# Deep Learning - Assignment 2: Recurrent Neural Networks

Ioannis Gatopoulos 12141666

May 19, 2019

Assignment 2 of Deep Learning Course 2019
MSc Artificial Intelligence • University of Amsterdam

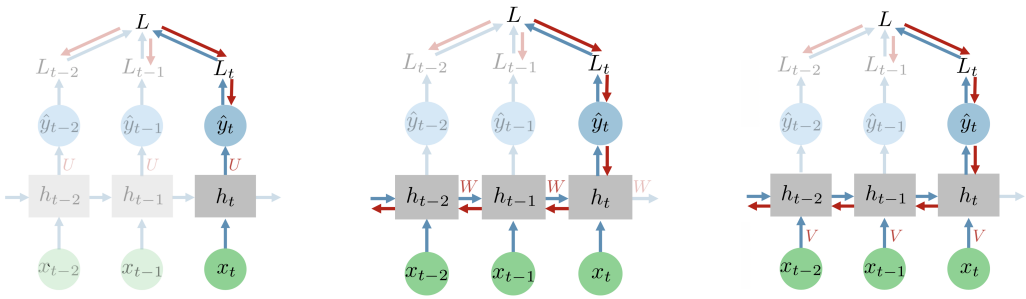
**Abstract**

In this report we mainly study and implement recurrent neural networks (RNNs). Initially, after exploring the fundamentals behind these kind of networks (back-propagation though time and problems that might occur during training) we implemented a vanilla RNN and a LSTM in order to predict the last number of a palindrome sequence. Though this example, we projected the superiority of the latter's to capture long-term dependencies efficiently, after their architecture was outlined. After we train a LSTM network in a book, we raised the bar and generate text from it with various of sampling techniques. Finally, we describe Graph Neural Networks (GNNs), in particularly the Graph Convolution Network, and explore the possible combination but also the differences between the RNNs.

## Contents

# 1 Vanilla RNN versus LSTM

## 1.1 Vanilla RNN in PyTorch




(a) BBTT - $\partial\mathcal{L}^{(t)}/\partial\mathbf{W}_{ph}$ (b) BBTT - $\partial\mathcal{L}^{(t)}/\partial\mathbf{W}_{hh}$ (c) BBTT - $\partial\mathcal{L}^{(t)}/\partial\mathbf{W}_{hx}$

Figure 1: Illustration of back-propagation through time (BBTT) for time-step $t$ of an (unfolded) RNN

**Question 1.1**

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

$$\begin{aligned}
\tilde{\mathbf{h}}^{(t)} &= \mathbf{W}_{hx}\mathbf{x}^{(t)} + \mathbf{W}_{hh}\mathbf{h}^{(t-1)} + \mathbf{b}_h \\
\mathbf{h}^{(t)} &= \tanh\left(\tilde{\mathbf{h}}^{(t)}\right) \\
\mathbf{p}^{(t)} &= \mathbf{W}_{ph}\mathbf{h}^{(t)} + \mathbf{b}_p \\
\hat{\mathbf{y}}^{(t)} &= \text{softmax}\left(\mathbf{p}^{(t)}\right) \\
\mathcal{L}^{(t)} &= -\sum_{k=1}^{K} \mathbf{y}_k^{(t)} \log \hat{\mathbf{y}}_k^{(t)}
\end{aligned}$$

With RNNs we can have an **arbitrary** sequence of length, using a fixed number of inputs at each time step and they address the problem of having large number of parameters by **sharing** all of them across time steps.

**i)** In general, for the gradient of loss function $\mathcal{L}$ wrt the weight matrix $\mathbf{W}_{ph}$ we would have

$$\frac{\partial\mathcal{L}}{\partial\mathbf{W}_{ph}} = \sum_{i=0}^{T} \frac{\partial\mathcal{L}_i^{(T)}}{\partial\mathbf{W}_{ph}}$$

Thus, for a specific time step, and in particular the last one ($T$), the above turns into

$$\frac{\partial\mathcal{L}^{(T)}}{\partial\mathbf{W}_{ph}} = \sum_k \frac{\partial\mathcal{L}^{(T)}}{\partial\hat{\mathbf{y}}_k{}^{(T)}} \frac{\partial\hat{\mathbf{y}}_k{}^{(T)}}{\partial\mathbf{W}_{ph}} = \sum_k \frac{\partial\mathcal{L}^{(T)}}{\partial\hat{\mathbf{y}}_k{}^{(T)}} \frac{\partial\hat{\mathbf{y}}_k{}^{(T)}}{\partial\mathbf{p}_k^{(T)}} \frac{\partial\mathbf{p}_k^{(T)}}{\partial\mathbf{W}_{ph}}$$

The back-propagation is visualized at Figure 1a. Because from now we will mostly refer to time-step $T$, and to make the equations clearer for the reader, when the reference to time-step is missing, it will refer to time-step $T$, unless its explicitly noted otherwise. For example, $\hat{y}^T$ will be just $\hat{y}$.

For the first two terms, instead of calculating them separately, we will do a 'trick' by taking advantage the fact that $\mathbf{y}$ is a one-hot-encoded vector. Taking the derivative wtr $\mathbf{p}_i$ we result in

$$\frac{\partial\mathcal{L}}{\partial\mathbf{p}_i} = \frac{\partial}{\partial\hat{\mathbf{p}}_i}\left(-\sum_{k=1}^{K}\mathbf{y}_k \log\hat{\mathbf{y}}\right) = -\sum_{k=1}^{K}\mathbf{y}_k\frac{\partial}{\partial\mathbf{p}_i}\log\hat{\mathbf{y}}_k = -\sum_{k=1}^{K}\mathbf{y}_k\frac{1}{\hat{\mathbf{y}}_k}\frac{\partial}{\partial\mathbf{p}_i}\hat{\mathbf{y}}_k$$

The last term would be

$$\begin{aligned}\frac{\partial}{\partial \mathbf{p}_i}\frac{\exp \mathbf{p}_k}{\sum_j \exp \mathbf{p}_j} &= \frac{1}{\sum_j \exp \mathbf{p}_j}\left(\frac{\partial}{\partial \mathbf{p}_i}\exp \mathbf{p}_k\right) + \exp \mathbf{p}_k \left(\frac{\partial}{\partial \mathbf{p}_i}\frac{1}{\sum_j \exp \mathbf{p}_j}\right) \\ &= \frac{\delta_{ik}\exp \mathbf{p}_k}{\sum_j \exp \mathbf{p}_j} - \frac{\exp \mathbf{p}_i \, \exp \mathbf{p}_k}{\left(\sum_j \exp \mathbf{p}_j\right)^2} \\ &= \delta_{ik}\hat{\mathbf{y}}_k - \hat{\mathbf{y}}_i \cdot \hat{\mathbf{y}}_k = \hat{\mathbf{y}}_k\left(\delta_{ik} - \hat{\mathbf{y}}_i\right)\end{aligned}$$

Putting all of them together

$$\frac{\partial \mathcal{L}}{\partial \mathbf{p}_i} = -\sum_{k=1}^{K} \mathbf{y}_k \frac{1}{\hat{\mathbf{y}}_k}\hat{\mathbf{y}}_k\left(\delta_{ik} - \hat{\mathbf{y}}_i\right) = -\sum_{k=1}^{K} \mathbf{y}_k\left(\delta_{ik} - \hat{\mathbf{y}}_i\right)$$

Assuming that $c$ denotes the true class, we will have

$$\frac{\partial \mathcal{L}}{\partial \mathbf{p}_i} = -\mathbf{y}_c\left(\delta_{ic} - \hat{\mathbf{y}}_i\right) = \mathbf{y}_c\hat{\mathbf{y}}_i - \mathbf{y}_c\delta_{ic} = \hat{\mathbf{y}}_i - \mathbf{y}_i \Rightarrow \frac{\partial \mathcal{L}}{\partial \mathbf{p}} = \hat{\mathbf{y}} - \mathbf{y} \equiv \boldsymbol{\delta}^{(L)}$$

where $\boldsymbol{\delta}^{(L)}$ is called **error term**.

