such an increase in improvement of the accuracy score, leading to the conclusion that LSTMs are indeed better suited to model long sequences. Tweaking its behaviour slightly, allows for reaping the full benefits of its more intricate model.

Namely, the forget gate is now able to learn how to model when to forget a certain cell state. Combined with the notion of the output gate, our model is now able to model more accurately how and when to forget or remember a certain state as opposed to the vanilla RNN. This allows it to model the palindromes in such a way, that it will know to remember the first character all the way through the sequence.

All in all, LSTMs have frequently been found to be able to model sequences of over 200 timesteps. A very important part of this may lie in one-hot encoding. This is the case because we don't want to model any non-existing inherent values of the actual numbers 0 - 86 which have no actual hierarchical, numerical order, which unfortunately happens in our current model. However, implementing one-hot encoding for the LSTM is beyond the scope of this part of the assignment.

Question 1.7

Figure 7 shows the vector norms on a log scale of the gradients throughout the time-steps of an LSTM network and an RNN network. This plot highlights the problem of vanishing gradients. Because we're multiplying very small numbers together so many times the gradient becomes extremely small near the first values for h_t. This poses a big problem for learning, especially at the earlier time-steps, as it becomes extremely difficult to learn when there is no gradient to begin with.

Moreover, it becomes apparent that even though the RNN was initialized with a far larger gradient norm, the gradient vanishes way faster than the LSTM's gradient. The LSTM is able to retain a gradient, albeit increasingly small, at time-steps very far away from the the last time-step. This corroborates the findings in question 1.6, where we found that LSTMs are able to model longer

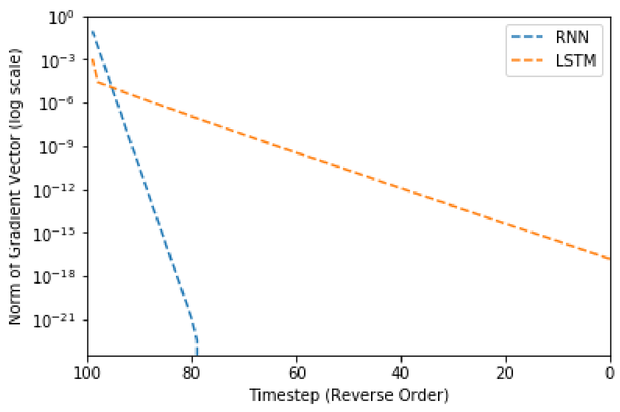

Figure 7: Plot of the gradient norms for LSTM and RNN

sequences better than RNNs, in part due to its capability of retaining a workable gradient over larger sequences.

Expected results if we would train a network for this task instead would be that the difference will be even more evident, as the LSTM will probably be able to retain a very large portion of its original gradient at the last time-step, whereas the RNN will keep having difficulty with this problem of vanishing gradients due to its less sophisticated nature which can't find a way to bring a workable gradient all the way back to the first time-step. These findings emphasize the importance of the gates and the cell state of the LSTM's architecture to retain a workable gradient throughout a large portion of very long sequences.

