Hannah Lim
10588973
hannah_lim@outlook.com

Assignment 2
Recurrent Neural Networks and Graph Neural Networks
Deep Learning, 2019

November 29, 2019

1 Vanilla RNN versus LSTM

1.1 Toy problem: Palindrome Numbers

1.2 Vanilla RNN in Pytorch

1.1)

$$\frac{\partial \mathcal{L}^{(T)}}{\partial W_{ph}} = \frac{\partial \mathcal{L}^{(T)}}{\hat{y}^{(T)}} \frac{\hat{y}_k^{(T)}}{\partial p^{(T)}} \frac{\partial p^{(T)}}{\partial W_{ph}}$$
$$= -\frac{y^{(T)}}{\hat{y}^{(T)}}(diag(\hat{y}^{(T)}) - \hat{y}^{(T)}\hat{y}^{(T)^T})h^{(T)}$$
$$\frac{\partial \mathcal{L}^{(T)}}{\partial W_{hh}} = \frac{\partial \mathcal{L}^{(T)}}{\hat{y}^{(T)}} \frac{\hat{y}_k^{(T)}}{\partial p^{(T)}} \frac{\partial p^{(T)}}{\partial h^{(T)}} \frac{\partial h^{(T)}}{\partial W_{hh}}$$
$$= -\frac{y^{(T)}}{\hat{y}^{(T)}}(diag(\hat{y}^{(T)}) - \hat{y}^{(T)}\hat{y}^{(T)^T})W_{ph}(1 - h^{(T)^2}(W_{hx}x^{(T)} + W_{hh}h^{(T-1)} + b_h))h^{(T-1)}$$

In the computed gradients we can see that the gradient of the loss with respect to W_ph is only dependent on the current hidden state, while the gradient of the loss with respect to W_hh is dependent on the current hidden state and that of the previous hidden state. The reason the gradient of the weight for the output is only dependent on the current hidden state is because the linear output is only dependent on the final state.
The latter gradient is recursively dependent on all the previously hidden states which can result in an exploding or vanishing gradient.

1.2)

See code.

1.3)

The recurrent network implemented in question 1.2 is trained on different palindrome length as can be seen in figure 1. The accuracy is measured after the model has trained 10000 steps or an accuracy of 100% is reached over the last 100 steps. If the average accuracy of the last 100 steps is 100 %, I assume the model has converged and we can train for the next palindrome length.
The model is trained with the default parameters:

- hidden units: 128
- batch size: 128
- learning rate: 0.001
- maximum train steps: 10000

Until a palindrome length of 8 the RNN predicts the last character well with an accuracy of 100 %. Hereafter the accuracy drops down and has a peak of 60 % at sequence length 12, which is still a pretty good prediction. From length 13 the accuracy is around the 10 %, which means the model is unable to predict the last character and randomly guesses (since there are 10 characters to choose from). It is remarkable that the model has a peak at a palindrome length of 17 and 27 of 20 %. A possible explanation could be that the model predicts well for a palindrome length of 7, so that the model would also predict better on a palindrome length of 37 % than for instance on a palindrome length of 36.

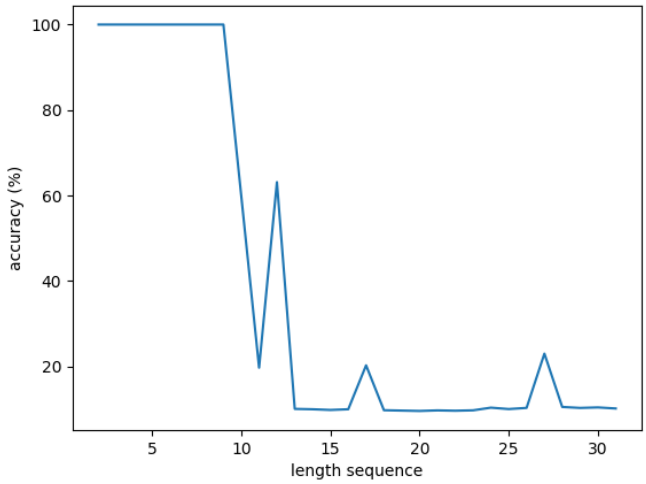

Figure 1: The accuracy (%) of predicting the last character over different palindrome length with a vanilla RNN

1.4)

Stochastic Gradient Descent (SGD) in all its possible forms is one of the most used optimization algorithms for deep learning [1]. Instead of computing the whole gradient, as done with full batch gradient descent, SGD approximates the gradient by averaging over a smaller batch (or even one training example). Therefore SGD converges much faster.