We will derive the last term in scalar form.

$$\frac{\partial \mathbf{p}_k}{\partial \mathbf{W}_{ph_{(ij)}}} = \frac{\partial}{\partial \mathbf{W}_{ph_{(ij)}}}\left(\mathbf{W}_{ph_{(k:)}}\mathbf{h} + \mathbf{b}_{p_{(k)}}\right) = h_j\delta_{ik} = \begin{bmatrix}\mathbf{0} \\ h_j \\ \mathbf{0}\end{bmatrix}$$

$$\Rightarrow \frac{\partial \mathbf{p}}{\partial \mathbf{W}_{ph}} = \begin{bmatrix}\mathbf{0}^T \\ \vdots \\ \mathbf{h}^T \\ \vdots \\ \mathbf{0}^T\end{bmatrix}$$

where $\mathbf{0}$ is a column-vector.

Finally putting them all together, we will have

$$\frac{\partial \mathcal{L}^{(T)}}{\partial \mathbf{W}_{ph}} = \sum_k \frac{\partial \mathcal{L}}{\partial \hat{\mathbf{p}}_k}\frac{\partial \mathbf{p}_k}{\partial \mathbf{W}_{ph}} = \sum_k (\hat{\mathbf{y}}_k - \mathbf{y}_k) \cdot \begin{bmatrix}\mathbf{0}^T \\ \vdots \\ \mathbf{h}^T \\ \vdots \\ \mathbf{0}^T\end{bmatrix} = [\hat{\mathbf{y}} - \mathbf{y}] \cdot h^T \quad \text{or} \quad \boldsymbol{\delta}^{(L)} \cdot h^T$$

*** Second way**: Trick for softmax derivative; Assuming that class $c$ would be the ground truth class, we would have

$$\mathcal{L} = -\sum_i y_i \log \tilde{y}_i = -y_c \log \tilde{y}_c = -y_c \log\left(\frac{\exp\left(p_c\right)}{\sum_{i=1}^{d_N}\exp\left(p\right)_i}\right) = -y_c\, p_c + \log\sum_{i=1}^{d_N}\exp\left(p\right)_i$$

Now taking the derivative wrt $\tilde{x}_j^{(N)}$ we would have

$$\frac{\partial L}{\partial \tilde{x}_j} = -y_c\,\delta_{cj} + \frac{1}{\sum_{i=1}^{d_N}\exp\left(p\right)_i} \cdot \exp\left(p_j\right) = \tilde{y}_j - y_j$$

And we continue as above.

**ii)** Considering only the last time step ($T$), the derivative of the loss $\mathcal{L}$ wrt $\mathbf{W}_{hh}$ would be

$$\frac{\partial \mathcal{L}^{(T)}}{\partial \mathbf{W}_{hh}} = \frac{\partial \mathcal{L}^{(T)}}{\partial \mathbf{p}^{(T)}} \frac{\partial \mathbf{p}^{(T)}}{\partial \mathbf{h}^{(T)}} \frac{\partial \mathbf{h}^{(T)}}{\partial \mathbf{W}_{ph}}$$

The first partial derivative was calculated before, therefore we are going to work our way with the last two.

- $$\frac{\partial \mathbf{p}^{(T)}}{\partial \mathbf{h}^{(T)}} = \frac{\partial}{\partial \mathbf{h}^{(T)}} \left( \mathbf{W}_{ph} \mathbf{h}^{(T)} + \mathbf{b}_p \right) = \mathbf{W}_{ph}^T$$

- $$\begin{aligned} \frac{\partial \mathbf{h}^{(T)}}{\partial \mathbf{W}_{hh}} &= \frac{\partial}{\partial \mathbf{W}_{hh}} \left( \tanh \left( \mathbf{W}_{hx} \mathbf{x}^{(T)} + \mathbf{W}_{hh} \mathbf{h}^{(T-1)} + \mathbf{b}_h \right) \right) \\ &= \operatorname{diag}(1 - \tanh^2(\tilde{\mathbf{h}}^{(T)})) \frac{\partial}{\partial \mathbf{W}_{hh}} \left( \mathbf{W}_{hx} \mathbf{x}^{(T)} + \mathbf{W}_{hh} \mathbf{h}^{(T-1)} + \mathbf{b}_h \right) \\ &= \operatorname{diag}(1 - (\mathbf{h}^{2^{(T)}})) \frac{\partial}{\partial \mathbf{W}_{hh}} \left( \mathbf{W}_{hh} \mathbf{h}^{(T-1)} \right) \\ &\overset{*}{=} \operatorname{diag}(1 - (\mathbf{h}^{2^{(T)}})) \left( \mathbf{W}_{hh}^T \frac{\partial}{\partial \mathbf{W}_{hh}} \mathbf{h}^{(T-1)} + \left( \mathbf{h}^{(T-1)} \right)^T \frac{\partial}{\partial \mathbf{W}_{hh}} \mathbf{W}_{hh} \right) \\ &= \operatorname{diag}(1 - (\mathbf{h}^{2^{(T)}})) \left( \mathbf{h}^{(T-1)} \right)^T + \operatorname{diag}(1 - (\mathbf{h}^{2^{(T)}})) \mathbf{W}_{hh}^T \frac{\partial \mathbf{h}^{(T-1)}}{\partial \mathbf{W}_{hh}} \end{aligned}$$

Observing that

$$\frac{\partial \mathbf{h}^{(T)}}{\partial \mathbf{h}^{(T-1)}} = \frac{\partial \mathbf{h}^{(T)}}{\partial \tilde{\mathbf{h}}^{(T)}} \frac{\partial \tilde{\mathbf{h}}^{(T)}}{\partial \mathbf{h}^{(T-1)}} = \operatorname{diag} \left( 1 - \mathbf{h}^{2^{(T)}} \right) \mathbf{W}_{hh}^T$$

the former can be written equally as

$$\frac{\partial \mathbf{h}^{(T)}}{\partial \mathbf{W}_{hh}} = \operatorname{diag} \left( 1 - \mathbf{h}^{2^{(T)}} \right) \left( \mathbf{h}^{(T-1)} \right)^T + \frac{\partial \mathbf{h}^{(T)}}{\partial \mathbf{h}^{(T-1)}} \frac{\partial \mathbf{h}^{(T-1)}}{\partial \mathbf{W}_{hh}}$$