2 Recurrent Nets as Generative Model

Question 2.1

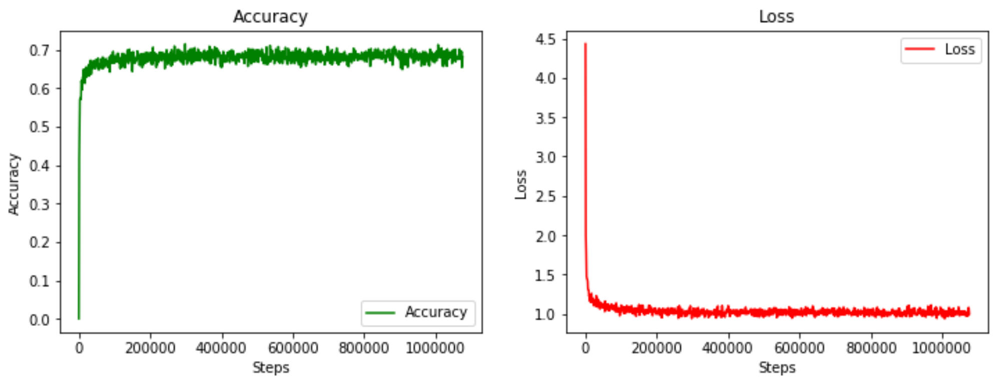

Figure 8: Accuracy and loss plots for 25 epochs on Democracy in the US

a) In Figure 8 the loss and accuracy during training have been plotted for the 2-layer LSTM model on the book "Democracy in the US". This book was chosen because it seemed that it might be slightly easier to model the more pragmatic language than its more story-telling counterparts. It becomes apparent that the loss and accuracy improve rapidly in the first 5000-10000 steps and then steadily converge at around 75,000 - 100,000 steps. After this, the model bounces around between an accuracy score of 65-70%. To train the model plotted in Figure 8, a variety of different parameter settings were tried. However, the following settings resulted in the best scores.

   The number of hidden layers per cell was chosen to be 128. Generally it's practical to initialize this as a multiple of 2, and 128 turned out to be just fine, because it allows for plenty of flexibility without overfitting. As required by the assignment, the number of layers was left at 2. The batch size was left to the default setting of 64, again a multiple of 2. We adhered to the rule of thumb that the batch size should be the largest multiple of 2 that fits nicely in a GPU's memory to allow for optimal use of parallelization. Moreover, because the amount of training steps is far larger than the amount of text we have in our book, we implemented an extra epoch parameter which was wrapped around the main for loop. In this way, we can go through the book more than once in the hopes of increasing the accuracy scores. As it turns out, multiple readings of the book were required to achieve the accuracy score it ultimately converged to.

b) In this subsection a discussion follows of the actual generated sentences over different stages of training. To start off, the first sentence that was generated by our model after the first iteration is the following:

$$\text{,aaSaaa aaa aa aaa aaa} \tag{20}$$

Obviously and unsurprisingly, this can't possibly be construed as an English sentence. The model has only seen a single sentence and it has no idea whatsoever of how to generate proper sentences in English yet. From Figure 8 we know that our model's accuracy rapidly increases in the first couple of thousand steps, so it may be interesting to keep track of the improvement of the validity of the sentences during this timeframe. After 5000 steps, the model generated the following sentence:

$$\text{quill the same subject of the} \tag{21}$$

After a mere 5000 steps, the model already starts to gain a minor understanding of the syntactic notion of English. The part 'the same subject' is a very coherent, albeit short, English sentence. However, it also becomes apparent that our model wants to include the word 'the' twice in the sentence. This relates to the fact that the word 'the' is the most common word in the English language, resulting in the fact that it has a very high probability to be sampled given a previous character, e.g. a space.

At 10,000 steps, the model generated the following sentence:

$$\text{. The power of the country in} \tag{22}$$

Apparently the model was fed a 'period' at the start of the sentence, after which it properly generated a space, because that is the only real probable character to follow a period, aside from floating point numbers. Unsurprisingly, the model generates the most common word in the English language again, after which it generates a grammatically correct sentence. The specific language used in the book is now expressed in the model, which starts talking about 'power' and a 'country'. The sentence constructed still consists of the same syntactic notion as sentence 21 where it tries to form sentences using 'the <noun> of the <noun>'. Will later timesteps yield more diverse sentences?

If we take a little leap to timestep 25,000, the following sentence is generated:

$$\text{Democratic republic in the Uni} \tag{23}$$

Which seems like a pretty cool improvement! For the first time the model has generated a sentence with an adjective in it: 'democratic', and it is properly connected to 'republic'. This is once again followed by a standard '<preposition> the' phrase, and the word 'uni' would've most likely been 'United States' had the model been granted more **freedom** regarding its sequence length.

Fast-forward to 65,000 steps when the model is relatively close to convergence, the model generates this sentence:

$$\text{He is a stranger the same time} \tag{24}$$

Although semantically not entirely something a native speaker would write, it embeds some interesting linguistic notions. For the first time, the model generates a complete sentence with the phrase 'he is a stranger' which could've also been followed by a period. The sentence consists of a pronoun, a verb and a noun phrase, constructing a full legitimate sentence. With a bit of imagination, the 'the same time' part, if preceded by a comma, could also be part of a longer, more poetic sentence. The model is quite close to generating grammatical, meaningful sentences!

