Deep Learning Assignment 2

Elias Kassapis - 12409782

1 Vanilla RNN versus LSTM

1.1 Toy Problem: Palindrome Numbers

1.2 Vanilla RNN in PyTorch

Question 1.1

We have the following equations

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

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

were $\mathbf{h}^{(t)}$ and $\mathbf{x}^{(t)}$ are the hidden state, and the input vector at timestep $t = 1, ..., T$. $\mathbf{W}_{hx}$ denotes the input-to-hidden weight matrix, $\mathbf{W}_{hh}$ is the hidden-to-hidden 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. Also

$$\hat{\mathbf{y}}^{(t)} = \text{softmax}\left(\mathbf{p}^{(t)}\right) \tag{3}$$

$$\mathcal{L}^{(T)} = -\sum_{k=1}^{K} \mathbf{y}_k \log \hat{\mathbf{y}}_k^{(T)} \tag{4}$$

were $\hat{\mathbf{y}}^{(t)}$ is the output at timestep t, $\mathcal{L}^{(T)} =$ is the cross-entropy loss computed at the last timestep, $k = 1, ..., K$ is the class, and $\mathbf{y}$ is the denotes a one-hot vector of length K, containing true labels.

- Expanding $\frac{\partial \mathcal{L}^{(T)}}{\partial \mathbf{W}_{ph}}$ using the chain rule gives

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

Now, the first term is given by

$$\begin{aligned}\frac{\partial \mathcal{L}^{(T)}}{\partial \hat{\mathbf{y}}^{(T)}} &= \frac{\partial}{\partial \hat{\mathbf{y}}^{(T)}} \left(-\sum_{k=1}^{K} \mathbf{y}_k \log \hat{\mathbf{y}}_k^{(T)}\right) \\ &= \begin{bmatrix} -\frac{y_1}{\hat{y}_1^{(T)}} & \cdots & -\frac{y_K}{\hat{y}_K^{(T)}} \end{bmatrix} \in \mathbb{R}^{1\times K}\end{aligned}$$

The second term is given by

$$
\begin{aligned}
\frac{\partial \hat{\mathbf{y}}^{(T)}}{\partial \mathbf{p}^{(T)}} &= \frac{\partial}{\partial \mathbf{p}^{(T)}} \left(\mathrm{softmax}\left(\mathbf{p}^{(t)}\right)\right) \\
&= \begin{bmatrix}
\hat{y}_1^{(T)}\left(1-\hat{y}_1^{(T)}\right) & -\hat{y}_1^{(T)}\hat{y}_2^{(T)} & \dots & -\hat{y}_1^{(T)}\hat{y}_K^{(T)} \\
-\hat{y}_2^{(T)}\hat{y}_1^{(T)} & \hat{y}_2^{(T)}\left(1-\hat{y}_2^{(T)}\right) & \dots & -\hat{y}_2^{(T)}\hat{y}_K^{(T)} \\
\vdots & \ddots & \ddots & \vdots \\
-\hat{y}_K^{(T)}\hat{y}_1^{(T)} & \dots & \dots & \hat{y}_K^{(T)}\left(1-\hat{y}_K^{(T)}\right)
\end{bmatrix} \in \mathbb{R}^{K\times K}
\end{aligned}
$$

And the third term is given by

$$
\begin{aligned}
\frac{\partial \mathbf{p}^{(T)}}{\partial \mathbf{W}_{ph}} &= \frac{\partial}{\partial \mathbf{W}_{ph}} \left(\mathbf{W}_{ph}\mathbf{h}^{(T)} + \mathbf{b}_p\right) \\
&= \left[\begin{bmatrix}
\frac{\partial p_1}{\partial W_{1,1}} & \cdots & \frac{\partial p_1}{\partial W_{1,1}} \\
\vdots & \ddots & \vdots \\
\frac{\partial p_1}{\partial W_{1,d_h}} & \cdots & \frac{\partial p_1}{\partial W_{1,d_h}}
\end{bmatrix} \cdots \begin{bmatrix}
\frac{\partial p_K}{\partial W_{K,1}} & \cdots & \frac{\partial p_K}{\partial W_{K,1}} \\
\vdots & \ddots & \vdots \\
\frac{\partial p_K}{\partial W_{K,d_h}} & \cdots & \frac{\partial p_K}{\partial W_{K,d_h}}
\end{bmatrix}\right] \\
&= \left[\begin{bmatrix}
h_1^{(T)} & 0 & \cdots & 0 \\
\vdots & \vdots & \ddots & \vdots \\
h_1^{(T)} & 0 & \cdots & 0
\end{bmatrix} \cdots \begin{bmatrix}
0 & \cdots & 0 & h_{d_h}^{(T)} \\
\vdots & \ddots & \vdots & \vdots \\
0 & \cdots & 0 & h_{d_h}^{(T)}
\end{bmatrix}\right] \in \mathbb{R}^{K\times K\times d_h}
\end{aligned}
$$

were d_h is the dimentionality of $\mathbf{h}^{(T)}$, and for the purpose of illustrative convenience, we have chosen $K = d_h$ (this is not the case).