Extend the last term using this equation (we apply it recursively), we can easily derive that

$$\begin{aligned} \frac{\partial \mathbf{h}^{(T)}}{\partial \mathbf{W}_{hh}} &= \operatorname{diag} \left( 1 - \mathbf{h}^{2^{(T)}} \right) \left( \mathbf{h}^{(T-1)} \right)^T + \frac{\partial \mathbf{h}^{(T)}}{\partial \mathbf{h}^{(T-1)}} \left( \operatorname{diag} \left( 1 - \mathbf{h}^{2^{(T-1)}} \right) \left( \mathbf{h}^{(T-2)} \right)^T + \frac{\partial \mathbf{h}^{(T-1)}}{\partial \mathbf{h}^{(T-2)}} \frac{\partial \mathbf{h}^{(T-2)}}{\partial \mathbf{W}_{hh}} \right) \\ &= \operatorname{diag} \left( 1 - \mathbf{h}^{2^{(T-1)}} \right) \left( \mathbf{h}^{(T-1)} \right)^T + \frac{\partial \mathbf{h}^{(T)}}{\partial \mathbf{h}^{(T-1)}} \operatorname{diag} \left( 1 - \mathbf{h}^{2^{(T-1)}} \right) \left( \mathbf{h}^{(T-2)} \right)^T + \frac{\partial \mathbf{h}^{(T)}}{\partial \mathbf{h}^{(T-1)}} \frac{\partial \mathbf{h}^{(T-1)}}{\partial \mathbf{h}^{(T-2)}} \frac{\partial \mathbf{h}^{(T-2)}}{\partial \mathbf{W}_{hh}} \\ &= \dots \end{aligned}$$

and, in general, for hidden layer ($k$), we result in

$$\frac{\partial \mathbf{h}^{(k)}}{\partial \mathbf{W}_{hh}} = \sum_{t=1}^{k} \left( \prod_{j=t+1}^{k} \frac{\partial \mathbf{h}^{(j)}}{\partial \mathbf{h}^{(j-1)}} \right) \operatorname{diag} \left( 1 - \mathbf{h}^{2^{(t)}} \right) \left( \mathbf{h}^{(t-1)} \right)^T \tag{0.1}$$

Finally, we end up with

$$\frac{\partial \mathcal{L}^{(T)}}{\partial \mathbf{W}_{hh}} = \frac{\partial \mathcal{L}^{(T)}}{\partial \mathbf{p}^{(T)}} \frac{\partial \mathbf{p}^{(T)}}{\partial \mathbf{h}^{(T)}} \frac{\partial \mathbf{h}^{(T)}}{\partial \mathbf{W}_{ph}} = [\hat{\mathbf{y}} - \mathbf{y}]^T \mathbf{W}_{ph} \left( \sum_{t=1}^{T} \left( \prod_{j=t+1}^{T} \frac{\partial \mathbf{h}^{(j)}}{\partial \mathbf{h}^{(j-1)}} \right) \operatorname{diag} \left( 1 - \mathbf{h}^{2^{(t)}} \right) \left( \mathbf{h}^{(t-1)} \right)^T \right)$$

**iii) What difference do you observe in temporal dependence of the two gradients?**

As is it also illustrated at Figure 1a, we saw that the gradient of loss function $\mathcal{L}$ wrt the weight matrix $\mathbf{W}_{ph}$ it is pretty much strait forward. This is because our prediction $\mathbf{p}$ depends only on one

*product rule $\partial(\mathbf{XY}) = (\partial \mathbf{X})\mathbf{Y} + \mathbf{X}(\partial \mathbf{Y})$ [1]

point wrt to $\mathbf{W}_{ph}$. Looking at the equation $\mathbf{p}^{(t)} = \mathbf{W}_{ph}\mathbf{h}^{(t)} + \mathbf{b}_p$, we can indeed confirm that $\mathbf{p}$ (our prediction) depends on $\mathbf{W}_{ph}$ only by the term $\mathbf{W}_{ph}$ ($\mathbf{h}^{(t)}$ does not depend on $\mathbf{W}_{ph}$). However, looking at the same equation, we derive that now the term $\mathbf{h}^{(t)}$ depends on $\mathbf{W}_{hh}$, which the former $\left(\mathbf{h}^{(t)} = \tanh\left(\mathbf{W}_{hx}\mathbf{x}^{(t)} + \mathbf{W}_{hh}\mathbf{h}^{(t-1)} + \mathbf{b}_h\right)\right)$ depends on the latter not in one point, but in two ($\mathbf{W}_{hh}$ and $\mathbf{h}^{(t-1)}$). In other words, there is not only one dependence between the hidden units $\mathbf{h}^{(t)}$ and weight matrix $\mathbf{W}_{hh}$ because the hidden units in the previous time step also rely on $\mathbf{W}_{hh}$. So here, can not just use a simple chain rule like $\partial\mathcal{L}^{(t)}/\partial\mathbf{W}_{ph}$, but we need to go "backward in time" and use a formula of total derivative. The illustration of this chain rule can be seen on Figure 1b.

**iv) Problems that might occur for the gradient $\partial\mathcal{L}^{(t)}/\partial\mathbf{W}_{hh}$ when training this RNN for a large number of timesteps.**

In theory, RNNs are absolutely capable of handling "long-term dependencies" but, sadly, in practice RNNs don not seem to be able to learn them. The problem was explored in depth by Hochreiter (1991) and Bengio, et al. (1994), where they talk about the so called **vanishing gradient** and **exploding gradient** problems.

$$\left\|\frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{h}^{(t-1)}}\right\|_2 < 1 \qquad (0.2) \qquad\qquad \left\|\frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{h}^{(t-1)}}\right\|_2 > 1 \qquad (0.3)$$

As we saw, to compute the gradient of the loss at time step t with respect to $\mathbf{W}_{hh}$, we should sum up the contributions from all the previous time steps to this gradient. Looking at the equation 0.1, we can see that the more steps between time moments $k$ and $T$, the more elements are in this products. So the values of Jacobian matrices have a very strong influence especially on the contributions from faraway steps.
It is clear that if all the Jacobian matrices are less than one in spectral matrix norm ($\|\cdot\|_2$), then their product goes to zero exponentially fast when the number of elements in this product tends to infinity. And on the contrary, if all the Jacobian matrices are more than one in spectral matrix norm, then their product goes to infinity exponentially fast. As a result, in the first case 0.2, the contributions from the faraway steps go to zero and the gradient contains only the information about nearby steps, making it difficult to learn long range dependencies with a simple recurrent neutral network. This problem is usually called the **vanishing gradient** problem, because a lot of elements in the gradient simply vanish and do not affect the training. In the second case 0.3, the contributions from faraway steps grow exponentially fast so the gradient itself grows too. If an input sequence is long enough the gradient may even become a NaN (in practice). This problem is called the **exploding gradient** problem and it makes the training very unstable. The reason is that if the gradient is a large number then we make a long step in the direction of this gradient in the parameter space. Since we optimize a very complex multimodal function and we use stochastic methods we may end up in a very poor point after such step.

### Question 1.2

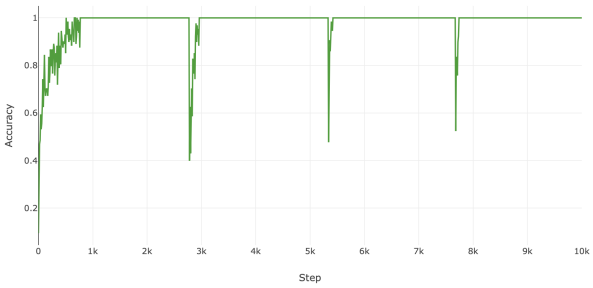
We implemented a Vanilla RNN model and test its performance on a palindrome prediction task.