In the case where a dimension has a very high gradient, but it is not in the dimension of the local minimum, the SGD will oscillate a lot and only slowly will move towards the minimum. These oscillations slows down the process and even can prevent to find the minimum. This is what **momentum** can solve. Momentum dampens the oscillations by using the previous gradients when updating.

Therefore, SGD with momentum improves the speed of learning by converging faster to the minimum. When updating the current weight, momentum takes the gradient of the previous weight and of the current weight into account. This new term (with a momentum parameter) causes the update to dampen the

oscillations because the momentum term will average out the oscillations and will put more weight on the gradients in the dimension of the minimum [2]. Momentum causes the model to become more stable and accelerate the learning, by increasing the updates of the parameters as long as the gradient is in the same direction (as the previous gradient) and will decrease the update in the direction where the gradients are perpendicular.

Another way to dampen the oscillations of SGD is by using an adaptive learning rate; updating the learning rate over time. Root Mean Squared Prop, **RMSProp**, is one of these algorithms that dampens the oscillations and causes faster convergence. RMSProp has a different learning rate for every weight and updates these at every step by speeding up the learning rate in the direction of the minimum and slowing down the learning rate in the other directions [1]. It does this by dividing the learning rate by the RMSProp term, which is dependent on the gradient of the current and the previous weight. This updated learning rate is therefore different for every weight. The dimension with a high current and previous gradient will therefore have a very small learning rate and the direction with a smaller current and previous gradient will have a bigger learning rate. This algorithm also solves the problem of choosing a good learning rate before training that is not too small or too big.

In addition, the **Adam** algorithm simply combines the momentum algorithm with the RMSProp algorithm. Therefore Adam uses two hyperparemters; the hyperparameter for momentum and the hyperparameter for RMSProp. Adam also uses bias correction for both the momentum and the RMSProp term. So Adam does a weight update by subtracting the learning rate times the momentum term divided by the RMSProp term [3].

1.3 Long-Short Term Network (LSTM) in PyTorch

1.5a)

All four gates take as input the previous output and the current input.

1. input modulation gate $g^{(t)}$. This gate creates a vector of possible candidates to store in the cell. The non-linearity tanh function is used here. A tanh function is used because it fixes the vanishing and exploding gradient problem.

2. input gate $i^{(t)}$. This gate decides which information we will store in the cell. This gate uses a non-linear sigmoid function. an output of 0 means not important and 1 means important and should be added to the cell state. switch that determines if the new input is relevant to add for new memory.

3. forget gate $f^{(t)}$. This gate makes the decision if the information from the previous cell state should be kept or forgotten. It does this with a non-linearity sigmoid function, where an output of 1 results in keeping the information and an output of 0 results in forgetting the information. The closer the value is to 0, means it must be forgotten and the closer the value is to 1, means the information must be kept.

4. output gate $o^{(t)}$. This gate decides which parts of the cell state will be the output. This is also done with a non-linearity sigmoid function. With a value closer to 0, meaning less relevant for the output and a value closer to 1, meaning relevant for the output.

1.5b)

$4(dn + n^2 + n)$

1.6)

The LSTM is trained in the same way the RNN is trained as can be seen in figure 2). The accuracy is measured after the model has trained 10000 steps or an accuracy of 100% is reached over the last 100 steps. The model is also trained with the default parameters:

- hidden units: 128
- batch size: 128
- learning rate: 0.001
- maximum train steps: 10000

Until at least a palindrome length of 30, the LSTM performs with an accuracy of 100%, which is extremely high. When doing training with the LSTM on a sequence length of 100, the LSTM still converges to an accuracy of 100 %. A possible explanation for this could be that the LSTM learns to only remember the first character, which is always the correct character for the end of the palindrome.
In figure 3 the RNN model and the LSTM model are compared. We can see that the LSTM performs much better than the RNN. This shows that the LSTM is more capable of long-term memory than the RNN.

