Assignment 2. Recurrent Neural Networks and Graph Neural Networks

University of Amsterdam – Deep Learning Course

November 18, 2020

The deadline for this assignment is November 29$^{\text{th}}$ at 23:59.

In this assignment you will study and implement recurrent neural networks (RNNs) and have a theoretical introduction to graph neural networks (GNNs). RNNs are best suited for sequential processing of data, such as a sequence of characters, words or video frames. Their applications are mostly in neural machine translation, speech analysis and video understanding. GNNs are specifically applied to graph-structured data, like knowledge graphs, molecules or citation networks.

The assignment consists of three parts. First, you will get familiar with the theory behind vanilla RNNs and another type of RNN called a Long Short-Term Memory (LSTM) network. Then you will implement LSTM variants and test them on a simple toy problem. This will help you understand the fundamentals of recurrent networks. After that, you will use the standard LSTM for learning and generating text. In the final part, you will analyze the forward pass of a graph convolutional neural network, and then discuss tasks and applications which can be solved using GNNs. In addition to the coding assignments, the text contains multiple pen and paper questions which you need to answer. We expect each student to hand in their code and individually and write a report that explains the code and answers the questions. **The page limit for the report is 12 pages.**

The (incomplete) code can be found **here**.

1 Recurrent Neural Networks **(Total: 30 points)**

1.1 Vanilla RNNs

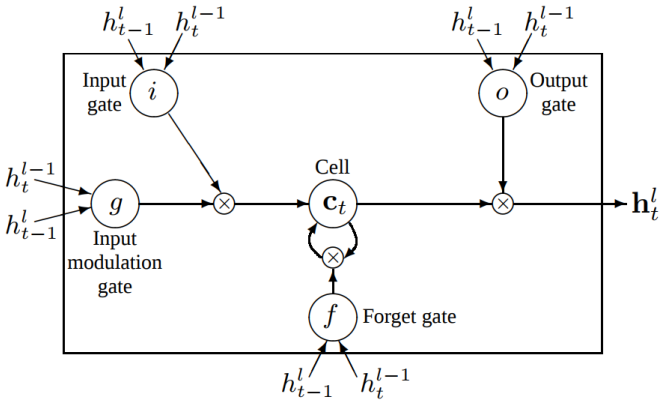
The vanilla RNN is formalized as follows. Given a sequence of input vectors $\mathbf{x}^{(t)}$ for $t = 1, \dots T$, the network computes a sequence of hidden states $\mathbf{h}^{(t)}$ and a sequence of output vectors $\mathbf{p}^{(t)}$ using the following equations for timesteps $t = 1, \dots, T$:

$$\mathbf{h}^{(t)} = \tanh(\mathbf{W}_{hx}\mathbf{x}^{(t)} + \mathbf{W}_{hh}\mathbf{h}^{(t-1)} + \mathbf{b}_h) \tag{1}$$

$$\mathbf{p}^{(t)} = \mathbf{W}_{ph}\mathbf{h}^{(t)} + \mathbf{b}_p \tag{2}$$

As you can see, there are several trainable weight matrices and bias vectors. $\mathbf{W}_{hx}$ denotes the input-to-hidden weight matrix, $\mathbf{W}_{hh}$ is the hidden-to-hidden (or recurrent) weight matrix, $\mathbf{W}_{ph}$ represents the hidden-to-output weight matrix and the $\mathbf{b}_h$ and $\mathbf{b}_p$ vectors denote the biases. For the first timestep $t = 1$, the expression $\mathbf{h}^{(t-1)} = \mathbf{h}^{(0)}$ is replaced with a special vector $\mathbf{h}_{init}$ that is commonly initialized to a vector of zeros. The output value $\mathbf{p}^{(t)}$ depends on the state of the hidden layer $\mathbf{h}^{(t)}$ which in its turn depends on all previous state of the hidden layer. Therefore, a recurrent neural network can be seen as a (deep) feed-forward network with shared weights.

Figure 1. A graphical representation of LSTM memory cells (Zaremba *et al.* (ICLR, 2015))



To optimize the trainable weights, the gradients of the RNN are computed via back-propagation through time (BPTT). The goal is to calculate the gradients of the loss $\mathcal{L}$ with respect to the model parameters $\mathbf{W}_{hx}$, $\mathbf{W}_{hh}$ and $\mathbf{W}_{ph}$ (biases omitted). Similar to training a feed-forward network, the weights and biases are updated using SGD or one of its variants. Different from feed-forward networks, recurrent networks can give output predictions $\hat{\mathbf{y}}^{(t)}$ at every timestep. In this assignment the outputs will be given by the softmax function, *i.e.* $\hat{\mathbf{y}}^{(t)} = \text{softmax}(\mathbf{p}^{(t)})$. For prediction, we compute the standard cross-entropy loss:

$$\mathcal{L} = -\sum_{k=1}^{K} \mathbf{y}_k \log \hat{\mathbf{y}}_k \tag{3}$$

Where k runs over the number of classes. In this expression, $\mathbf{y}$ denotes a one-hot vector of length K containing true labels.

