Deep Learning Assignment 1: MLPs, CNNs and Backpropagation

Paul ten Kaate

10743367

paultenkaate@outlook.com

November 29, 2019

1 Vanilla RNN in Pytorch

Question 1.1

The gradient $\frac{\partial \mathcal{L}^{(t)}}{\partial W_{ph}}$ is computed by using the chain rule:

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

The softmax we already got at the first Deep Learning assignment

$$\frac{\partial \hat{y}^{(t)}}{\partial p^{(t)}} = diag(\hat{y}^{(t)}) - \hat{y}^{(t)} \otimes \hat{y}^{(t)} \tag{2}$$

The gradient of $\frac{\partial p^{(t)}}{\partial W_{ph}}$ we also calculated in the first assignment, namely:

$$\frac{\partial \tilde{p}_i^{(t)}}{\partial W_{jk}^{(t)}} = \begin{cases} h_k^{(t)} & \text{if j = i} \\ 0 & \text{if j} \neq \text{i} \end{cases} \tag{3}$$

Thus, the **complete gradient** is:

$$\frac{\partial \mathcal{L}^{(\sqcup)}}{\partial W_{ph}} = \frac{\partial \mathcal{L}^{(t)}}{\partial \hat{y}^{(t)}} \left(diag(\hat{y}^{(t)}) - \hat{y}^{(t)} \otimes \hat{y}^{(t)} \right) \otimes h^{(t)} \tag{4}$$

The gradient of $\frac{\partial \mathcal{L}^{(t)}}{\partial W_{hh}}$ is composed of the following chain rule:

$$\frac{\partial \mathcal{L}^{(t)}}{\partial W_{ph}} = \frac{\partial \mathcal{L}^{(t)}}{\partial \hat{y}^{(t)}} \frac{\partial \hat{y}^{(t)}}{\partial p^{(t)}} \frac{\partial p^{(t)}}{\partial h^{(t)}} \frac{\partial h^{(t)}}{\partial \hat{h}^{(t)}} \frac{\partial \hat{h}^{(t)}}{\partial W_{hh}^{(t)}} \tag{5}$$

The derivative of $\frac{\partial p^{(t)}}{\partial h^{(t)}}$ is again computed in assignment 1.

$$\frac{\partial \mathcal{L}}{\partial h^{(t)}} = \frac{\partial \mathcal{L}}{\partial p^{(t)}} \frac{\partial p^{(t)}}{\partial h^{(t)}} = \frac{\partial \mathcal{L}}{\partial p^{(t)}} W^{(t)} \tag{6}$$

The derivative of $\frac{\partial h^{(t)}}{\partial \hat{h}^{(t)}}$ is as follows:

$$\frac{\partial h^{(t)}}{\partial \hat{h}^{(t)}} = 1 - \tanh^2 \hat{h}^{(t)} \tag{7}$$

The gradient of $\frac{\partial \hat{h}^{(t)}}{\partial W_{hh}^{(t)}}$ is again

$$\frac{\partial \hat{h}_i^{(t)}}{\partial W_{jk}^{(t)}} = \begin{cases} h_k^{(t-1)} & \text{if j = i} \\ 0 & \text{if j} \neq \text{i} \end{cases} \tag{8}$$

Thus the **complete gradient** is:

$$\frac{\partial \mathcal{L}^{(t)}}{\partial W_{hh}^{(t)}} = \frac{\partial \mathcal{L}^{(t)}}{\partial \hat{y}^{(t)}} \left(diag(\hat{y}^{(t)}) - \hat{y}^{(t)} \otimes \hat{y}^{(t)} \right) W_{ph}^{(t)} \left(1 - \tanh^2 \hat{h}^{(t)}\right) \otimes h^{t-1} \tag{9}$$

Question 1.2

First, the *__init__()* and *forward()* function are created. The weights are initialized by using *nn.init.xavier_uniform()*. The biases are set to zero. The *forward()* function is implemented with matrix multiplications in a for-loop.