Therefore, plugging the analytical expansion of each of the three terms in $\frac{\partial \mathcal{L}^{(T)}}{\partial \mathbf{W}_{ph}}$, we get

$$
\begin{aligned}
\frac{\partial \mathcal{L}^{(T)}}{\partial \mathbf{W}_{ph}} =& \frac{\partial \mathcal{L}^{(T)}}{\partial \hat{\mathbf{y}}^{(T)}} \frac{\partial \hat{\mathbf{y}}^{(T)}}{\partial \mathbf{p}^{(T)}} \frac{\partial \mathbf{p}^{(T)}}{\partial \mathbf{W}_{ph}} \\
=& \begin{bmatrix} -\frac{y_1}{\hat{y}_1^{(T)}} & \cdots & -\frac{y_K}{\hat{y}_K^{(T)}} \end{bmatrix}
\begin{bmatrix}
\hat{y}_1^{(T)}\left(1-\hat{y}_1^{(T)}\right) & -\hat{y}_1^{(T)}\hat{y}_2^{(T)} & \dots & -\hat{y}_1^{(T)}\hat{y}_K^{(T)} \\
-\hat{y}_2^{(T)}\hat{y}_1^{(T)} & \hat{y}_2^{(T)}\left(1-\hat{y}_2^{(T)}\right) & \dots & -\hat{y}_2^{(T)}\hat{y}_K^{(T)} \\
\vdots & \ddots & \ddots & \vdots \\
-\hat{y}_K^{(T)}\hat{y}_1^{(T)} & \dots & \dots & \hat{y}_K^{(T)}\left(1-\hat{y}_K^{(T)}\right)
\end{bmatrix} \\
& \left[\begin{bmatrix}
h_1^{(T)} & 0 & \cdots & 0 \\
\vdots & \vdots & \ddots & \vdots \\
h_1^{(T)} & 0 & \cdots & 0
\end{bmatrix} \cdots \begin{bmatrix}
0 & \cdots & 0 & h_{d_h}^{(T)} \\
\vdots & \ddots & \vdots & \vdots \\
0 & \cdots & 0 & h_{d_h}^{(T)}
\end{bmatrix}\right] \\
=& \begin{bmatrix} y_1 \sum_{i=1}^{K} \hat{y}_i & \cdots & y_1 \sum_{i=1}^{K} \hat{y}_i \end{bmatrix}
\left[\begin{bmatrix}
h_1^{(T)} & 0 & \cdots & 0 \\
\vdots & \vdots & \ddots & \vdots \\
h_1^{(T)} & 0 & \cdots & 0
\end{bmatrix} \cdots \begin{bmatrix}
0 & \cdots & 0 & h_{d_h}^{(T)} \\
\vdots & \ddots & \vdots & \vdots \\
0 & \cdots & 0 & h_{d_h}^{(T)}
\end{bmatrix}\right]
\end{aligned}
$$

were

$$\frac{\partial \mathcal{L}^{(T)}}{\partial \mathbf{W}_{hh}} \in \mathbb{R}^{K \times d_h}$$

- Expanding $\frac{\partial \mathcal{L}^{(T)}}{\partial \mathbf{W}_{hh}}$ using the chain rule gives

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

We have already computed the analytical expansion of the first two terms. Now the third term is given by

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

So far we have

$$\begin{aligned} \frac{\partial \mathcal{L}^{(T)}}{\partial \mathbf{W}_{hh}} &= \frac{\partial \mathcal{L}^{(T)}}{\partial \hat{\mathbf{y}}^{(T)}} \frac{\partial \hat{\mathbf{y}}^{(T)}}{\partial \mathbf{p}^{(T)}} \frac{\partial \mathbf{p}^{(T)}}{\partial \mathbf{h}^{(T)}} \frac{\partial \mathbf{h}^{(T)}}{\partial \mathbf{W}_{hh}} \\ &= \begin{bmatrix} -\frac{y_1}{\hat{y}_1^{(T)}} & \cdots & -\frac{y_K}{\hat{y}_K^{(T)}} \end{bmatrix} \begin{bmatrix} \hat{y}_1^{(T)} \left(1 - \hat{y}_1^{(T)}\right) & -\hat{y}_1^{(T)} \hat{y}_2^{(T)} & \cdots & -\hat{y}_1^{(T)} \hat{y}_K^{(T)} \\ -\hat{y}_2^{(T)} \hat{y}_1^{(T)} & \hat{y}_2^{(T)} \left(1 - \hat{y}_2^{(T)}\right) & \cdots & -\hat{y}_2^{(T)} \hat{y}_K^{(T)} \\ \vdots & \ddots & \ddots & \vdots \\ -\hat{y}_K^{(T)} \hat{y}_1^{(T)} & \cdots & \cdots & \hat{y}_K^{(T)} \left(1 - \hat{y}_K^{(T)}\right) \end{bmatrix} \mathbf{W}_{ph} \left(\frac{\partial \mathbf{h}^{(T)}}{\partial \mathbf{W}_{hh}} \right) \end{aligned}$$