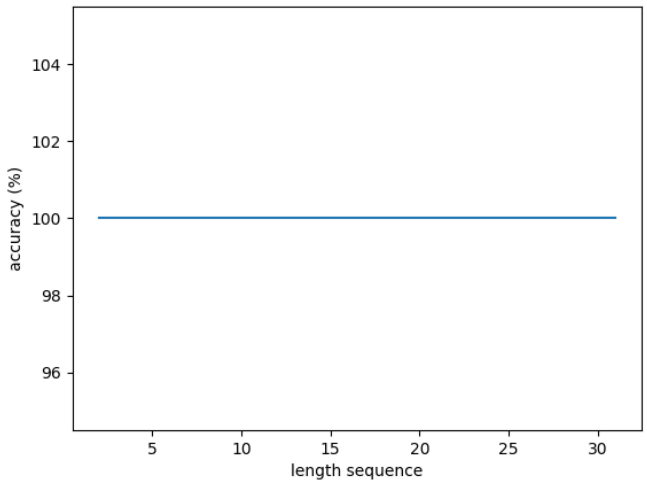

Figure 2: The accuracy (%) of predicting the last character over different palindrome length with a LSTM

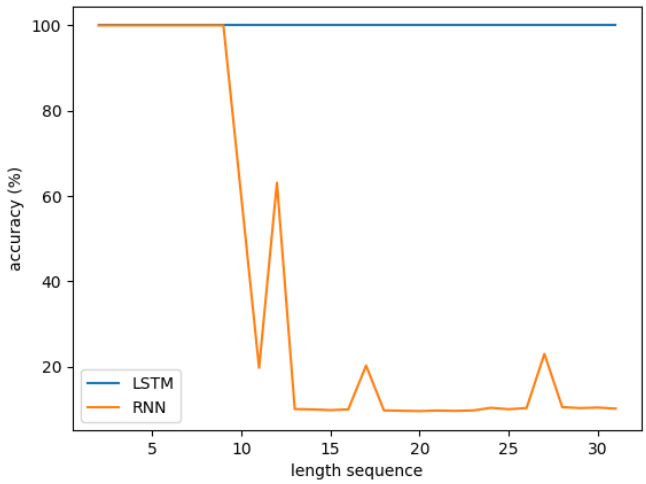

Figure 3: The accuracy (%) of predicting the last character over different palindrome length with a vanilla RNN and a LSTM

1.7)

To get more insights in the difference of RNN and LSTM we can look at the gradient of the loss with respect to the hidden state at every timestep as can be seen in figure 4. The gradients are computed at every hidden state for a model which gets an input of palindrome length of 10. These gradients are averaged from 1000 times iteration of this process to make the gradient more robust. As expected the latest steps have a higher gradient an thus will influence the outcome more than the previous gradient. This is for both models the case. I would have expected that the RNN would have a steeper slope, since it would be more dependent of the last character whereas the LSTM would be able to have a bigger gradient for more states.
If we look closely we do see that the gradient of the first state of the LSTM is not 0, which does confirm that the LSTM is able to maintain the first state in the memory.

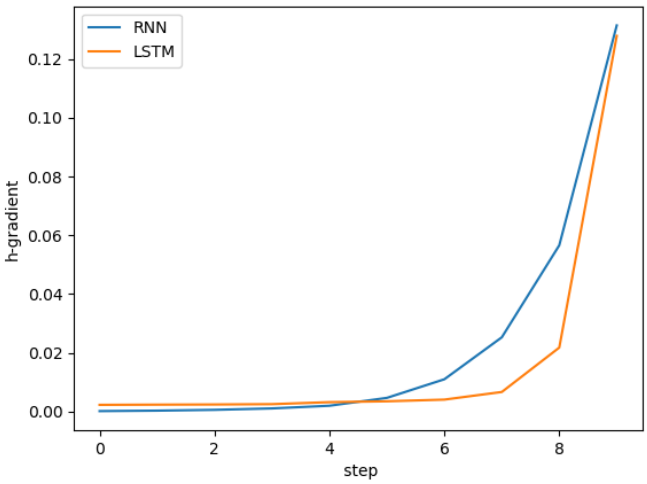

Figure 4: hidden state gradient over time for the RNN and LSTM model (for a palindrome length of 10) without training