In the *train.py* file, the code to run the neural network is given. *torch.nn.CrossEntropyLoss()* is used as loss function, and *torch.optim.RMSprop()* is used as optimizer. Each forward pass is executed with the custom *forward()* function, for the backpropagation, the standard *backward()* and *step()* function are used. *Torch.nn.utils.clip_grad_norm_* is used so that the gradients do not explode

Question 1.3

The network is trained 4 times for each palindrome length between a range of 5 and 35. The maximum number of iterations is set to 1000, as testing showed that if the network still produces an accuracy of around 0.1 at 1000 iterations, it will

not increase anymore. The network stops either when the test accuracy reaches 1, or the maximum number of steps is reached. Figure 1 shows the individual results per training, and the mean and standard deviation per palindrome length. As can be seen, until a palindrome length of 15, the network does always produce a score of 1. Hereafter, the relative amount of times the network reaches an accuracy of 1 decreases.

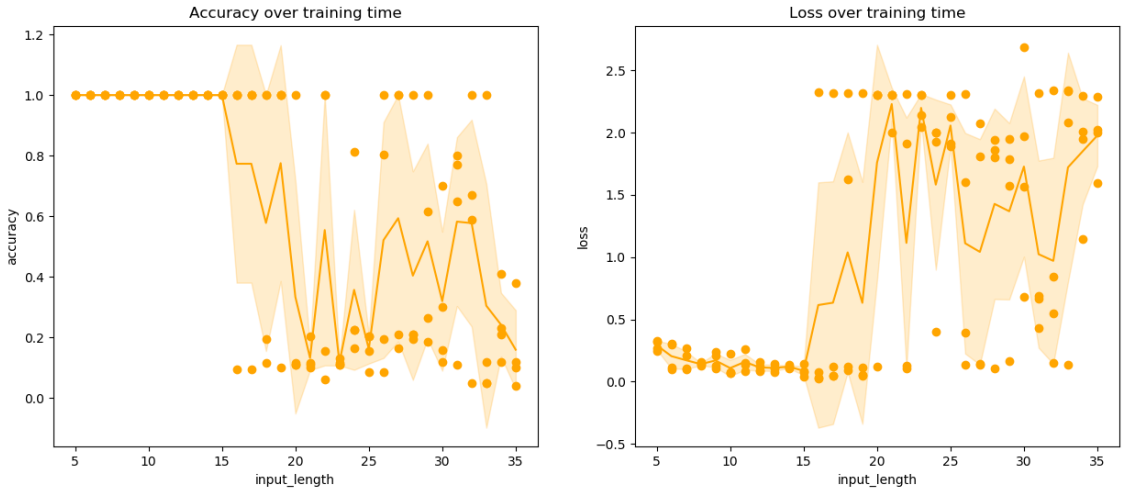

Figure 1: Loss and accuracy with the Vanilla RNN

Question 1.4

The advantage of the RMSprop optimizer is that it uses an adaptive learning rate to update the weights. SGD uses a constant learning rate. With a fairly constant loss surface, SGD can work fine. However, with a noisy loss surface or a surface with a plateau, the gradient can become either to large or too small to (quickly) converge to an optimum. With adaptive learning rate, the RMSprop tackles this problem, as the learning rate increases when the gradients become very small, and decreases when the weight become large, to respectively leave plateaus or e.g. get to the bottom of a loss valley.

Adam takes this approach, but adds momentum. The Adam optimizer can be seen as a heavy ball rolling down a hill. It does not only take into account the last gradient, but also the past gradients. The Adam optimizer therefore converges quicker to a minimum when the past gradients are similar, and becomes less susceptible to noise in the loss surface.

Question 1.5

a) The LTSM has multiple gates. Each of them has a different purpose.

   (i) The *input modulation gate* $\boldsymbol{g}^{(t)}$ brings all inputs to a value between -1 and 1. This is good to show what effect the inputs should have (positive of negative).

   (ii) The *input gate* $\boldsymbol{i}^{(t)}$ decides which values will be updated. the sigmoid is a suitable non-linearity, 0 means the value is not important and thus not taken into acount, the node contribution vanishes if multiplied by 0.

