Assignment 2. Recurrent Neural Networks and Graph Neural Networks

University of Amsterdam – Deep Learning Course

November 27, 2019

The deadline for this assignment is November 29th 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). Recurrent neural networks 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. These networks are very powerful and have found their way into many production environments. For example **Google's neural machine translation system** relies on Long-Short Term Networks (LSTMs). Graph Neural Networks 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 vanilla RNNs and LSTMs on a simple toy problem. This will help you understand the fundamentals of recurrent networks. After that, you will use LSTMs 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 questions which you need to answer. We expect each student to hand in their code and individually write a report that explains the code and answers the questions.

Before continuing with the first assignment, we highly recommend each student to read this excellent blogpost by Chris Olah on recurrence neural networks: **Understanding LSTM Networks**. For the second part of the assignment, you also might want to have a look at the **PyTorch recurrent network documentation**.

1 Vanilla RNN versus LSTM **(Total: 50 points)**

For the first task, you will compare vanilla Recurrent Neural Networks (RNN) with Long-Short Term Networks (LSTM). PyTorch has a large amount of building blocks for recurrent neural networks. However, to get you familiar with the concept of recurrent connections, in this first part of this assignment you will implement a vanilla RNN and LSTM from scratch. The use of high-level operations such as `torch.nn.RNN`, `torch.nn.LSTM` and `torch.nn.Linear` is not allowed until the second part of this assignment.

1.1 Toy Problem: Palindrome Numbers

In this first assignment, we will focus on very simple sequential training data for understanding the memorization capability of recurrent neural networks. More specifically,

we will study *palindrome* numbers. Palindromes are numbers which read the same backward as forward, such as:

303

4224

175282571

682846747648286

We can use a recurrent neural network to predict the next digit of the palindrome at every timestep. While very simple for short palindromes, this task becomes increasingly difficult for longer palindromes. For example when the network is given the input 68284674764828_ and the task is to predict the digit on the _ position, the network has to remember information from 14 timesteps earlier. If the task is to predict the last digit only, the intermediate digits are irrelevant. However, they may affect the evolution of the dynamic system and possibly erase the internally stored information about the initial values of input. In short, this simple problem enables studying the memorization capability of recurrent networks.

For the coding assignment, in the file `part1/dataset.py`, we have prepared the class `PalindromeDataset` which inherits from `torch.utils.data.Dataset` and contains the function `generate_palindrome` to randomly generate palindrome numbers. You can use this dataset directly in PyTorch and you do not need to modify contents of this file. Note that for short palindromes the number of possible numbers is rather small, but we ignore this sampling collision problem for the purpose of this assignment.

1.2 Vanilla RNN in PyTorch

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.

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 logits $\hat{\mathbf{y}}^{(t)}$ 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 the task of predicting the final palindrome number, we compute the standard cross-entropy loss *only* over the last timestep:

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

Where k runs over the number of classes ($K = 10$ because we have ten digits). In this expression, $\mathbf{y}$ denotes a one-hot vector of length K containing true labels.

Question 1.1 (10 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 $\mathbf{W}_{hx}$. 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.
Do the same for $\frac{\partial \mathcal{L}^{(T)}}{\partial \mathbf{W}_{hh}}$. 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.

Question 1.2 (10 points)

Implement the vanilla recurrent neural network as specified by the equations above in the file `vanilla_rnn.py`. For the *forward* pass you will need Python's `for`-loop to step through time. You need to initialize the variables and matrix multiplications yourself without using high-level PyTorch building blocks. The weights and biases can be initialized using `torch.nn.Parameter`. The *backward* pass does not need to be implemented by hand, instead you can rely on automatic differentiation and use the RMSProp optimizer for tuning the weights. We have prepared boilerplate code in `part1/train.py` which you should use for implementing the optimization procedure.

Question 1.3 (5 points)

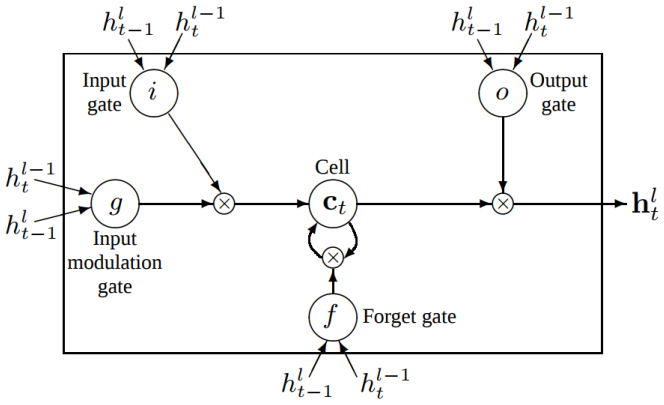
As the recurrent network is implemented, you are ready to experiment with the memorization capability of the vanilla RNN. Given a palindrome of length T, use the first $T - 1$ digits as input and make the network predict the last digit. The network is *successful* if it correctly predicts the last digit and thus was capable of memorizing a small amount of information for T timesteps.

Start with short palindromes ($T = 5$), train the network until convergence and record the accuracy. Repeat this by gradually increasing the sequence length and create a plot that shows the accuracy versus palindrome length. As a sanity check, make sure that you obtain a near-perfect accuracy for $T = 5$ with the default parameters provided in `part1/train.py`. To plot your results, evaluate your trained model for each palindrome length on a large enough separate set of palindromes, and repeat the experiment with different seeds to display stable results.

Question 1.4 (5 points)

To improve optimization of neural networks, many variants of stochastic gradient descent have been proposed. Two popular optimizers are RMSProp and Adam and/or the use of momentum. In practice, these methods are able to converge faster and obtain better local minima. In your own words, write down the benefits of such methods in comparison to vanilla stochastic gradient descent. Your answer needs to touch upon the concepts of **momentum** and **adaptive learning rate**.