Question 1.1 (5 points)

Recurrent neural networks can be trained using backpropagation through time. Similar to feed-forward networks, the goal is to compute the gradients of the loss w.r.t. $\mathbf{W}_{ph}$, $\mathbf{W}_{hh}$ and $\mathbf{W}_{hx}$. Here $\mathcal{L}^{(T)}$ is the loss on the output predictions at time step $t = T$.

(a) Write down an expression for the gradient $\frac{\partial \mathcal{L}^{(T)}}{\partial \mathbf{W}_{ph}}$ in terms of the variables that appear in Equations (1) and (2). Expand the gradient with sums and products but do not expand recurrent components. (answer up to 2 lines)[a]

(b) Do the same for $\frac{\partial \mathcal{L}^{(T)}}{\partial \mathbf{W}_{hh}}$. (answer up to 3 lines)[a]

(c) What difference do you observe in temporal dependence of the two gradients? Study the latter gradient and explain what problems might occur when training this recurrent network for a large number of timesteps.

[a] Equations can have multiple equal signs and transitions in the same line

1.2 Long Short-Term Memory (LSTM) network

Training a vanilla RNN for remembering its inputs for an increasing number of timesteps is difficult. The problem is that the influence of a given input on the hidden layer (and therefore on the output layer), either decays or blows up exponentially as it unrolls the

network. In practice, the *vanishing gradient problem* is the main shortcoming of vanilla RNNs. As a result, training a vanilla RNNs to consistently learn tasks containing delays of more than ~ 10 timesteps between relevant input and target is difficult. To overcome this problem, many different RNN architectures have been suggested. The most widely used variant is the Long Short-Term Memory networks (LSTMs). An LSTM (Figure 1) introduces a number of gating mechanisms to improve gradient flow for a more stable training procedure. Before continuing, we recommend to read the following blogpost to get familiar with the LSTM architecture: **Understanding LSTM Networks**.

In this assignment we will use the following LSTM definition:

$$\mathbf{g}^{(t)} = \tanh(\mathbf{W}_{gx}\mathbf{x}^{(t)} + \mathbf{W}_{gh}\mathbf{h}^{(t-1)} + \mathbf{b}_g) \tag{4}$$
$$\mathbf{i}^{(t)} = \sigma(\mathbf{W}_{ix}\mathbf{x}^{(t)} + \mathbf{W}_{ih}\mathbf{h}^{(t-1)} + \mathbf{b}_i) \tag{5}$$
$$\mathbf{f}^{(t)} = \sigma(\mathbf{W}_{fx}\mathbf{x}^{(t)} + \mathbf{W}_{fh}\mathbf{h}^{(t-1)} + \mathbf{b}_f) \tag{6}$$
$$\mathbf{o}^{(t)} = \sigma(\mathbf{W}_{ox}\mathbf{x}^{(t)} + \mathbf{W}_{oh}\mathbf{h}^{(t-1)} + \mathbf{b}_o) \tag{7}$$
$$\mathbf{c}^{(t)} = \mathbf{g}^{(t)} \odot \mathbf{i}^{(t)} + \mathbf{c}^{(t-1)} \odot \mathbf{f}^{(t)} \tag{8}$$
$$\mathbf{h}^{(t)} = \tanh(\mathbf{c}^{(t)}) \odot \mathbf{o}^{(t)} \tag{9}$$

$$\mathbf{p}^{(t)} = \mathbf{W}_{ph}\mathbf{h}^{(t)} + \mathbf{b}_p \tag{10}$$
$$\hat{\mathbf{y}}^{(t)} = \text{softmax}(\mathbf{p}^{(t)}). \tag{11}$$

In these equations $\odot$ is element-wise multiplication and $\sigma(\cdot)$ is the sigmoid function. The first six equations are the LSTM's core part whereas the last two equations are just the linear output mapping. Note that the LSTM has more weight matrices than the vanilla RNN. As the forward pass of the LSTM is relatively intricate, writing down the correct gradients for the LSTM would involve a lot of derivatives. Fortunately, LSTMs can easily be implemented in PyTorch and automatic differentiation takes care of the derivatives.

Question 1.2 (5 points)

The LSTM extends the vanilla RNN cell by adding four gating mechanisms. Those gating mechanisms are crucial for successfully training recurrent neural networks.