Following from this, it could be interesting to sample a couple of sentences from when the model has basically converged for a while, e.g. after 102,000 steps, 307,000, 409,000 and 421,000 steps respectively :

he armed for them the politica (25)

"New York contests another. Bu (26)

Mississippi-share of the citiz (27)

Europeans are then ten years t (28)

All of these sentences, except for sentence 27 consist of a noun, followed by a verb phrase and then some. Clearly, the LSTM was able to model syntactic notions used in English, by the rules of which it is now able to more or less generate valid sentences. Sentence 27 does not follow the same structure as the other 3 sentences, but was included because it generated the awesome phrase 'mississippi-share of <something>', which is a correct phrase that many a native speaker would not even construct these days. All in all, it becomes apparent that with the used settings, the model is able to capture the syntactic structure of English on sentences of the sequence length it was trained for. However, it could be interesting to see what happens if we generate sequences that are longer or shorter than the sequence length it was trained for.

Change in output sequence lengths

Consequently, the model has been tested to generate sentences with a sequence length of 120, generating the first sequence after a 1000 steps:

ce the promer the promer the promer the promer the promer
the promer the promer the promer the promer the promer the pro (29)

which again can't possibly be construed as a valid English sentence due to its frequent repetition of the word 'the' and 'promer' which doesn't appear to be a word. The model appears to follow the same trend in improvement as for the sequences of length 30, so let's focus on sentences generated when the model has converged to allow for true analysis of the final performance on longer and shorter sequences.
Subsequently, after 405,000 steps, it generated the following sentence:

$1890 ways the same principles which are not always a series
of making an actions of the country in the United States (30)

and after 413,000 steps it generated:

790 the most permanent to the same state of things which they
are the same principles of the Union are the constitutiona (31)

It becomes apparent that these sentences form a repetition of sequences that resemble the sentences we uncovered for the sequence length of 30 on which it was trained. Even though at first glance the sentences could almost be construed as some creative legal language, it becomes clear fairly quickly that these are quite incoherent sentences constructed from relatively coherent smaller phrases, bringing about the conclusion that the generated 30-character sentences are more coherent.
Subsequently, the model has also been tested to generate sentences with a smaller sequence length of only 18. This resulted in the first sequence after a 1000 steps:

and the promer the (32)

Again it generates the non-existing word 'promer' and it appears to have a preference for the word 'the' again, which is not unsurprising given the short amount of time the model had to train. Moreover, also for these short sentences, it appears to converge in the same way as is the case for the previous sequence lengths, so it may be interesting to see outputs of later parts, such as at timestep 400,000:

Government which i (33)

which is the beginning of an almost correct sentence. Again, it becomes clear that the model is able to construct words with very few, if any, errors. The sequence length of 18 also allows for very little semantic expressiveness, so the actual sentences do not necessarily convey a lot of meaning. Again, when we look at a sentence from timestep 430,000:

$$\text{English and the pr} \tag{34}$$

We see that the model is capable of forming syntactically proper sentences. However, in part due to the very limited sequence length, the sentences generated are slightly less coherent than the sentences generated with a sequence length of 30. All in all, it becomes clear that the model from this part of the assignment is capable of generating syntactically correct sentences most of the time. For the sequence length on which it was trained, it quite often generates coherent sentences. However, it still has a long way to go if it wants to be a convincing writer that is capable of conveying true meaning in this world.

c) In the previous section we've seen a discussion of model's performance on a variety of sequence lengths in which we used greedy sampling. However, how does the temperature parameter τ affect the performance?

The effect of the temperature parameter τ on the sampling process depends on its value. Because we multiply the inputs to the softmax by the parameter τ, a value of 1 would mean that we're sampling from the same distribution as before. However, now we are actively sampling instead of always picking the one with the highest probability. A value of 0 smooths all the values in the probability distribution to the same value, which will result in fully random sampling. Values higher than 1 will have the opposite effect and increasingly force our model to sample the character with the highest probability. This temperature model has been implemented in the same file as before.

With a temperature of 0.5, after a 1000 steps the model generates the following sentence:

$$\text{I, Lriy.n\textbackslash nTr priguutetu smesUq} \tag{35}$$

Pointing to the fact that a τ value of 0.5 approaches random sampling a lot. The sentence generated at these early points are very incoherent.