* Implemented on `part1/vanilla_rnn.py` and `part1/train.py`

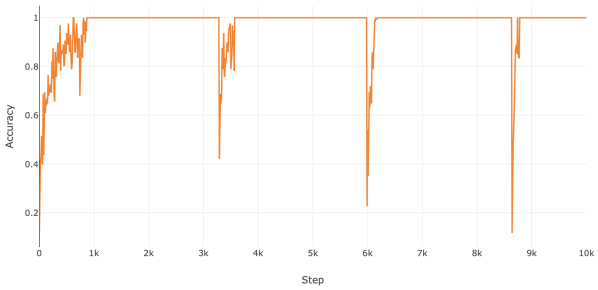
### Question 1.3

According to Figures 2 and 3, we can derive that as the length of the palindrome increases, the RNN performs poorer and poorer. However, for lengths until 40 its performance is quite good. As the sequence of the numbers increase, it becomes more and more difficult for the neural network to curry the first element of the palindrome that actually should predict. This is because the gradient of the loss function decays exponentially with time (called the vanishing gradient problem). This is becomes clear when looking at the network performance on palindromes of length 80 and 160; the accuracy can not pass 0.4 and 0.2.

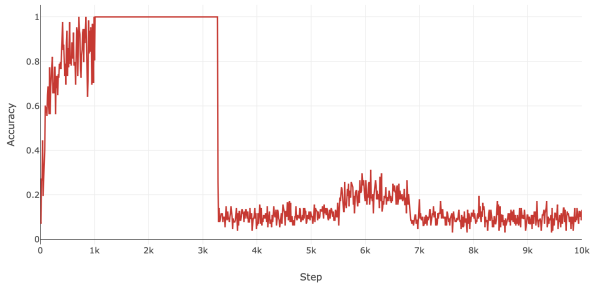
The specifications of the RNN that was used are:
**Learning rate:** `0.001`, **Batch size:** `128`, **Number of hidden units:** `128` and **Length of an input sequence:** `20`.



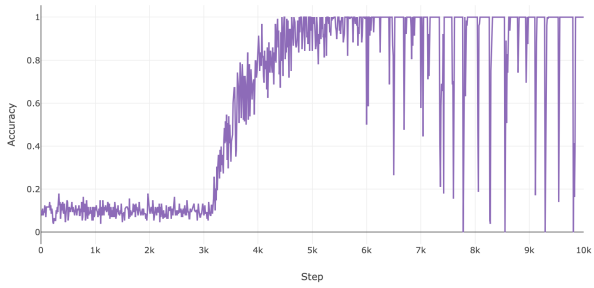

(a) Accuracy performance of `RNN` model with T=5



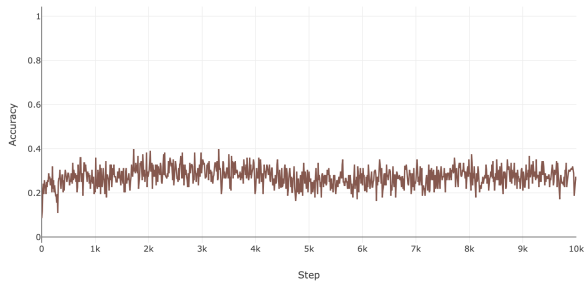

(b) Accuracy performance of `RNN` model with T=10



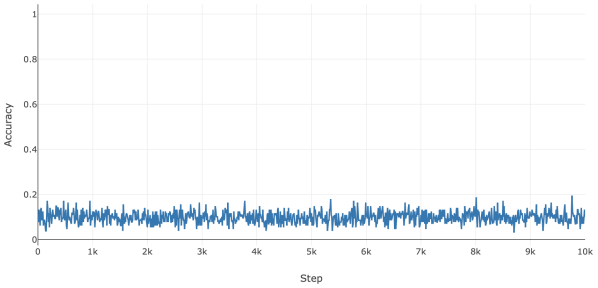

(c) Accuracy performance of `RNN` model with T=20




(d) Accuracy performance of `RNN` model with T=40




(e) Accuracy performance of `RNN` model with T=80




(f) Accuracy performance of `RNN` model with T=160

Figure 2: Performance of vanilla `RNN` model with various lengths T.

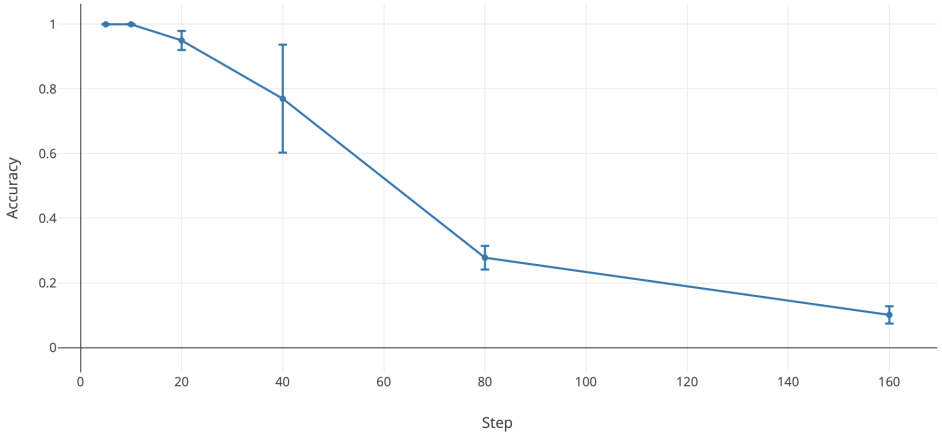
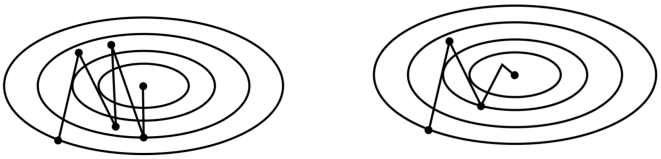



Figure 3: Average RNN accuracies with respect to sequence length T over multiple runs. The error bars indicates the std of our averages.

**Question 1.4: Benefits of optimizers like RMSProp and Adam in comparison to vanilla stochastic gradient descent**

Optimizing difficult functions with SGD, will probably lead to oscillation and thus, the it will take many iteration to coverage to its minimum (4a). On method that have been proposed to overcome this, is

**momentum**. What momentum does is to maintain the gradient direction from the previous steps. The result is that the gradient will make large steps for those component that maintain the same sign on every iteration but close to none to those which change constantly. So the gradient now will have a more smooth route, just like the one on figure 4b. Momentum methods like **nesterov** work better with difficult functions with complex level sets. However, they still require a choice for learning rate and they are sensitive to it.



(a) Optimization using SGD

(b) Optimization using SGD with momentum

Figure 4: Illustration of function optimization with and without momentum. The elliptic level lines indicate that the function is a difficult one.

To overcome this issue, there are some optimization methods that try to choose the learning rate adaptively. An extension of **AdaGrad**, called **RMSprop** proposes to take an exponentially weighted average of squares of gradients on every step.

$$G_j^t = \alpha G_j^{t-1} + (1-\alpha) g_{tj}^2$$
$$w_j^t = w_j^{t-1} - \frac{\eta_t}{\sqrt{G_j^t + \epsilon}} g_{tj}$$

As it chooses separate learning rates for each dimension and tune it individually, osculations that resulted from some dimensions of the data are not longer exist. Thus, this method is very useful for sparse data as well

In this approximation, one can notice that $G_j^t$ has some bias towards zero, especially at first steps, because we initialize it with zero. To get rid of this bias, we can divide it by $1-\alpha^t$ and then, this normalization will be large at the first steps, but it will almost equal to 1 for larger $t$. However, as we already discussed above, approximation of gradient from one step can be noisy and lead to oscillations. So we want to replace the term $g_{tj}$ from $w_j^t$ with a smoother one. To achieve this, we can introduce momentum by maintaining another auxiliary variable, $m$, that is essentially a sum of gradients. Noting $G_j^t$ as $v_j^t$, the final equations are

$$m_j^t = \frac{\beta_1 m_j^{t-1} + (1-\beta_1)\, g_{tj}}{1-\beta_1^t}$$
$$v_j^t = \frac{\beta_2 v_j^{t-1} + (1-\beta_2)\, g_{tj}^2}{1-\beta_2^t}$$
$$w_j^t = w_j^{t-1} - \frac{\eta_t}{\sqrt{v_j^t + \epsilon}} m_j^t$$

This method is called **Adam** and combines both momentum methods and adaptive learning rate methods. And in practice, it achieves better convergence and faster convergence.