(a) (3 points) The LSTM has an *input modulation gate* $\mathbf{g}^{(t)}$, *input gate* $\mathbf{i}^{(t)}$, *forget gate* $\mathbf{f}^{(t)}$ and *output gate* $\mathbf{o}^{(t)}$. For each of these gates, write down a brief explanation of their purpose; explicitly discuss the non-linearity they use and motivate why this is a good choice. (up to 3 sentences per gate)

(b) (2 points) Given the LSTM cell as defined by the equations above and an input sample $\mathbf{x} \in \mathbb{R}^{T \times d}$ where T denotes the sequence length and d is the feature dimensionality. Let n denote the number of units in the LSTM and m represents the batch size. Write down the formula for the *total number* of trainable parameters in the *LSTM cell* as defined above, in terms of N_{hidden}, N_{input} and N_{output} for the number of weights associated with the hidden input and output layers respectively. (up to 2 lines)[a]

1.3 LSTMs in PyTorch

For Question 1.3 and 1.4 you are assigned a toy problem (Baum Sweet Numbers, Random Combinations or Binary Palindrome) and an LSTM model variant (GRU, biLSTM or peepLSTM) based on your tutorial group. The group letter here corresponds to the tutorial group letter on datanose. Table 1 below states the group division. Please check

before starting to make sure you work on the right task, LSTM variant and the right hyperparameter settings. The hyperparameters are shown in table 2

Group	Dataset	LSTM variant
A	Baum Sweet Sequence	GRU
B, C	Random Combinations	bidirectional LSTM
D, E	Binary Palindrome	peepLSTM

Table 1. Dataset and LSTM variant to be used for this assignment per group. The groups are corresponding to the tutorial groups on datanose.

For clarity, question 1.3 requires completing the *same LSTM network for all groups* but using a *different dataset per group*. Question 1.4 requires a *different network* per group, using the same dataset you have already used in question 1.3. You are not to skip any questions. Below we provide a detailed description of the three toy problems/datasets.

Baum Sweet Sequence A Baum Sweet Sequence is a sequence of numbers created by converting a natural number to its binary representation and counting the number of zeros in consecutive blocks of zeros. If the binary representation of the number contains no block of consecutive 0s of odd length, the Baum Sweet number is 1, otherwise it is 0. Here are some examples, on the left you see the number, in the center the binary representation of this number and on the right the baum sweet number.

$$3234 \rightarrow 11\underbrace{00}_{\text{even}}1\underbrace{0}_{\text{odd}}1\underbrace{000}_{\text{odd}}1\underbrace{0}_{\text{odd}} \rightarrow 0$$

$$124 \rightarrow 11111\underbrace{00}_{\text{even}} \rightarrow 1$$

Your goal will be to predict the corresponding Baum Sweet number given a binary representation of a natural number. Below, an example training datapoint is shown:

$$\begin{aligned} \textit{Input}: &\ 110010100010 \\ \textit{Label}: &\ 0 \end{aligned}$$

Hence, the input will be a sequence of 0's and 1's and the label is either 1 or 0. The dataloader we provide for this task will return inputs of shape [`batch_size`, `sequence_length`, `feature_dimension`]. The input sequence_length returned by the dataloader depends on the maximum length of the natural number provided to the dataloader (`int_length`) and can be varied. For example, if the int_length is 4, the maximum binary representation length will be the length of the binary representation of 10 exp(4), which is 14. The `feature_dimension` is 1. Any input sequence shorter than the maximum length of the binary presentation is padded to ensure input sequences of fixed length.

Random Combinations

A Random Combination without replacement is a string of natural numbers where each number appears only once. The length of the sequence is equal to the maximum number in the string. In this setting, zeros are not included. Examples are shown below. On the left the input sequence is shown and on the right of the arrow the label is shown.

Group	A	B, C	D, E
Seq_length	[4,5,6]	[5, 10, 15]	[10,20]
Optimizer	Adam	Adam	Adam
Learning rate	0.0001	0.001	0.0001
Update iterations	3000	2000	3000
Batch size	256	128	256
Nr hidden	256	128	256
Max norm[1]	10	10	10
Initialization	Kaiming_normal	Kaiming_normal	Kaiming_normal

Table 2. Hyperparameters and sequence lengths for which to run the experiments per group. To be used for question 1.3 and 1.4.

$$12346789 \rightarrow 5 \tag{12}$$
$$69571248 \rightarrow 3 \tag{13}$$

This is a classification task. In the examples above, the string of 8 digits to the left of the arrow is the input and the digit to the right of the arrow is the label which needs to be predicted.

Binary Palindrome

A palindrome is a sequence which reads the same from left to right as from right to left. In our case we convert a randomly sampled string of natural numbers to their binary representation and concatenate the mirrored binary representation to the original. To formulate it as a prediction problem, we remove the last digit from this binary palindrome and use it as the label. For example:

$$\underbrace{124}_{\text{number}} \rightarrow \underbrace{1111100}_{\text{binary}}, \underbrace{0011111}_{\text{mirrored}} \rightarrow \underbrace{1111100001111}_{\text{input}}, \underbrace{1}_{\text{label}}$$

If you have gone through the code in `datasets.py`, you will see that sometimes the original binary representation is flipped, before mirroring. This is done to increase the frequency of palindromes ending on 0. Otherwise the data would be very imbalanced. The network will have to predict the last binary number of the palindrome. The maximum palindrome sequence length can be calculated from the maximum natural number used to generate the binary palindrome (integer_seq_length, which is an argument for the loader). The input sequence length is calculated as `integer_seq_length` $* 4 + 2 - 1$. This input is padded to a constant size.