2 Recurrent Nets and Generative Model

2.1a)

In figure 5 the accuracy and the loss are shown on the 'book_EN_grimms_fairy_tails.txt' text with the following hyperparameters:

- number of hidden layers: 2
- number of hidden units: 128
- optimizer: RMSProp
- learning rate: 2e-3
- batch size: 64
- sequence length: 30
- maximum steps: 8435 (due to the length of the text)

The final accuracy is 63.7% with a loss of 1.1. After already 4000 steps the model predicts pretty well with an accuracy of approximately 60%. In further training the accuracy slowly keeps increasing and the loss keeps decreasing.
When increasing the number of hidden units to 256 units, the two layer LSTM reaches an accuracy of 69.01 % and a loss of 0.98. Even more increasing the number of hidden units to 512 results in an accuracy of 75% with a loss of 0.75 as can be seen in figure 6.
Increasing the amount of layers from 2 to 3 gives an accuracy of 64.64 % and a loss of 1.15.

Training on 'book_NL_darwin_reis_om_de_wereld.txt' gives an accuracy of 60% and a loss of 1.29 after 19670 train steps and training on the 'book_EN_democracy_in_the_US.txt' gives an accuracy of 67.0 % and a loss of 1.08 after 11100 train steps.

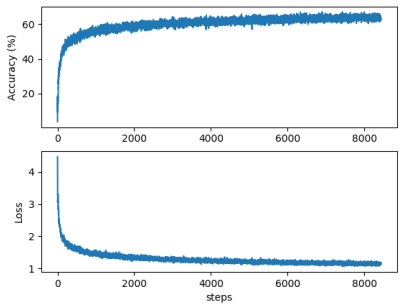

Figure 5: top graph: accuracy (%), bottom graph: loss of predicting the next character on the grimms fairy tale text with default settings

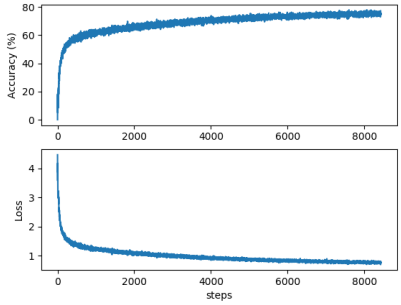

Figure 6: top graph: accuracy (%), bottom graph: loss of predicting the next character on the grimms fairy tale text with all hyperparameters on default settings except the hidden units adjusted to 512 units

2.1b)

By randomly setting the first character of a sentence for generating a sentence with length 30 a few example sentences after a certain amount of training steps are shown in table 1. Because it also selects symbols instead of letters as first character I choose to set the first character by randomly selecting a lower- or uppercase letter from the alphabet. These results can be seen in the tables 2 and 3. Also I experimented with initializing with a self chosen letter so the difference over time could be noticed more accurately. This is seen in the next question (2.1c).

As we can see from the different tables the sentences make more sense when more training steps have been taken. If only 1000 training steps are down the model is biased towards the words the and despite the input character will likely predict the word the to appear soon in the generated sentence. Generating sentences with only a length of 10 makes it more difficult for us to discuss the quality of the generator. Creating sentences from a longer length increases the ability to analyse the model. After 8400 steps with an initialization of a random lowercase

letter we can see in table 3 a grammatically correct sentence generated. This is harder to conclude from the sentence with sequence length 30.