The benefit of RMSprop and Adam lies of the fact that they both use local information from the dataset to adapt the learning process to the task at hand. The former, uses adaptive learning rate, providing better optimization for model parameters, while the latter takes advantage also of the momentum method, allowing us to work with some complex functions with complex level sets.

## 1.2 Long-Short Term Network (LSTM) in PyTorch

### Question 1.5

As we saw, when we do a back propagation through a simple recurrent layer, the gradients needs to go through non-linearity and for multiplication by their current weight matrix. Both of these things can

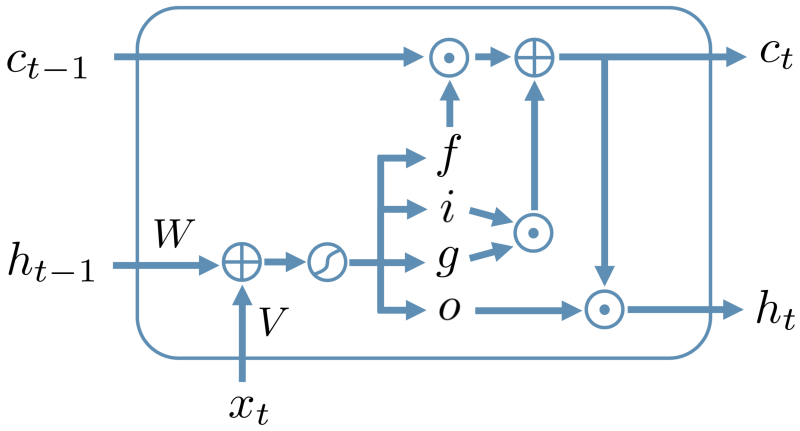



Figure 5: LSTM cell

cause the vanishing gradient problem. The main idea here is to create a 'shorter' way for them by adding a new separately way through recurrent layer. Thus, every layer would have its own internal memory $c$, which other layers on the network don't have access to. As a result, we will have now two ways through such layer; one between hidden units $h_{t-1}$ and $h_t$ and the second one between memory cells $c_{t-1}$ and $c_t$. The above describes a **LSTM** cell, which essentially provides a more effective primitive function to compute hidden units. The LSTM does have the ability to remove or add information to the cell state, carefully regulated by structures called gates. Gates are a way to optionally let information through. They are composed out of a sigmoid neural net layer and a pointwise multiplication operation. An LSTM has four of these gates, to protect and control the cell state.

**a) Brief explanation of LSTM gates**

**Forget Gate** A forget gate is responsible for removing information from the previous time step cell state $\mathbf{c}^{(t-1)}$ that is of less importance for the current or future time steps, and it is required for optimizing the performance of the LSTM network. This layer looks at the previous hidden state $\mathbf{h}(t-1)$ and the current input $\mathbf{x}(t)$, and via the **sigmoid** activation function, outputs values ranging from 0 to 1, deciding which values to keep and which to discard. If a 0 is output for a particular value in the cell state, it means that the forget gate wants the cell state to forget that piece of information completely. Similarly, a 1 means that the forget gate wants to remember that entire piece of information.

**Input Gate** The input gate is responsible for the addition of information to the cell state and it basically controls what to store in the memory. Similarly with the forget gate, to compute it we use a non linearity (sigmoid) over the linear combination of the previous hidden state $\mathbf{h}^{(t-1)}$ and the current input $\mathbf{x}^{(t)}$. The output that ranging from '0' to '1', now indicates at what extent to keep a piece of information, from redundant to very important respectively.

**Input Modulation Gate** To ensure that only that information is added to the cell state that is important and is not redundant, this gate is cross-multiplied with the output of the input gate, and regulates the amount of each peace of information contributes to the cell state. It is computed similarly to the input gate, but now the tanh activation function is used, which preserves the centering of the data and the scale of the output.

* At this point, using the above gates, we can update the cell state by forgetting from the old one ($\mathbf{c}^{(t-1)}\mathbf{f}^{(t)}$) and learn for the current input ($\mathbf{i}^{(t)}g^{(t)}$).

**Output Gate** Because not all information that runs along the cell state is fit for being output at a certain time, the output gate controls what to read from the memory and return to the outer world.

After making a filter using, again, just like the input gate, the values of $h_t - 1$ and $x_t$ with a sigmoid function on top of them, it can regulate the values that need to be output by multiplying it with the cell state. The tanh function was applied on the latter one, so that the values to be scaled to the range -1 to +1.

**b) Formula for the total number of trainable parameters in the LSTM cell.**

In the LSTM cell, there are

- Four wight matrices of size *input* $\times$ *hidden*. Thus $4 \cdot d \cdot n$.
- Four wight matrices of size *hidden* $\times$ *hidden*. Thus $4 \cdot n \cdot n$.
- Four biases, each for every gate, vectors of size *hidden* $\times$ *1*. Thus $4 \cdot n$.

Putting them all together, the total number of parameters would be

$$4 \cdot d \cdot n + 4 \cdot n \cdot n + 4 \cdot n = 4n(d + n + 1)$$

**Question 1.6**

We return to our problem to predict the last digit of a palindrome sequence, but this time we tackle it via LSTM network. The results, presented on figures 7 and 6, show clearly improvements comparing to those with RNN. The network seems capable of learning long-term dependencies, as it can achieve perfect accuracy even with a sequence of 160, where we remind that the RNN did not went above 0.2.

The specifications of the LSTM network that was used are:
**Learning rate:** `0.03`, **Batch size:** `128`, **Number of hidden units:** `128` and **Length of an input sequence:** `20`.

* Implemented on `part1/lstm.py`.

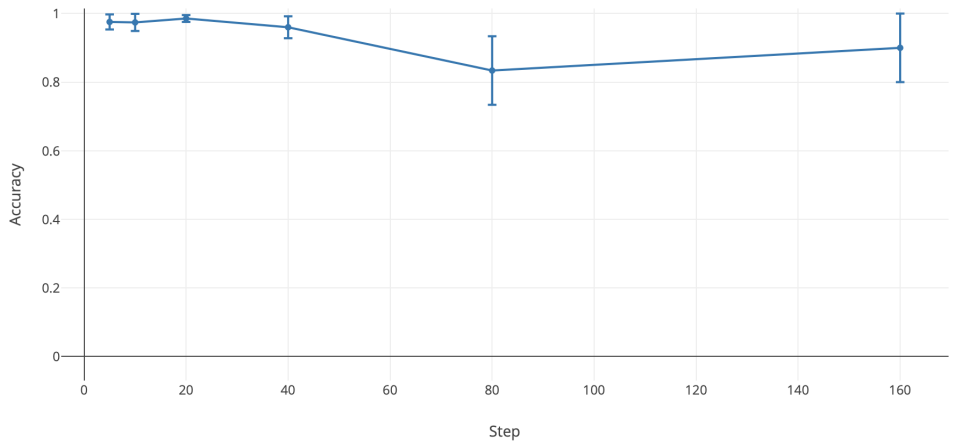
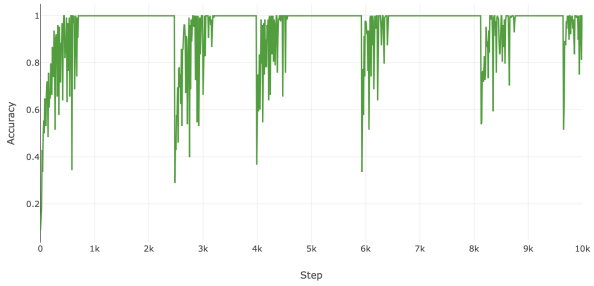
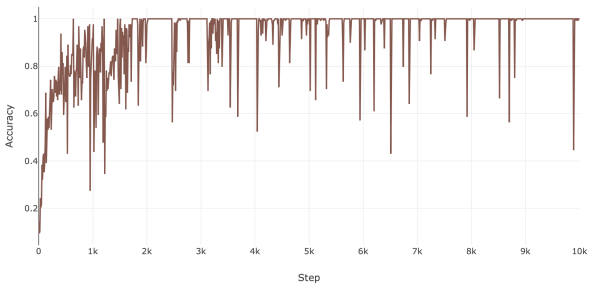