Dataloaders

The data generators are provided in the file `datasets.py` and the datasets are implemented as `torch.utils.data.Dataset` objects. Use the dataset corresponding to your group. Note that for the baum sweet and binary palindrome dataset the returned sequences are padded to the maximum length of the binary representation.

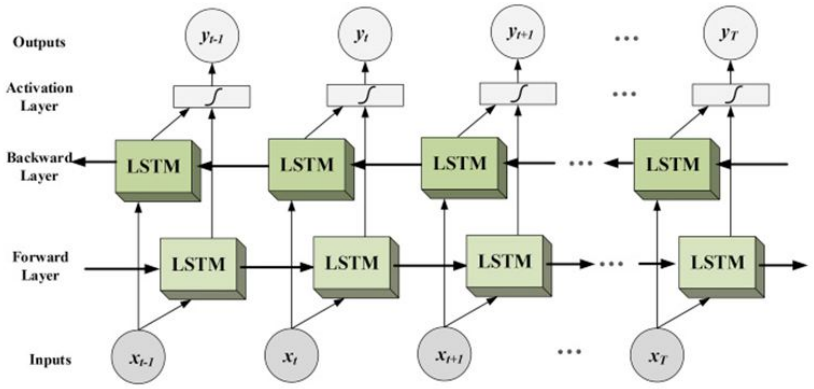


Figure 2. Example illustration for a bidirectional LSTM 2

Question 1.3 (15 points)

First, implement an LSTM network as specified by the equations (4) - (11) above in the file `lstm.py`. The sequence length and batch size are variable, and you need to use embeddings to represent each digit of the input sequence (use `nn.Embedding`). You are required to implement the model without any high-level PyTorch functions, but you are allowed to use `nn.Parameter`. You do not need to implement the *backward* pass yourself, but instead you can rely on automatic differentiation. For the optimization part we have prepared the code in `train.py`.

Using the input of your assigned toy problem, perform experiments with corresponding hyperparameters and sequence lengths that have been specified above. You do not have to design a train or test split and simply evaluate the performance of your model on generated data. Train the network until convergence. Create a plot of your results for the prespecified sequence lenghts. Report your results averaged over 3 random seeds for each experiment, including their standard deviation.
As a sanity check, your LSTM model should obtain:

- perfect accuracy on the random combinations dataset for $T = 6$
- perfect accuracy on the Baum Sweet Sequence dataset for $T = 6$
- perfect accuracy on the binary palindrome dataset for $T = 10$

with your assigned hyperparameter settings where T is the sequence length. (up to a 10-sentences paragraph + Figures)

Next, you are going to implement another LSTM variant in PyTorch and test it on your assigned toy task. For this, we provide a description of all models below.

Bidirectional LSTM

Bidirectional RNNs were introduced first in the work of *Schuster et al. 1997* [1] and later their LSTM variant in the work of *Graves et al. 2005* [2]. Bidirectional LSTMs run an LSTM cell both in the forward pass, starting from the first token, and another LSTM cell in the backward pass, from the last token running back to the first token (not to be confused with the back-propagation pass); as shown in Figure 2.

The forward layer (a standard LSTM cell) is defined as:

$$\overrightarrow{\mathbf{g}}^{(t)} = \tanh(\overrightarrow{\mathbf{W}}_{gx}\mathbf{x}^{(t)} + \overrightarrow{\mathbf{W}}_{gh}\overrightarrow{\mathbf{h}}^{(t-1)} + \overrightarrow{\mathbf{b}}_{g}) \tag{14}$$

$$\overrightarrow{\mathbf{i}}^{(t)} = \sigma(\overrightarrow{\mathbf{W}}_{ix}\mathbf{x}^{(t)} + \overrightarrow{\mathbf{W}}_{ih}\overrightarrow{\mathbf{h}}^{(t-1)} + \overrightarrow{\mathbf{b}}_{i}) \tag{15}$$

$$\overrightarrow{\mathbf{f}}^{(t)} = \sigma(\overrightarrow{\mathbf{W}}_{fx}\mathbf{x}^{(t)} + \overrightarrow{\mathbf{W}}_{fh}\overrightarrow{\mathbf{h}}^{(t-1)} + \overrightarrow{\mathbf{b}}_{f}) \tag{16}$$

$$\overrightarrow{\mathbf{o}}^{(t)} = \sigma(\overrightarrow{\mathbf{W}}_{ox}\mathbf{x}^{(t)} + \overrightarrow{\mathbf{W}}_{oh}\overrightarrow{\mathbf{h}}^{(t-1)} + \overrightarrow{\mathbf{b}}_{o}) \tag{17}$$

$$\overrightarrow{\mathbf{c}}^{(t)} = \overrightarrow{\mathbf{g}}^{(t)} \odot \overrightarrow{\mathbf{i}}^{(t)} + \overrightarrow{\mathbf{c}}^{(t-1)} \odot \overrightarrow{\mathbf{f}}^{(t)} \tag{18}$$