   (iii) The *forget gate* $\boldsymbol{f}^{(t)}$ concatenates info of the previous state and the current state. This gate decides which info can be kept, and which info must be forgetten. The sigmoid non-linearity is suitable for this, as it maps the output between 0 and 1, where nothing happens if the output is 1, and the previous info is forgotten if the previous cell state is multiplied by 0.

   (iv) The *output gate* $\boldsymbol{o}^{(t)}$ decides what the hidden state should be for the next cell. THe sigmoid is again used as a good non-linearity function to show the importance of each input.

b) For each of the four gates, a weight matrix mapping x to h is trained with $d \times n$ parameters. Also a weigth matrix mapping h to h is updated with $n \times n$ parameters. furthermore every gate has a bias of size n The total amount of parameters therefore is equeal to $4dn + 4n^2 + 4n$.

Question 1.6

The network is run with the same default parameters as in Question 1.3. The maximum number of steps is set to 4000 for computational considerations, a test showed that a larger maximum step size is not needed for the LSTM to converge. The network is trained 4 times for each palindrome length between a range of 1 and 39.

The *Adam* optimizer proved to converge quicker and find an optimum more regularly than the *RMSprop* optimizer. When initializing the weights with *nn.init.xavier_uniform_*, the network consistently converges to a maximum until a palindrome length of 33. The results of this LSTM can be found in Figure 2.

When using the *nn.init.kaiming_normal_* initialisation, the the network consistently converges to a maximum until a palindrome length of 39. The results are shown in Figure 3. When comparing these results, you can conclude that the LSTM performs better on larger palindromes than the Vanilla RNN.

Question 1.7

Figure 4 shows the gradient magnitude of each h for both the RNN and the LSTM network. What can be seen, is that the gradient of earlier states decrease faster for the LSTM. This is counter-intuitive, as this would mean that the LSTM suffers greater from vanishing gradients. Vanishing gradients have as consequence that the network performs bad on longer sequences, however in Question 1.3 and 1.6, we saw that the LSTM performs better on longer gradients than the Vanilla RNN. Therefore, you would expect the labels to be switched.

After some training, if the loss decreases and converges, you would expect the gradient magnitudes to decrease, as the weights also converge to an optimum.

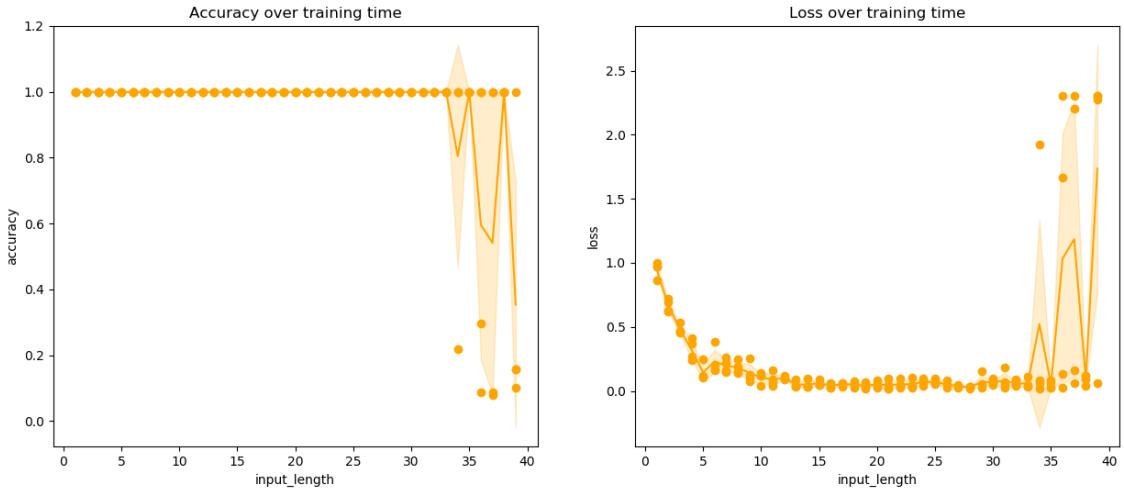

Figure 2: Loss and accuracy with a LSTM and weights initialized with the Xavier initialisation.