Figure 6: Average LSTM accuracies with respect to sequence length T over multiple runs. The error bars indicates the std of our averages.

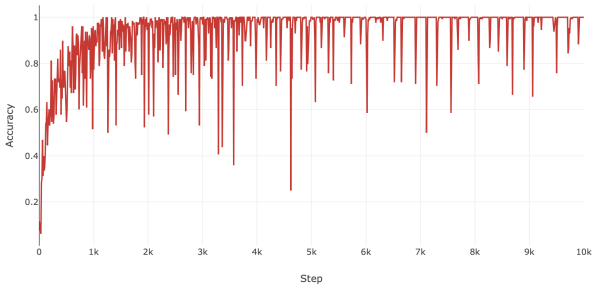



(a) Accuracy performance of `LSTM` model with T=5

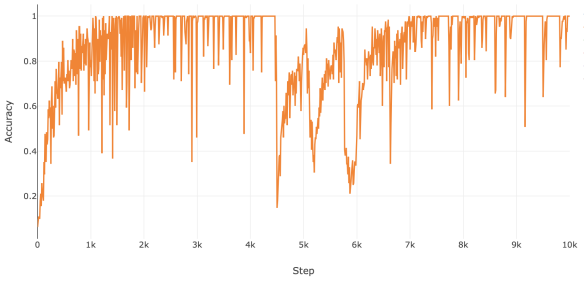



(b) Accuracy performance of `LSTM` model with T=10

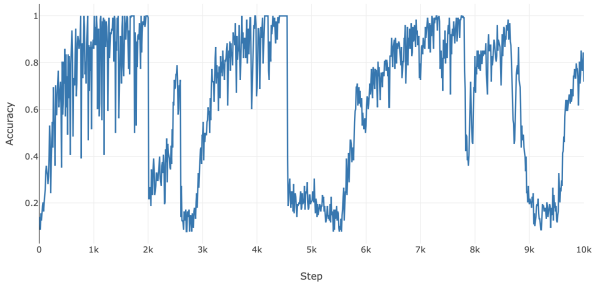
95



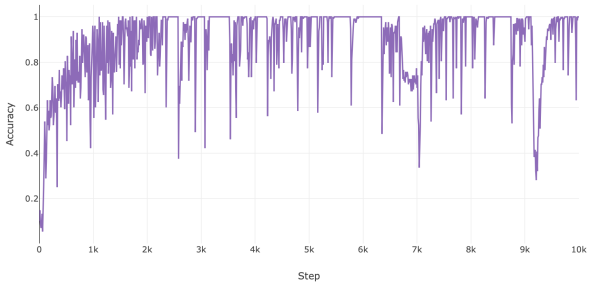

(c) Accuracy performance of `LSTM` model with T=20




(d) Accuracy performance of `LSTM` model with T=40




(e) Accuracy performance of `LSTM` model with T=80




(f) Accuracy performance of `LSTM` model with T=160

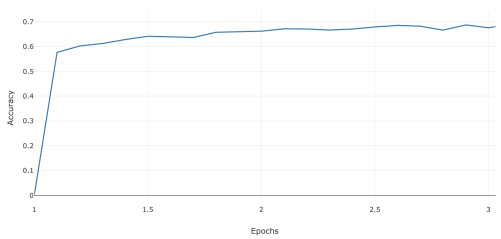
Figure 7: Performance of vanilla `LSTM` model with various lengths T.

# 2 Recurrent Nets as Generative Model

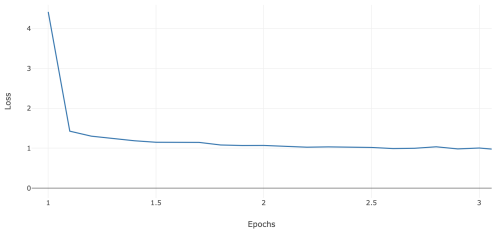
### Question 2.1

**a)** We implemented a `LSTM` neural network model in order to be trained on a text, like a book, and then, given a character to generate new text. We will use Leo Tolstoys' *Anna Karenina* book. Our network consisted by 2 with 512 hidden units each LSTMs layers, using a batch size of 128, learning rate 0.003 and dropout of 0.5. The accuracy and the loss are illustrated on figure 8.

* Implemented on `part2/model.py` and `part2/train.py`.




(a) Accuracy performance of `lstm` model




(b) Loss performance of `lstm` model

Figure 8: Performance of `lstm` neural network with PyTorch implementation on book 'Anna'.

**(b) & (c)** At this stage, we generate new sentences of length 30 by randomly setting the first character of the sentence. To achieve this, we used **four** different sampling methods; **greedy**, selecting always the top predicted character and, in order to give some element of randomness, with **temperature** sampling with values $[0.5, 1, 2]$. At table 1 we report 5 text samples generated by the network over evenly spaced intervals during training.

| | **0.5** | **1** | **2** | **greedy** |
|---|---|---|---|---|
| **(1, 500)** | *he wit hist and rhe enond her* | *owes,omcas hat masering" cuwy* | *Apry-2lymhaliv-.c-Am&Cleaics-* | *the tore the tore the tore th* |
| **(1, 3500)** | *Anna could not help saying th* | *eaker'ves and did not find tho* | *ed ?.1, '40 quiver'6lveoe) is* | *etersburg and the conversation* |
| **(1, 6000)** | *e, and he could not have been* | *uite unjust as the prayer were* | *wros all Ctthaolsdvt out after* | *said the princess with a smil* |
| **(2, 3000)** | *The princess went out and p* | *"Mercy on miles; without himse* | *; but Sergey Alexandrovna If .* | *ou say that I shall be delight* |
| **(2, 7000)** | *n the fact that the worst of f* | *uth," said Vronsky, taking the* | *-one?" Dolly, drationsky drew* | *the princess was sitting in the* |

Table 1: Sampling Methods. The (∗,∗) represents (epoch, step).

Overall, we can tell that the model by the first 500 steps have learn to distinguish words with spacing. Although it has the right intuition about the length of every word, most of them consist of random characters. By the 3500 step, the models' spelling and syntax is almost accurate. Special characters like `' , '` and `' " '` appear and used efficiently after the first epoch. Finally, on the last steps the model can now is capable of including main characters but also conversations between them, copping the style of the original book.