$$\overrightarrow{\mathbf{h}}^{(t)} = \tanh(\overrightarrow{\mathbf{c}}^{(t)}) \odot \overrightarrow{\mathbf{o}}^{(t)} \tag{19}$$

where the $\rightarrow$ indicates the forward pass network parameter, while the backward LSTM cell is defined as:

$$\overleftarrow{\mathbf{g}}^{(t)} = \tanh(\overleftarrow{\mathbf{W}}_{gx}\mathbf{x}^{(t)} + \overleftarrow{\mathbf{W}}_{gh}\overleftarrow{\mathbf{h}}^{(t+1)} + \overleftarrow{\mathbf{b}}_{g}) \tag{20}$$

$$\overleftarrow{\mathbf{i}}^{(t)} = \sigma(\overleftarrow{\mathbf{W}}_{ix}\mathbf{x}^{(t)} + \overleftarrow{\mathbf{W}}_{ih}\overleftarrow{\mathbf{h}^{(t+1)}} + \overleftarrow{\mathbf{b}}_{i}) \tag{21}$$

$$\overleftarrow{\mathbf{f}}^{(t)} = \sigma(\overleftarrow{\mathbf{W}}_{fx}\mathbf{x}^{(t)} + \overleftarrow{\mathbf{W}}_{fh}\overleftarrow{\mathbf{h}}^{(t+1)} + \overleftarrow{\mathbf{b}}_{f}) \tag{22}$$

$$\overleftarrow{\mathbf{o}}^{(t)} = \sigma(\overleftarrow{\mathbf{W}}_{ox}\mathbf{x}^{(t)} + \overleftarrow{\mathbf{W}}_{oh}\overleftarrow{\mathbf{h}}^{(t+1)} + \overleftarrow{\mathbf{b}}_{o}) \tag{23}$$

$$\overleftarrow{\mathbf{c}}^{(t)} = \overleftarrow{\mathbf{g}}^{(t)} \odot \overleftarrow{\mathbf{i}}^{(t)} + \overleftarrow{\mathbf{c}}^{(t+1)} \odot \overleftarrow{\mathbf{f}}^{(t)} \tag{24}$$

$$\overleftarrow{\mathbf{h}}^{(t)} = \tanh(\overleftarrow{\mathbf{c}}^{(t)}) \odot \overleftarrow{\mathbf{o}}^{(t)} \tag{25}$$

where the $\leftarrow$ indicates the backward pass network parameter.

Finally, in this assignment we base the prediction on the last hidden state of forward and backward pass and concatenate these hidden representations along the hidden dimension:

$$\mathbf{H} = [\overrightarrow{\mathbf{h}}^{(T)}; \overleftarrow{\mathbf{h}}^{(0)}] \tag{26}$$

$$\mathbf{p}_b^{(t)} = \mathbf{W}_{ph}\mathbf{H} + \mathbf{b}_p \tag{27}$$

$$\hat{\mathbf{y}}^{(t)} = \text{softmax}(\mathbf{p}^{(t)}) \tag{28}$$

The final layer represents a single linear layer taking the concatenated hidden states as input followed by a softmax activation function yielding the class probabilities.

Gated Recurrent Unit(GRU)

The GRU was first introduced in the work of *Cho et al. 2014* [3]. It combines the forget and input gates from LSTM models into a single "update gate". It also merges the cell and hidden state avoiding the use of a memory unit like the LSTM. The resulting model is simpler than standard LSTM models, and has been growing increasingly popular.

The equations below describe a GRU:

$$\mathbf{z}^{(t)} = \sigma(\mathbf{W}_z[\mathbf{h}^{(t-1)}, \mathbf{x}^{(t)}]) \tag{29}$$

$$\mathbf{r}^{(t)} = \sigma(\mathbf{W}_r[\mathbf{h}^{(t-1)}, \mathbf{x}^{(t)}]) \tag{30}$$

$$\tilde{\mathbf{h}}^{(t)} = \tanh(\mathbf{W}[\mathbf{r}^{(t)} * \mathbf{h}^{(t-1)}, \mathbf{x}^{(t)}]) \tag{31}$$

$$\mathbf{h}^{(t)} = (1 - \mathbf{z}^{(t)})\mathbf{h}^{(t-1)} + \mathbf{z}^{(t)}\tilde{\mathbf{h}}^{(t)} \tag{32}$$

$$\mathbf{p}^{(t)} = \mathbf{W}_{ph}\mathbf{h}^{(t)} + \mathbf{b}_p \tag{33}$$

$$\hat{\mathbf{y}}^{(t)} = \text{softmax}(\mathbf{p}^{(t)}). \tag{34}$$

LSTM with Peephole Connections (Peephole LSTM)

LSTM with peephole connections was introduced in the work of *Gers et al. 2002* [4]. The name originates from the introduction of "peephole connections" from the cell state to the gate inputs; essentially allowing gates to "look and consider" the cell state. Some variants of the Peephole LSTMs choose which gates to "peep" a hole to and some chose to "peep" through all of them.