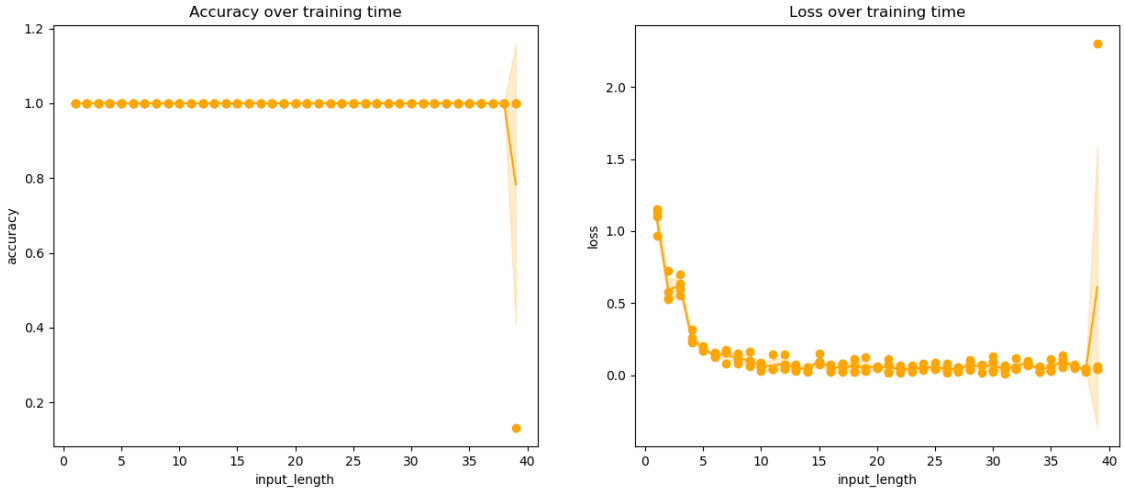

Figure 3: Loss and accuracy with a LSTM and weights initialized with the Kaiming initialisation.

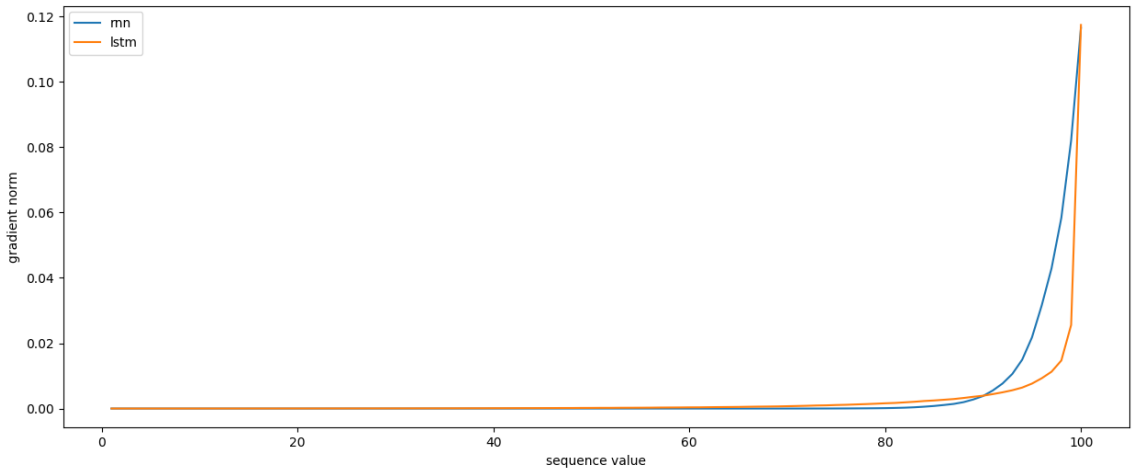

Figure 4: The gradient magnitude of each h after backpropagation.