Subsequently, even after 10,000 steps, the model will generate highly diverse sentences which can however hardly be construed as valid sentences:

$$\text{1ch mox, oril mecome,]\textbackslash nAs suth} \tag{36}$$

To top it off, even after 400,000 steps the model will output rather incoherent sentences, such as:

$$\text{, tra-soude as every clam; but} \tag{37}$$

Clearly demonstrating the effect of a τ value that approaches random sampling.

With a temperature of 1 the following sentence was generated after a 1000 steps:

$$\text{rn be.\textbackslash nI. The To, alreven,\textbackslash n man} \tag{38}$$

which is due to the increase in odds of random sampling as opposed to greedy sampling and because the model hasn't had the time yet to learn a proper output distribution to sample from.

Taking the output generated at timestep 10,000 we get the following:

$$\text{of the physure of this acces} \tag{39}$$

Again, the words are pretty close to English words with slight mistakes here and there. Generally, it appears to be a bit more creative and diverse than the sentences generated at this point by the greedy sampling method, but this comes at the cost of making more mistakes.

After 400,000 steps, the model will have converged for a while, and picking a sentence generated at this point should reflect the ultimate performance of the model:

$$\text{quality of justice, and \textbackslash nthe em} \tag{40}$$

Again, it becomes apparent that a temperature of 1 allows for the formation of syntactically and semantically more sentences that are also more coherent, yet still allows for some odd mistakes in writing that occurred far less frequently in the greedy sampling case.

With a temperature of 2, after a 1000 steps the model generates the following sentence:

$$\text{/s in the fors ats of the pros} \tag{41}$$

In which it makes more word errors than was the case for the greedy sampling approach. When it comes to the sentence generated after 10,000 steps, we see a same pattern:

$$\text{: See "The State of the condit} \tag{42}$$

However, it also becomes apparent that the sentences generated tend to quickly become more diverse than the ones found in greedy sampling, which consisted of a lot of repetition. Subsequently, the model generates the following sentence after 400,000 steps:

$$\text{Americans are to be said to ob} \tag{43}$$

Which starts to resemble a very qualitative English sentence (if we complete 'ob' as 'obey <something>'. All in all, the model with a temperature of 2 seems to make slightly more syntactical errors than the greedy sampling approach but it also generates a greater diversity of novel sentences which may convey more interesting semantic interpretations.

3 Graph Neural Networks

3.1 GCN Forward Layer

Question 3.1

a) This layer exploits the structural data in the graph data by using its adjacency matrix. The result of using the adjacency matrix is that during an update with matrix multiplication, each node aggregates information only from its neighbours! Basically, the graph convolution layer makes sure that each node becomes a (weighted) sum of the values of its neighbours. In this way, the structural information of the graph is exploited because a specific node will receive most information from its closest neighbours. By repeatedly doing so, each node will ultimately pass its message over the graph to nodes that are ultimately connected to it through a network of edges.

   By looking at the provided graph in Figure 2 in the assignment, node A does not have a vertex going to node C. However, during the first update, node B will become an aggregation of its neighbourhood nodes A, C and D and node C becomes an aggregation of its neighbouring nodes D and B. Repeating this a second time, the node representation of C will again be another aggregation of the neighbours nodes D and B. However, this time around, the node representation of B includes a component that was sent to it by node A. In this way, C now also incorporates the message of node A that was sent earlier and in this way, a GCN layer can be seen as performing message passing over the graph.

b) An important drawback to GCNs is their memory requirement. Graphs can get very big which may make it very difficult to store them in memory. The use of mini-batches may counteract this, but because data on graphs is related to each other through their edges, the generation of mini-batches is dependent on the different layers so generating mini-batches is not as straightforward as would be the case for say CNNs. Using mini-batches may require approximations for these layers instead of calculating the real deal for dense, large graphs.