In this assignment we will use the following Peephole LSTM definition:

$$\mathbf{f}^{(t)} = \sigma(\mathbf{W}_{fx}\mathbf{x}^{(t)} + \mathbf{W}_{fh}\mathbf{c}^{(t-1)} + \mathbf{b}_f) \tag{35}$$

$$\mathbf{i}^{(t)} = \sigma(\mathbf{W}_{ix}\mathbf{x}^{(t)} + \mathbf{W}_{ih}\mathbf{c}^{(t-1)} + \mathbf{b}_i) \tag{36}$$

$$\mathbf{o}^{(t)} = \sigma(\mathbf{W}_{ox}\mathbf{x}^{(t)} + \mathbf{W}_{oh}\mathbf{c}^{(t-1)} + \mathbf{b}_o) \tag{37}$$

$$\mathbf{c}^{(t)} = \sigma(\mathbf{W}_{cx}\mathbf{x}^{(t)} + \mathbf{b}_c) \odot \mathbf{i}^{(t)} + \mathbf{c}^{(t-1)} \odot \mathbf{f}^{(t)} \tag{38}$$

$$\mathbf{h}^{(t)} = \tanh(\mathbf{c}^{(t)}) \odot \mathbf{o}^{(t)} \tag{39}$$

$$\mathbf{p}^{(t)} = \mathbf{W}_{ph}\mathbf{h}^{(t)} + \mathbf{b}_p \tag{40}$$

$$\hat{\mathbf{y}}^{(t)} = \text{softmax}(\mathbf{p}^{(t)}). \tag{41}$$

Question 1.4 (20 points)

Implement your assigned LSTM model variant as specified above in the corresponding file (`gru.py`, `bi_lstm.py` or `peep_lstm.py`). Just like for the LSTM from question 1.3, you are required to implement the model without any high-level PyTorch functions. You do not need to implement the backward pass yourself, but instead can rely on the automatic differentiation. For the optimization part we have prepared the code in `train.py`.
Using your assigned sequence toy dataset as training data, perform the same experiments you have done in *Question 1.3* with your assigned hyperparameter settings. Train the network until convergence. Again, create a plot of your results for the sequence lengths specified for your group and average over 3 random seeds. Also report the standard deviation of the experiments. Compare your results to the results you got with the LSTM from *Question 1.3*. (up to a 10-sentences paragraph + Figures)

As a sanity check, your model should obtain:

- GRU: perfect accuracy for $T = 6$
- biLSTM: perfect accuracy for $T = 5$
- peepLSTM: perfect accuracy for $T = 6$

where T is the sequence length.

2 Recurrent Nets as Generative Model **(Total: 30+5 points)**

In this assignment you will build an LSTM for generation of text. By training an LSTM to predict the next character in a sentence, the network will learn local structure in text. You will train a two-layer LSTM on sentences from a book and use the model to generate new text. Before starting, we recommend reading the blog post **The Unreasonable Effectiveness of Recurrent Neural Networks** and take a look at the **PyTorch recurrent network documentation**.

Given a training sequence $\mathbf{x} = (x^1, \ldots, x^T)$, a recurrent neural network can use its output vectors $\mathbf{p} = (p^1, \ldots, p^T)$ to obtain a sequence of predictions $\mathbf{y}^{(t)}$. In the first part of this assignment you have used the recurrent network as *sequence-to-one* mapping, here we use

the recurrent network as *sequence-to-sequence* mapping. The total cross-entropy loss can be computed by averaging over all timesteps using the target labels $\mathbf{y}^{(t)}$.

$$\mathcal{L}^{(t)} = -\sum_{k=1}^{K} \mathbf{y}_k^{(t)} \log \hat{\mathbf{y}}_k^{(t)} \tag{42}$$

$$\mathcal{L} = \frac{1}{T} \sum_{t} \mathcal{L}^{(t)} \tag{43}$$

Again, k runs over the number of classes (vocabulary size). In this expression, $\mathbf{y}$ denotes a one-hot vector of length K containing true labels. Using this sequential loss, you can train a recurrent network to make a prediction at every timestep. The LSTM can be used to generate text, character by character that will look similar to the original text. Just like multi-layer perceptrons, LSTM cells can be stacked to create deeper layers for increased expressiveness. Each recurrent layer can be unrolled in time.

For training you can use a large text corpus such as publicly available books. We provide a number of books in the `assets` directory. However, you are also free to download other books, we recommend Project Gutenberg as good source. Make sure you download the books in plain text (.txt) for easy file reading in Python. We provide the `TextDataset` class for loading the text corpus and drawing batches of example sentences from the text.

The sequence length specifies the length of training sentences which also limits the number of timesteps for backpropagation in time. When setting the sequence length to 30 steps, the gradient signal will never backpropagate more than 30 timesteps. As a result, the network cannot learn text dependencies longer than this number of characters.

Question 2.1 a) (10 points)

Study the code and its outputs in `part2/dataset.py` to sample sentences from the book to train with. Also, have a look at the parameters defined in `part2/train.py`. You need to implement the corresponding PyTorch code in `part2/train.py` to make the features work. You may need to tune them depending on your own implementation.