Recurrent Nets as Generative Model

Question 2.1

a) A two-layer neural network is implemented in this part of the assignment. the LSTM layer is created with *nn.LSTM()*, and the output is generated with a *nn.Linear()* layer. The following parameters are used.

Hyperparameter	**value**
Sequence length	30
Hidden layer size	128
Batch size	64
Learning rate	2e-3
Number of hidden layers	2

The increasing the hidden layer size does not significantly change the accuracy, however it does increase the computational expense. Furthermore, changing the batch size did only decrease the amount of steps until

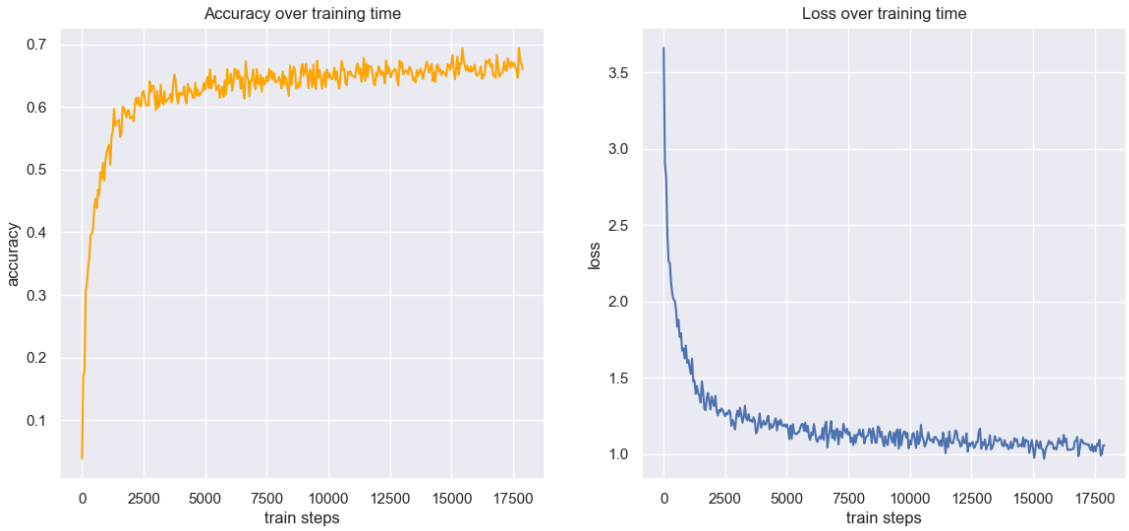

Figure 5: Accuracy and loss of the LSTM algorithm

convergence, however it also increases the computational expense per step.

For this experiment, the file *assets/book_EN_democracy_in_the_US.txt* is used. The dataset is first pre-processed by filtering out everything except letters, numbers and spaces, to create a simpler dataset. Before training and predicting with the dataset, the batches are one-hot encoded.

In Figure 5 the accuracy and loss of the LSTM algorithm is reported. The accuracy converges around 0.65, and the loss converges to 1.1.

b) For the second part of the assignment, during several stages of the network training, random characters are initialized and fed to the network, and sentences with different lengths are generated. Table 1 shows these sentences.The network is still trained on a sequence length of 30.

A few things can be noted from these results. First of all, quite quickly the algorithm produces existing words already. The variation increases when the training progresses. However, what can be seen, is that, when generating longer sentences, the predictions start to show repetition. From 4000 steps, the generated sentences with 30 characters do not show any repetition, but the generated sentences with 60 characters do. This does not change during training. A good explanation for this could be that the model hasn't seen any sentences that are longer than 30 characters. Therefore it is not trained to predict sentences longer than 30 characters either and falls into repetition.