For a better text generator, training on sentences with longer sequence lengths will cause the model to generate better constructed sentences, but this is obviously computationally more expensive.

steps	**sequences (T=30) begin random**
100	Gut the wat the wat the wat the
1000	Qo the wild not see the will se
2000	* THE THE BROTHE TO THE FOU
5000	the bear was to be a child was
8400	g the work and the servant said

Table 1: Examples of generated sentences after training steps by randomly selecting a character from the vocabulary

steps	**sequences (T=60)**	**sequences (T=30)**	**sequences (T=10)**
100	F the was the was the was the was the was the was the	I the the the the the the t	But the the
1000	D The second the first the first the first the first the firs	And the streaked the streaked t	Ver had not
2000	Gretel said, 'I will see the world said, 'I will see the worl	Cand the stand to himself and t	QENSEN AND
5000	VER THE SAORY There was a great great coming the door to the	USE AND THE SEVEN THE TALES Th	QEARD THIS
8400	But the woman said, 'I will not be a boon of the way to me.'	Jorinda who was a great feast,	Hans with t

Table 2: Examples of generated sentences with different sequence lengths after training steps by selecting an uppercase letter

steps	**sequences (T=60)**	**sequences (T=30)**	**sequences (T=10)**
100	g the with the with the with the with the with the with the w	g on the the the the the the the	g s the the t
1000	o the second to the could not the second to the could not the	g the words of the words of the	in the work
2000	for the stars, and the stars, and the stars, and the stars, a	the wolf was to be to the world	ing to the
5000	ll the wood, and the soldier was so much to the forest and sa	ve you to the fire and said, 'I	d the bird
8400	and the second day she came to the horse, and said, 'I am not	quite servant was a pig out of	quite had a

Table 3: Examples of generated sentences with different sequence lengths after training steps by selecting an lowercase letter

2.1c)

In tables 4 and 5 we can see that using a temperature parameter of 0.5 gives the sequence a little more logic and still preserves grammatical structure. An increasing temperature, will decrease the grammatical structure of the sequence and the sequence will be more meaningless. With a temperature of 2, non-existing words are created, new lines appear frequently, capitals and lower characters

appear more randomly after each other and more different symbols appear. With a temperature of 2, it seems like the next character is almost chosen randomly and the sentences make no sense at all.

Greedy sampling uses more frequent words to appear in the sequence, since these have a higher probability to occur and in the mean time remains a kind of grammatical structure. Adding a temperature parameter of 0.5 seems to increase the meaning of a sequence since the grammatical structure is remained but not only the most probable words are chosen.

temperature	**sequences**
-	So the wind was a window and said: 'I have not the second tro
0.5	So the king said, 'We will not really and said, 'Let the tree
1.0	Since and sister with the twelve holes, and do you should tak
2.0	S-xtill mountain. niqueponspell soun, of some ray Ej WHTY [Ac

Table 4: The sequence of characters with sequence length 60 created after full training with different temperatures (or no temperature parameter added). Note: the first character is 'S', so it is better to compare.

temperature	**sequences (T=120)**
-	She was to be done the street to the forest and said: 'I will give you a good for me, I will go to the forest and said: '
0.5	So he was brought him to be away, and began to turn dalent of the way, and said to himself: 'If you can see the table was
1.0	Since and sister with the twelve holes, and do you should tak
2.0	S-xtill mountain. niqueponspell soun, of some ray Ej WHTY [Ac

Table 5: The sequence of characters with sequence length 120 created after full training with different temperatures (or no temperature parameter added). Note: the first character is 'S', so it is better to compare.

3 Graph Neural Networks

3.1 GCN Forward Layer

3.1a)

The adjacency matrix A contains the information if vertices are adjacent or not, the identity matrix I_N is added to the adjacency matrix to add selfloops because otherwise the information from the node itself is lost. This addition results in the adjacency matrix $\tilde{A}$, which contains all the information of the adjacency of the vertices including to the node itself. In the GCN Propagation rule the hidden features in the next layer are defined with a non linear activation function with the input of the previous layer multiplied with the normalized adjacency matrix and the weights. The weights contain the weight of the information transfer for

every edge. This weight can be global for all the edges or can be different for every edge [8], making it more flexible and more representative.
At every time-step the message from a node will be passed further around the graph to the neighboring nodes and itself, so after every time-step every nodes contains more information which will iteratively will be passed on to the neighboring nodes plus itself in the next time-step.

3.1b)

A drawback of GCNs is that at every layer a node only gathers information from its neighbors and itself and not further. Therefore if we for instance want to transfer information from the first node of a long sequence to the last node, we need n amount of layers to get the information at the last node with n being the amount of nodes in the sequence. This is how we overcome this, but it is not very efficient.