As for the different sampling methods, the **temperature** sampling with $\beta = 0.5$ seem to be the best, as it it has a good balance of greedy and completely random sampling. For the former, we can see that its sampling process sometimes can lead into repetitions of a word, or always include the most probable words into its text, like the word *princess*, because they appear the most into the original text. With $\beta = 2$, the model performs the worst, but as the number of the steps grow, the sampling processes becomes better. Finally, $\beta = 1$ is in close tie with $\beta = 0.5$ as it provides a good structured text, but with some random characters between the words.

**Bonus Question 2.2** In this final section we got more serious in discovering the capabilities and flaws of our model. Again, during training, we gave the model a small sentence in order to continue it by generating words. Initially, we perform three experiments; we gave (i) a main character of the book (2), (ii) a sentence with verb (3) and finally combination of these two (4). At this time, we used two sampling methods; the $\beta = 0.5$ from before, and another one, called **top_k**. In the latter case, we sampled from the $k$ more probable characters using a power distribution. In out implementation we used the $k = 5$. Looking at the results, the model seems to perform pretty good in both cases. The model seem to be more comfortable when we use a main hero of the book, as it has saw this patter on words more frequently. Using a random verb, like *"went"*, gives the model a hard time in the beginning. However, the results from the final sampling, where the model spend more time training, are appear to be very good, and it can combine the verb with various of follow up words, like `went "back"`, `went "into"` and `went "away"`. For the final, and probably the most difficult one, where the model get a name of a hero and it's condition (*"Anna said that"*, its performance it is not as good as in the previous two cases. The model sometimes struggling to connect it with something relevant in terms of context. However, after some steps, because we include the word *"said"*, the model in some cases combines it with a conversation, identified with the symbol `' " '`. We believe that a better hyperparameter tuning and more training may resolve this issue, or, more realistically speaking, reduce it.

| | top_k | beta |
|---|---|---|
| **(1, 500)** | *Stepan Arkadyevitchest, sole he the wise tottongering ander has tontere teet his at hesented and to hame his sime wet hi* | *Stepan Arkadyevitchang beong the onding ton, tele ind bet the sothe sore the sorett tait the,at onde soul the has ant s* |
| **(1, 1000)** | *Stepan Arkadyevitch, and how all the morrow of the stifute of myself. "Oh," he said, who had been so sent the peasant,* | *Stepan Arkadyevitch to me that it was life into the baby since the stronger shind in his prefence was been so you have b* |
| **(2, 0)** | *Stepan Arkadyevitch, who had seeing the words of the sofa, and the work holding her hand, and was expected, assuring him* | *Stepan Arkadyevitch said to her in all her heart as he had come for the first place, with a man who was already left by* |
| **(2, 6500)** | *Stepan Arkadyevitch was seeking for the sake on this argument, and he stood still and galognez feeling with them. How sh* | *Stepan Arkadyevitch, who has been so many things and an idea that he had not been through that had been so long ago he w* |
| **(2, 7000)** | *Stepan Arkadyevitch went out onto the further day.\n "Well, here you are a matter of family one could be so that he could* | *Stepan Arkadyevitch smiled at them.\n "If you are sure to make you my, but it's always better to go..." "Yes, I have nev* |

Table 2: Generated text given the sentence *Stepan Arkadyevitch* with `top_k` and `beta` methods. The (*,*) represents (epoch, step).

| | top_k | beta |
|---|---|---|
| **(1, 500)** | *He went hos old wash therithe here hing of tild and, thined, hit ine olladgerid,. tottinging and teand here* | *He went herend thet tithang to his he sidid, ander, ind the lit and rored ander toectoud hat fett and and li* |
| **(1, 1000)** | *He went to the matters of the simply shresting to me out to him. "It was then the morning simply then what* | *He went on the fingers with she had been she was in the string back and coming marriage. The standing of her* |
| **(1, 3500)** | *He went away. The cause of the while, and to say that he was almost this, he thought to her she had taken up* | *He went out of the district, and the question of the hat and was in his presence of his wife and his study a* |
| **(2, 6500)** | *He went into his friends, and had been annoyed. "Why are you, I'm a happy month 1.E.3.,*1.E.7.* Don't you* | *He went up to his mother, and the new drawing room was already to call in the colonel he was to blame for he* |
| **(2, 7000)** | *He went back to her hand, some bag to the first thing that summer with his wife, but the servants, who would* | *He went into the hall with the stream and sending the peasants and the smart guards and his throat from the* |

Table 3: Generated text given the sentence *He went* with `top_k` and `beta` methods. The (*,*) represents (epoch, step).

| | top_k | beta |
|---|---|---|
| **(1, 500)** | *Anna said thathatheren sot sot toris tere and to cering the wamit athe hit thang ole to the totting, thanges tiseri* | *Anna said thatce ane and hereedy and the fale and tore the hathang tat the cather anle the peren hery tote thit per* |
| **(1, 1000)** | *Anna said that she was a sechicated the man on a pince, said their play alone, whole, had an evought in the coldran* | *Anna said that his eyes of the same and sitting the baby and was all the children when he was simply one of the ans* |
| **(1, 5500)** | *Anna said that he hid his spend the downchores of the peasant with a smile, smiling.,"You don't know that he comes* | *Anna said that he was distressed, and the carriages the dining room the sick man with a smile of the room. \n "No, I* |
| **(2, 4000)** | *Anna said that the princess's satisfaction that she was not merely feeling a long and too, to tell you about her fr* | *Anna said that he could not leave him. But he was not a pity she had not been married so, he had to go on like the* |
| **(2, 5000)** | *Anna said that he had always been said again with wides. He was seeing her soul shouldered gaises of manuring her.* | *Anna said that he could not have been at the rest of the property. He was standing up with a smile.,"I was a new m* |

Table 4: Generated text given the sentence *Anna said that* with `top_k` and `beta` methods. The (*,*) represents (epoch, step).

We finally present a fully generated text, starting with the word `Anna`, of 2000 words. Initially, the model uses commas more that is necessary, but this is resolved as it generates more characters. Despite the fact that the overall structure of the text is impressive (grammatically, vocabulary, creates new line when a conversation starts), unfortunately there is no clear cohesion and meaning in the story. This could be improved by using a different sequence length but also a different network architecture, in order to achieve better accuracy in predicting the next character. In general the first bunch of characters will be a little rough since it hasn't built up a long history of characters to predict from.

```
Anna, and without waiting for the simple-tentific right, but at all she was to have been
sitting in her hand, went out of the questions of the public bog of her handsome, recognized
him with a shoulder, which he shruggled her face will strange all the dreadful, soft for
as they had several times were struggling with her, so terrible and well-finisy, and the
whole family, and he went up to him by his self-possession of the house, as an inner voice.
"Yes, I don't know."
"Why shouldn't I true!"
"Oh, no!" answered Vronsky's prayer, was a strangen in which her even through all her elder
stood the lower and light and sorts of looked suddenly the same departure who had been
sending on the scenes to the house and triging with his face, shining at her.  "We've no
need to blame!  There's not only fond of that?"
"No, I'm so happy!...  Oh, it was out of keeping with them than your life, and see, as
you think."
"Yes, yes," said Alexey Alexandrovitch to her head, the low the beginning of their break
now that he had been said that it were to confonce, and always forgetfulness, though sufficient
is awful!" said Vronsky, looking intently at her.  He does not know.  I have no room, the
considerations, which has gone."
"And the sice man, here it means in the ship," said Vronsky, ledings of this sight.
Alexey Alexandrovitch self-conversation was straightfing on his heart.  He started a longing
to see you.  I won't and I believe, I should be independently as a matter on the committee,
and he felt that the chalk was still, and the counter advantages in all its of all.  And
would not be done with my between you and say about me, but how?  What?" said Laska, rose
significantly, with a shampfropic, started and washed, always agreed to herself that he
had not yet seen Alexey Alexandrovitch said that she could not but attempting the platform
as he did not know what to do.  He has been laying a little!  To be presented to you, I
shall say, I admit the
```