Step	Length	Sentence
1000	20	mere the anter the an
	30	the anter the anter the anter
	60	ver the anter the anter the anter the anter the anter the ant
2000	20	f the constitution of
	30	contreal the constitution of th
	60	le the constitution of the constitution of the constitution o
3000	20	100
	30	be the states of the states of
	60	n the states of the states of the states of the states of the
4000	20	in the constitution o
	30	7 and the american in the cons
	60	present of the constitution of the constitution of the consti
5000	20	ver the states of the
	30	quently are the states of the s
	60	7 the states of the states of the states of the states of th

Table 1: Generated sentences at different stages in the training

c) The temperature is used to soften or exaggerate the output of the LSTM when predicting new letters. Before calculating probabilities by putting the output in the softmax function, the output is multiplied by the temperature. The function is shown here under.

$$\text{softmax}(\tilde{x}) = \frac{\exp(\tau\tilde{x})}{\sum_i \exp\left(\tau\tilde{x}_i\right)}$$

By multiplying the LSTM output a small temperature, the output range becomes smaller, resulting in probabilities that are closer to each other. On the other hand, when multiplying the output with a temperature larger than 1, the output range increases, resulting in a larger probability differences.

Besides, after multiplying the output with the temperature and applying the softmax, instead of taking the letter with the maximum probability, we get the next letter by sampling from the probability distribution, so that the next letter becomes more random. Using sampling instead of getting the maximum probability, the generated sentence becomes more random. With this, you solve problems like getting stuck in a loop, as we saw in the previouw question. A smaller temperature results in a more smoothed probability distribution, so the next chosen letter letter will be more random than with a larger temperature.

Table 2 shows generated sentences with 200 characters after 5000 steps, for different temperatures. As can be concluded, the the randomness indeed increases with a lower temperature. However, at a temperature of 0.5 and 1, the distribution is thus far smoothed that the network creates non-existent words. A temperature of 2 seems to work better, as the network is able to create existing words, but it does not fall into repetition like seen in Question 2.2.

Temperature	Sentence
0.5	3 s on his its drawem him were duppinted orbolys drink am inti uswam cater was eegesbeeveshappugning immensbaniso true 218unisarickssent flems oz fou now thround anrobs equir another kia
1	for my with his dirit i will till they could love in project gutenberg seven round her eling a princesser and beautable now that less down to speel must in purples and he ohen he hung d
2	oor the king was asleep and the morning the court of many me on the sixth and he could not make a preasion and the old man was seen the door the water and he was set and gretel and he too

Table 2: Generated sentences with different temperatures.

Graph Neural Networks

Question 3.1

a) The adjacency matrix A is a matrix that contains the structural information of the graph. At each layer, the previous output is multiplied with the matrix A, and thus the information of the past layer is summed only for the nodes with connections. Therefore, this can be seen as a message being passed through nodes with direct links each layer.

b) As mentioned in Kipf and Welling (2016), one of the drawbacks in a GCN in this setup is that it presumes the self-connection is as important as all other relations. This can be overcome by multiplying the identity matrix in the Eq. 16

1	1	0	0	1	1
1	1	1	1	0	0
0	1	1	1	0	0
0	1	1	1	0	1
1	0	0	0	1	1
1	0	0	1	1	1

Question 3.2

a) The table has a 1 on nodes with relations, and zeros else. The table looks as follows:

b) Looking at Figure 2 in the assignment, the minimum amount of nodes that need to be passed to arrive at E from node C is three. As each layer passses information one node further, three updates are needed to take information from C to E.

Question 3.3

As written in an online article written by Jadhav (2019) one of the applications is in Chemistry, where by using the the ion and atom structure, properties of new molecules are predicted by using known molecules and their properties. Also, new molecules can be found by using GNNs. Another example that is mentioned is using a GNN as an alternative to heuristic methods to solve problems like the traveling salesman problem.

1.1 Question 3.4

a) An RNN is suitable when in the dataset there is a clear directional relationship between the data. E.g. for time-related data, the direction is clear, and the info should go one way when iterating through the data. When there is no direction between the data relations however, a GNN is more suitable for the task.

References

Jadhav, A. (2019). *Applications of graph neural networks.* Retrieved 2019-11-29, from `https://towardsdatascience.com/https-medium-com-aishwaryajadhav-applications-of-graph-neural-networks-1420576be574`

Kipf, T. N., & Welling, M. (2016). Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*.