(a) (10 points) Implement a *two*-layer LSTM network to predict the next character in a sentence by training on sentences from a book. You are allowed to use the PyTorch function `nn.LSTM`. Train the model on sentences of length $T = 30$ from your book of choice. Define the total loss as average of cross-entropy losses over all 30 timesteps (Equation 43).
Plot the loss and accuracy during training, and report all the relevant hyperparameters that you used and shortly explain why you used them (even if they are the default ones). You can use TensorBoard for creating the plots. For convenience use only a train set, no validation nor test sets. (answer up to a 8-sentences paragraph + Figures).
Hint: To train the model efficiently, we recommend taking a look at the concept of teacher forcing.

Question 2.1 b) (10 points)

(b) (10 points) Make the network generate new sentences of length $T = 30$ after 1/3, 2/3 and 3/3 of your training iterations. You can do this by randomly setting the first character of the sentence, and always picking the token with the highest probability predicted by your model. Store your hidden state in between token generations to prevent doing duplicate calculations. Report 5 text samples with different start characters generated by the network over the different stages of training. Carefully study the text generated by your network. What changes do you observe when the training process evolves? For your dataset, some patterns might be better visible when generating sentences longer than 30 characters. Discuss the difference in the quality of sentences generated (e.g. coherency) for a sequence length of less and more than 30 characters. (answer up to a 10-sentences paragraph)

Question 2.1 c) (10 points)

(c) (10 points) Your current implementation uses *greedy sampling*: the next character is always chosen by selecting the one with the highest probability. On the complete opposite, we could also perform *random sampling*: this will result in a high diversity of sequences but they will be meaningless. However, we can interpolate between these two extreme cases by using a *temperature* parameter τ in the softmax:

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

(for details, see Goodfellow *et al.*; Section 17.5.1).

- Explain the effect of the temperature parameter τ on the sampling process.
- Extend your current model by adding the temperature parameter τ to balance the sampling strategy between fully-greedy and fully-random.
- Report generated sentences for temperature values $\tau \in \{0.5, 1.0, 2.0\}$ of your fully trained model. What do you observe for different temperatures? What are the differences with respect to the sentences obtained by greedy sampling?

(up to a 15-sentences paragraph)
Note that using one single dataset is sufficient to get full points. However, we encourage you to experiment using different datasets to gain more insight. We suggest to start with relatively small (and possibly with simple language) datasets, such as Grim's fairy tales, so that you can train the model on your laptop easily. If the network needs training for some hours until convergence, it is advised to run training on the SurfSara cluster. Also, you might want to save the model checkpoints now and then so you can resume training later or generate new text using the trained model.

Bonus Question 2.2 (5 points)

It could be fun to make your network finish some sentences that are related to the book that your are training on. For example, if you are training on Grimm's Fairytales, you could make the network finish the sentence "*Sleeping beauty is ...*". Be creative and test the capabilities of your model. What do you notice? Discuss what you see. (up to 5 sentences)

3 Graph Neural Networks **(Total: 25 points)**

Make sure to check the UvA Deep Learning Tutorial 7 about GNNs (link) before continuing with this part.

3.1 GCN Forward Layer

Graph convolutional neural networks are widely known architectures used to work with graph-structured data, and a particular version (GCN, or Graph Convolutional Network) was firstly introduced in https://arxiv.org/pdf/1609.02907.pdf. Consider Eq. 44, describing the propagation rule for a layer in the Graph Convolutional Network architecture to answer the following questions.

$$H^{(l+1)} = \sigma(\hat{A}H^{(l)}W^{(l)}) \tag{44}$$

Where $\hat{A}$ is obtained by:

$$\hat{A} = \tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}} \tag{45}$$

$$\tilde{A} = A + I_N \tag{46}$$

$$\tilde{D}_{ii} = \sum_j \tilde{A}_{ij} \tag{47}$$

In the equations above, $H^{(l)}$ is the $N \times d$ matrix of activations in the l-th layer, A is the $N \times N$ adjacency matrix of the undirected graph, I_N is an identity matrix of size N, and $\tilde{D}$ is a diagonal matrix used for normalization (you don't need to care about this normalization step, instead, you should focus on discussing Eq. 44). $\tilde{A}$ is the adjacency matrix considering self-connections, N is the number of nodes in the graph, d is the dimension of the feature vector for each node. The adjacency matrix A is usually obtained from data (either by direct vertex attribute or by indirect training). W is a learnable $d^{(l)} \times d^{(l+1)}$ matrix utilized to change the dimension of feature per vertex during the propagation over the graph.

Question 3.1 (6 points)

(a) (4 points) Describe how this layer exploits the structural information in the graph data, and how a GCN layer can be seen as performing message passing over the graph. (up to 5 sentences)

(b) (2 points) There are drawbacks in the presented GCN regarding memory scalabilty and limiting assumptions of the model. Explain one of the those drawbacks and propose a solution to overcome it. (up to 4 sentences)

Question 3.2 (4 points)

Consider the following graph in Figure 3 for the following questions:

(a) (2 points) Give the adjacency matrix $\tilde{A}$ for the graph as specified in Equation 46. (1 line with the matrix. It's a big matrix though.)