# 3 Graph Neural Networks

Machine learning on graphs is an important and ubiquitous task with applications ranging from drug design to friendship recommendation in social networks. The primary challenge in this domain is finding a way to represent, or **encode**, graph structure so that it can be easily exploited by machine learning models. In other words, the goal is to encode nodes so that similarity in the embedding space (e.g., dot product) approximates similarity in the original network.

$$\text{similarity}(u, v) \approx \boldsymbol{z}_u^T \boldsymbol{z}_v$$

Thus, the definition of the **encoder**, that maps each node to a low-dimensional vector, is needed. To resolve this, a key idea is to generate node embeddings based on local neighborhoods, known as **neighborhood aggregation**. The intuition behind it, is that nodes aggregate information from their neighbors using neural networks, allowing for parameter sharing in the encoder and inductive learning.

Key distinctions between all the various methods that have been proposed lies in how different approaches aggregate messages. A basic approach, described at [2], is to average neighbor information and apply a neural network. However, instead of aggregating information from neighbors, the intuition behind **Graph Neural Networks** GNNs [3] is that graphs can be viewed as specifying scaffolding for a message passing algorithm between nodes.

## 3.1 GCN Forward Layer

### Question 3.1

**a)** The propagation rule for a layer in the GCNs architecture is given by the following equation

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

Exploiting the matrix product $\hat{A}H^{(l)}$, and defying the $N(v)$ as the neighbors of node $v$ (there is an edge between them), we end up with

$$\left(\hat{A}H^{(l)}\right)_{uv} = \sum_k \hat{A}_{uk} H_{kv}^{(l)} = \sum_{k \in N(u) \cup u} H_{kv}^{(l)}$$

Thus, the hidden embedding $H^{(l+1)}$ mixes all the neighbor nodes of $u$ (including itself).

In general, the GCNs neighborhood aggregation is described by

$$\mathbf{h}_v^k = \sigma\left(\mathbf{W}_k \sum_{u \in N(v) \cup v} \frac{\mathbf{h}_u^{k-1}}{\sqrt{|N(u)||N(v)|}}\right)$$

For the above we can tell that the same matrix $W_k$ is used for self and neighbor embeddings and instead of simple average, the normalization varies across neighbors. This configuration results into more parameter sharing and down-weight high degree neighbors.

**b)** To propagate information of nodes 3 hops away in the graph, we would need at most three layers. This can be proven by the induction method; For l = 1, information propagates of nodes that are one step away. Then, at step l=2, the nodes that were originally two steps away (at l=1), now they will be one, and can be considered as neighbors. Thus, by the step l=3, the information of a node will be propagated to nodes that are three hops away.

## 3.2 Applications of GNNs

### Question 3.2

Many machine learning applications seek to make predictions or discover new patterns using graph-structured data as feature information. For example, one might wish to classify the role of a protein in a biological interaction graph [6], predict the role of a person in a collaboration network , recommend new friends to a user in a social network [7], or predict new therapeutic applications of existing drug molecules, whose structure can be represented as a graph [8].

## 3.3 Comparing and Combining GNNs and RNNs

### Question 3.3

**a)** As mentioned before, RNN units are modified to accommodate both the node representations as well as their neighborhood information. Thus, dataset with temporal dependencies, such as financial forecasting and text processing, they are the ideal network architecture to exploit and process the information of the data efficiently. Moreover, even though in these tasks we care about long dependencies, the really long ones are actually unnecessary and unwanted. However, in some cases, like patient's health history, or even better, a graph-tree representation of all his family, we want to keep the entire history in order to successfully track down hereditary diseases. For this matter GNNs are preferred, as we make sure to have all the data we want and also their hole purpose is to exploit information from graphs effectively. Furthermore, data like a social network, does not have any clear order in terms of its nodes (they are more like categorical variables). Thus, again this time GNNs would perform better than the RNNs.

**b)** As implemented in [10] and [11], RNNs can be used as an alternative solution to encode graphs into low-dimensional vectors in order to be successfully exploited by machine learning algorithms. In contrast with the GCN framework, where the feature vectors of the neighbors it is a weighted sum followed by a non-linearity, in [11] we pass the neighbors though a modified LSTM and try to predict, for example graph labels, if it is a classification problem. The general pipeline is illustrated on figure 9.

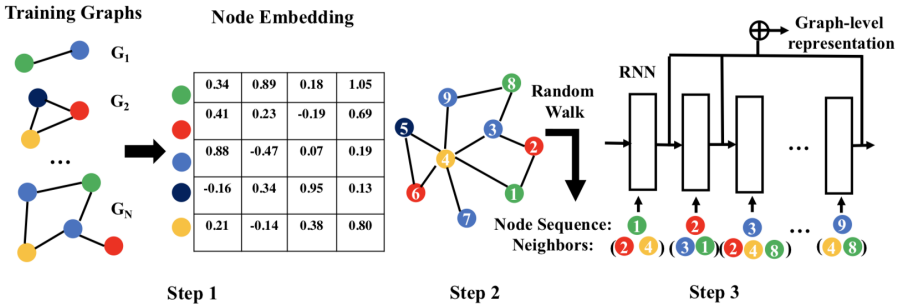



Figure 9: GCN - LSTM pipeline

# 4 Conclusion

In this report, we explore the neural networks that are preferable when we deal with sequential data, the RNNs. We illustrated two popular implementations of the latter, the Vanilla RNNs and the LSTMs and we showed the importance of the gated architecture of the latter's in learning long-term relationships. The former's (possible) reasons of failures where explored (vanishing and exploding gradient), while providing motivations on the importance of momentum and adaptive learning rate when dealing with complex data. Moreover, we exploit the power of LSTMs, as after we trained it on a book, we could generate sentences and form hole paragraphs. Even though in term of the context, the generated text was well formalized, correctly grammatically and provided a nice writing flow. In order to take full advantages of the graph structured data, GNNs where proposed, which we studied here, and particularly the CGN. Finally, we draw some conclusions between the GNNs and RNNs, on how they are different but also how they could be combined.

# Bibliography

[1] Part 2: Graph Neural Networks (link).

[2] Representation Learning on Graphs: Methods and Applications (link).

[3] SEMI-SUPERVISED CLASSIFICATION WITH GRAPH CONVOLUTIONAL NETWORKS, Kipf & Welling (link).

[4] A Comprehensive Survey on Graph Neural Networks (link).

[5] Graph Neural Networks: A Review of Methods and Applications (link).

[6] Alex Fout, Jonathon Byrd, Basir Shariat, and Asa Ben-Hur. Protein interface prediction using graph convolutional networks (link).

[7] SocialGCN: An Efficient Graph Convolutional Network based Model for Social Recommendation (link).

[8] Deeply learning molecular structure-property relation- ships using attention- and gate-augmented graph con- volutional network (link).

[9] GraphRNN: Generating Realistic Graphs with Deep Auto-regressive Models (link).

[10] GATED GRAPH SEQUENCE NEURAL NETWORKS (link).

[11] Learning Graph-Level Representations with Recurrent Neural Networks (link).