Question 3.2

a) See table

	A	B	C	D	E	F
A	1	1	0	0	1	1
B	1	1	1	1	0	0
C	0	1	1	1	0	0
D	0	1	1	1	0	1
E	1	0	0	0	1	1
F	1	0	0	1	1	1

b) It will take 3 updates to forward the information from node C to node E

3.2 Applications of GNNs

Question 3.3
A large variety of possible real-world applications for GNNs exist. A broad category that GNNs can be applied to is text applications. E.g. GNNs extend pretty well to POS tagging, sentiment classification, relational reasoning, and neural machine translation [Zhou and Sun, 2018] because they all contain an element of relation between elements in their internal structure. For example, specific types of word relate to each other in a sentence, a structure which can be used by GNNs to improve translation.

Moreover, GNNs can be used very well on image data. For example, social relationship understanding [Wang and Lin, 2018] can be better understood by propagating a node message through the graph to uncover specific interactions between the persons in an image. Moreover, this extends so tasks such as object detection and semantic segmentation, where we can use the relational structure to better predict whether we are dealing with which object in an image.

Perhaps one of the strongest use cases of GNNs lies in science, especially physics and chemistry. Molecules and proteins (e.g. for folding of proteins) can nearly be seen as a 1:1 transcription of a graph, and this is a good example of where GNNs can be very useful e.g. in protein structure classification problems [Zamora-Resendiz and Crivelli, 2019].

3.3 Comparing and Combining GNNs and RNNs

Question 3.4

a) RNNs and GNNs share a lot of similarities, in the sense that they both work on structured data. In case we model the same type of data as a sequential representation to be fed to the RNN and as a graph repressentation for the GNN, they would possibly still outperform each other on different tasks.

   For example, let's say we have a jazz music score and we have to improvise over this. This means that our model will have to predict a new note to be played based on the previous notes that were played. Because music is highly sequential in nature, an RNN, especially one with an architecture similar to that of an LSTM, will probably outperform the GNN on this task, because the new note should rather be a follow-up to a sequence than an aggregation of its (predecessing) neighbours which the GNN would probably model. This would more likely result in notes that are off-tune (given that we're playing more smooth jazz rather than highly free jazz).

   On the other hand, GNNs would probably outperform RNNs when it comes to image classification [Di Massa and Gori, 2006]. An image could be represented both as sequential data and as a graph in which neighbouring pixels are connected by edges. However, since images are not really sequential in nature, trying to model it using the sequential nature of RNNs would probably not result in very good classifications. This happens because exploiting the temporal component of RNNs does not really capture the spatial relations required to classify an image. GNNs however, would work on a graph representation of an image, which potentially is not all that bad. It leaves the notion of spatial relations between components of the image intact, allowing for the GNN to model specific relations more accurately, in turn allowing it to capture the objects present in the image to improve classifications [Di Massa and Gori, 2006].

   All in all, sequence representations are better suited to tackling tasks that consist of a temporal component, such as sentence generation and playing improv over musical scores and are thus more expressive for data that is sequential in nature. Graphical representations on the other hand, are more expressive for data that has a strong spatial component that can be expressed properly in a graph. Examples of this are actual graphs of friends on social media, images, and geographical data.

b) A very interesting application could be in GCRNNs [Ruiz and Ribeiro, 2019] to identify the epicenter of an earthquake from seismic waves. For example, seismic waves consists of a graph component in their relation to each other and can thus be represented as such, whereas their movements can be tracked through time with an RNN. The combination of these two architectures should bring the best of both worlds to accurately identifying the epicentre of an earthquake given the data on seismic waves.

References

Cui G. Zhang Z. Yang C. Liu Z. Zhou, J. and M. Sun. Graph neural networks: A review of methods and applications. *arXiv*, 2018.

Chen T. Ren J. Yu W. Cheng H. Wang, Z. and L. Lin. Deep reasoning with knowledge graph for social relationship understanding. *arXiv preprint arXiv:1807.00504*, 2018.

R. Zamora-Resendiz and S. Crivelli. Structural learning of proteins using graph convolutional neural networks. *bioRxiv*, page 610444, 2019.

Monfardini G. Sarti L. Scarselli F. Maggini M. Di Massa, V. and M. Gori. A comparison between recursive neural networks and graph neural networks. *IEEE*, pages 778–785, 2006.

Gama F. Ruiz, L. and A. Ribeiro. Gated graph convolutional recurrent neural networks. *arXiv*, 2019.