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



1.3 Long-Short Term Network (LSTM) in PyTorch

As you have observed after implementing the previous questions, 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 Network (LSTMs). An LSTM (Figure 1) introduces a number of gating mechanisms to improve gradient flow for better training. Before continuing, please read the following blogpost: **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 1.5 (5 points)

(a) (3 points) The LSTM extends the vanilla RNN cell by adding four gating mechanisms. Those gating mechanisms are crucial for successfully training recurrent neural networks. 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.

(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.

Question 1.6 (10 points)

Implement the LSTM network as specified by the equations above in the file `lstm.py`. Just like for the Vanilla RNN, 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 you can rely on automatic differentiation and use the RMSProp optimizer for tuning the weights. For the optimization part we have prepared the code in `train.py`.

Using the palindromes as input, perform the same experiment you have done in *Question 1.3*. Train the network until convergence. You might need to adjust the learning rate when increasing the sequence length. The initial parameters in the prepared code provide a starting point. Again, create a plot of your results by gradually increasing the sequence length. Write down a comparison with the vanilla RNN and think of reasons for the different behavior. As a sanity check, your LSTM should obtain near-perfect accuracy for $T = 5$.

We have now implemented the vanilla RNN and LSTM networks and have compared the difference in their performance at the task of palindrome prediction. We will now study the difference of their *temporal dependence of the gradients* as discussed in **Question 1.1** for both the variants of the RNNs in more detail.

Question 1.7 (5 points)

Modify your implementations of RNN and LSTM to obtain the gradients between time steps of the sequence, namely $\frac{dL}{dh^t}$. You do not have to train a network for this task. Take as input a palindrome of length 100 and predict the number at the final time step of the sequence. Note that the gradients over time steps does not imply the gradients of the RNN/LSTM cell blocks, but the message that gets passed between time-steps (i.e, the hidden state). Plot the gradient magnitudes for both the variants over different time steps and explain your results. Do the results correlate with the findings in **Question 1.6**? What results will you expect if we actually train the network instead for this task? Submit your implementation as a separate file called `grads_over_time.py`.

`Hint:` In PyTorch you can declare variables with requires_grad = True to add them to the computation graph and get their gradients when back-propagating the loss. Use this to extract gradients from the hidden state between time steps.

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 next. Before starting, we recommend reading the blog post **The Unreasonable Effectiveness of Recurrent Neural Networks**.

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 $\hat{\mathbf{y}}^{(t)}$. 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 $\mathbf{y}^{(t)}$.

$$\mathcal{L}^{(t)} = -\sum_{k=1}^{K} \mathbf{y}_k^{(t)} \log \hat{\mathbf{y}}_k^{(t)} \tag{12}$$

$$\mathcal{L} = \frac{1}{T} \sum_{t} \mathcal{L}^{(t)} \tag{13}$$

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 files `train_text.py` and `model.py` provide a starting point for your implementation. 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 (30 points)

We recommend reading **PyTorch's documentation on RNNs** before you start. Study the code and its outputs for `part2/dataset.py` to sample sentences from the book to train with. Also, have a look at the parameters defined in `part2/train.py` and implement the corresponding PyTorch code to make the features work. We obtained good results with the default parameters as specified, but 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. Train the model on sentences of length $T = 30$ from your book of choice. Define the total loss as average of cross-entropy loss over all timesteps (Equation 13). Plot the model's loss and accuracy during training, and report all the relevant hyperparameters that you used and shortly explain why you used them.

(b) (10 points) Make the network generate new sentences of length $T = 30$ now and then by randomly setting the first character of the sentence. Report 5 text samples generated by the network over 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.

(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 $\tau \in \{0.5, 1.0, 2.0\}$. What do you observe for different temperatures? What are the differences with respect to the sentences obtained by greedy sampling?

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 (2 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.

Bonus Question 2.3 (3 points)

There is also another method that could be used for sampling: Beam Search. Shortly describe how it works, and mention when it is particularly useful. Implement it (you can choose the beam size), and discuss the results.

3 Graph Neural Networks (Total: 20 points)

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. 14, describing the propagation rule for a layer in the Graph Convolutional Network architecture to answer the following questions.

$$H^{(l+1)} = \sigma(\hat{A}H^{(l)}W^{(l)}) \tag{14}$$

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

$$\hat{A} = \tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}} \tag{15}$$

$$\tilde{A} = A + I_N \tag{16}$$

$$\tilde{D}_{ii} = \sum_j \tilde{A}_{ij} \tag{17}$$

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. 14). $\tilde{A}$ 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.

(b) (2 points) There are drawbacks to GCNs. Mention one, and explain how we can overcome this.

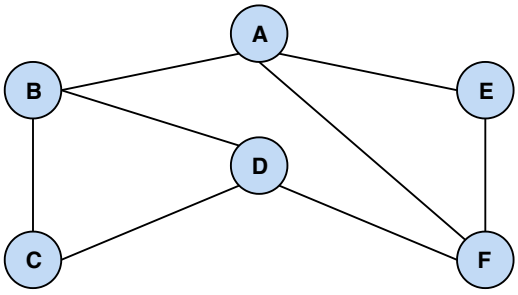


Figure 2. Example graph for **Question 3.2**. The graph consists of 6 nodes ($\mathcal{V} = \{A, B, C, D, E, F\}$), where connections between them represent the undirected edges of the graph.

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 16.

(b) (2 points) How many updates (as defined in Equation 14) will it take to forward the information from node C to node E?

3.2 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 3.3 (2 points)

Take a look at the publicly available literature and name a few real-world applications in which GNNs could be applied.

3.3 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 3.4 (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?

(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.

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.