3.2a)

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

3.2b)

It will take 3 updates to forward the information from node C to node E, with the following sequence: C-D-F-E or C-B-A-E.

3.2 Applications of GNNs

3.3)

Examples of real-world applications in which GNNS could be applied:

- molecule detection: GNNS can be used to recognize molecules by its structure. With this recognition these molecules can be labeled as a drug for instance [5]. In addition GNNs could be used to learn molecular fingerprints [4].
- text classification: text maybe is not direct structured data, but it can be structured using dependency trees of sentences. From this structured data of the text the GNN can learn certain NLP tasks like for instance classification of bad or good movie reviews.
- image recognition: GNNs can also be used to recognize semantic relationships between objects in images in a similar way as with text . This can be very helpful in the field of Computer Vision [6].

3.3 Comparing and Combining GNNs and RNNs

3.4a)

The main difference between RNNs and GNNs is that GNNs can take as input any kind of graph, while RNNs can only take in directed and acyclic graphs [7]. Therefore RNNs are particularly useful for training on sequential data and not so much on graph structured data. Therefore RNNs could be useful for tasks like the first question of this assignment. Predicting the last character of a palindrome. This is a sequence assignment where the GNNs would be less useful. Since, as mentioned in question 3.1b, GNNs can be used to transfer information of sequential data but therefore needs to have the same amount of layers in the model as the amount of nodes to be able to transfer the information. This is not very sufficient, so therefore RNNs could be a better fit than GNNs.
GNNs are particularly useful to model and comprehend graph structured data, whereas RNN would not be able to capture such a structure. For the data to fit the RNN model, DFS or BFS needs to be done. Where the GNN model computes the next information in every node simultaneously in every layer, the RNN needs to perform multiple steps whereas this is computationally less efficient. GNNs take into account all the neighbors at every step which is very useful for graph structured data.

3.4b)

GNN makes use of a static graph as input. It would be interesting when combining the GNN with a RNN model, so the graphs could change over time. The RNN model, for instance an LSTM, could assign a node or edge to be forgotten in the next graph.

References

[1] Ian Goodfellow and Yoshua Bengio and Aaron Courville. *Deep Learning*. MIT Press, `http://www.deeplearningbook.org`, 2016.

[2] Ning Qian *On the Momentum Term in Gradient Descent Learning Algorithms*. Center for Neurobiology and Behavior, Columbia University

[3] Andrew Ng *Adam Optimization Algorithm (C2W2L08)*. `https://www.youtube.com/watch?v=JXQT_vxqwIs&t=3s`, 25 aug. 2017

[4] Jie Zhou, Ganqu Cui, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, Maosong Sun *Graph Neural Networks: A Review of Methods and Applications*. Cornell University, 20 Dec 2018

[5] Alexander Gaunt *Graph neural networks: Variations and applications*. `https://www.youtube.com/watch?v=cWIeTMklzNg`, Microsoft, 20-04-2018.

[6] Zonghan Wu, Shirui Pan, Member, IEEE, Fengwen Chen, Guodong Long, Chengqi Zhang, Senior Member, IEEE, Philip S. Yu, Fellow, IEEE *A Comprehensive Survey on Graph Neural Networks* JOURNAL OF LATEX CLASS FILES, VOL. XX, NO. XX, AUGUST 2019 1

[7] Vincenzo Di Massa, Gabriele Monfardini, Lorenzo Sarti, Franco Scarselli, Marco Maggini, Marco Gori *A Comparison between Recursive Neural Networks and Graph Neural Networks* Proceedings of the International Joint Conference on Neural Networks, IJCNN 2006, part of the IEEE World Congress on Computational Intelligence, WCCI 2006, Vancouver, BC, Canada, 16-21 July 2006

[8] Lingkai Shen and reviewed by Qiyang L *Deep Learning on Graphs For Computer Vision — CNN, RNN, and GNN* `https://medium.com/@utorontomist/deep-learning-on-graphs-for-computer-vision-cnn-rnn-and-gnn-c114d6004678` UTMIST,Oct 28, 2018