(b) (2 points) How many updates (as defined in Equation 44) will it take to forward the information from node C to node E? (up to 2 sentences)

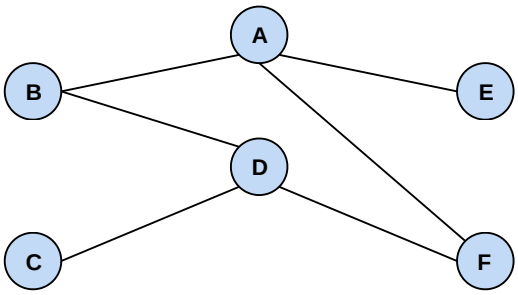


Figure 3. Example graph for **Question 3.2**. The graph consists of 6 nodes ($\mathcal{V} = \{A, B, C, D, E, F\}$), where connections between them represent the undirected edges of the graph.

3.2 Graph Attention Networks

Following we introduce the concept of attention to Graph Neural Networks. We can express the Graph Convolution Layer from equation 44 as follows:

$$h_i^{(l+1)} = \sigma\left(\sum_{j \in \mathcal{N}(i)} \frac{1}{\sqrt{D_{ii} D_{ij}}} W^{(l)} h_j^{(l)}\right) \tag{48}$$

Notice this is the exact same operation than in eq. 44, we only changed the notation of the nodes. The now equation indexes over each node of the graph such that $h_i^{(l)} \in \mathbb{R}^d$ is a vector of activations for a node i in the l-th layer. $\mathcal{N}(i)$ defines the set of neighbors for that node i (including self-connections). Notice this notation shows more clearly the aggregation of neighbor nodes. For simplicity, from now on we will ignore the normalizing factor $\frac{1}{\sqrt{D_{ii} D_{ij}}}$ leading to the following equation:

$$h_i^{(l+1)} = \sigma\left(\sum_{j \in \mathcal{N}(i)} W^{(l)} h_j^{(l)}\right) \tag{49}$$

Question 3.3 (3 points)

We want our Graph Convolutional Layer from equation 49 to attend over the neighbor nodes of node i. What would you add to the equation in order perform attention? Provide the updated equation and justify your changes. If you get stuck you can take a look at Graph Attention Networks paper. (up to 4 sentences + 1 line for the equation)

3.3 Applications of GNNs

Models based on Graph Neural Networks can be efficiently applied for a variety of real-world applications where data can be described in form of graphs.

Question 3.4 (4 points)

Take a look at the publicly available literature and name 2 real-world applications in which GNNs could be applied and justify your answers. (up to 6 sentences)

3.4 Comparing and Combining GNNs and RNNs

Structured data can often be represented in different ways. For example, images can be represented as graphs, where neighboring pixels are nodes connected by edges in a pixel graph. Also sentences can be described as graphs (in particular, trees) if we consider their Dependency Tree representation. On the other direction, graphs can be represented as sequences too, for example through DFS or BFS ordering of nodes. Based on this idea, answer the following open-answer questions (notice that there is not a unique, correct answer in this case, so try to discuss your ideas, supporting them with what you believe are valid motivations).

Question 3.5 (8 points)

(a) (6 points) Consider using RNN-based models to work with sequence representations, and GNNs to work with graph representations of some data. Discuss what the benefits are of choosing either one of the two approaches in different situations. For example, in what tasks (or datasets) would you see one of the two outperform the other? What representation is more expressive for what kind of data? (up to 6 sentences)

(b) (2 points) Think about how GNNs and RNNs models could be used in a combined model, and for what tasks this model could be used. Feel free to mention examples from literature to answer this question. (up to 5 sentences)

Report

We expect each student to write a small report about recurrent neural networks with explicitly answering the questions in this assignment. Please clearly mark each answer by a heading indicating the question number. Again, use the NIPS LaTeX template as was provided for the first assignment.

Deliverables

Create ZIP archive containing your report and all Python code. Please preserve the directory structure as provided in the Github repository for this assignment. Give the ZIP file the following name: `lastname_assignment2.zip` where you insert your lastname. Please submit your deliverable through Canvas. We cannot guarantee a grade for the assignment if the deliverables are not handed in according to these instructions.

The deadline for this assignment is November 29th at 23:59.

References

[1] Mike Schuster and K. Paliwal. Bidirectional recurrent neural networks. *IEEE Trans. Signal Process.*, 45:2673–2681, 1997. 6

[2] A. Graves and J. Schmidhuber. Framewise phoneme classification with bidirectional lstm and other neural network architectures. *Neural networks : the official journal of the International Neural Network Society*, 18 5-6:602–10, 2005. 6

[3] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014. 7

[4] Felix A Gers, Nicol N Schraudolph, and Jürgen Schmidhuber. Learning precise timing with lstm recurrent networks. *Journal of machine learning research*, 3(Aug):115–143, 2002. 8