Now, the fourth term is given by

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

We have

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

As we can see above, $\frac{\partial \mathbf{h}^{(T)}}{\partial \mathbf{W}_{hh}}$ depends on $\mathbf{h}^{(T-1)}$, and $\mathbf{h}^{(T-1)}$ depends on $\mathbf{h}^{(T-2)}$, and so on and on. Therefore, the dependency of the hidden state at each timestep to the hidden state at the previous timestep is recurrently propagated all the way back to the start of the sequence $\left(\mathbf{h}^{(T)} \rightarrow \mathbf{h}^{(T-1)} \rightarrow ... \rightarrow \mathbf{h}^{(0)}\right)$. Therefore, $\dfrac{\partial \mathbf{h}^{(T)}}{\partial \mathbf{W}_{hh}}$ itself is subject to the chain rule as follows

$$\frac{\partial \mathbf{h}^{(T)}}{\partial \mathbf{W}_{hh}} = \frac{\partial \mathbf{h}^{(T)}}{\partial \mathbf{h}^{(T-1)}} \frac{\partial \mathbf{h}^{(T-1)}}{\partial \mathbf{h}^{(T-2)}} \cdots \frac{\partial \mathbf{h}^{(2)}}{\partial \mathbf{h}^{(1)}} \frac{\partial \mathbf{h}^{(1)}}{\partial \mathbf{W}_{hh}}$$

Now

$$\begin{aligned}
\frac{\partial}{\partial \mathbf{W}_{hh}} \left(\mathbf{W}_{hx}\mathbf{x}^{(1)} + \mathbf{W}_{hh}\mathbf{h}^{(0)} + \mathbf{b}_h\right) &= \frac{\partial}{\partial \mathbf{W}_{hh}} \left(\mathbf{W}_{hh}\mathbf{h}^{(0)}\right) \\
&= \begin{bmatrix} \begin{bmatrix} \frac{\partial h_1}{\partial W_{1,1}} & \cdots & \frac{\partial h_1}{\partial W_{1,1}} \\ \vdots & \ddots & \vdots \\ \frac{\partial h_1}{\partial W_{1,d_h}} & \cdots & \frac{\partial h_1}{\partial W_{1,d_h}} \end{bmatrix} & \cdots & \begin{bmatrix} \frac{\partial h_{d_h}}{\partial W_{d_h,1}} & \cdots & \frac{\partial h_{d_h}}{\partial W_{d_h,1}} \\ \vdots & \ddots & \vdots \\ \frac{\partial h_{d_h}}{\partial W_{d_h,d_h}} & \cdots & \frac{\partial h_{d_h}}{\partial W_{d_h,d_h}} \end{bmatrix} \end{bmatrix} \\
&= \begin{bmatrix} \begin{bmatrix} h_1^{(T-1)} & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ h_1^{(T-1)} & 0 & \cdots & 0 \end{bmatrix} & \cdots & \begin{bmatrix} 0 & \cdots & 0 & h_{d_h}^{(T-1)} \\ \vdots & \ddots & \vdots & \vdots \\ 0 & \cdots & 0 & h_{d_h}^{(T-1)} \end{bmatrix} \end{bmatrix} \in \mathbb{R}^{d_h \times d_h \times d_h}
\end{aligned}$$

Therefore $\dfrac{\partial \mathbf{h}^{(1)}}{\partial \mathbf{W}_{hh}}$ is given by

$$\frac{\partial \mathbf{h}^{(1)}}{\partial \mathbf{W}_{hh}} = diag\left(1 - \left(\mathbf{h}^{(1)}\right)^2\right) \begin{bmatrix} \begin{bmatrix} h_1^{(0)} & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ h_1^{(0)} & 0 & \cdots & 0 \end{bmatrix} & \cdots & \begin{bmatrix} 0 & \cdots & 0 & h_{d_h}^{(0)} \\ \vdots & \ddots & \vdots & \vdots \\ 0 & \cdots & 0 & h_{d_h}^{(0)} \end{bmatrix} \end{bmatrix}$$

were

$$\frac{\partial \mathbf{h}^{(1)}}{\partial \mathbf{W}_{hh}} \in \mathbb{R}^{d_h \times d_h \times d_h}$$

Also

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

Therefore, we get

$$\begin{aligned}
\frac{\partial \mathbf{h}^{(T)}}{\partial \mathbf{W}_{hh}} &= \frac{\partial \mathbf{h}^{(T)}}{\partial \mathbf{h}^{(T-1)}} \frac{\partial \mathbf{h}^{(T-1)}}{\partial \mathbf{h}^{(T-2)}} \cdots \frac{\partial \mathbf{h}^{(2)}}{\partial \mathbf{h}^{(1)}} \frac{\partial \mathbf{h}^{(1)}}{\partial \mathbf{W}_{hh}} \\
&= \left(\prod_{t=2}^{T} diag\left(1 - \left(\mathbf{h}^{(t)}\right)^2\right)\right) \left(\mathbf{W}_{hh}\right)^{T-1} diag\left(1 - \left(\mathbf{h}^{(1)}\right)^2\right) \begin{bmatrix} \begin{bmatrix} h_1^{(0)} & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ h_1^{(0)} & 0 & \cdots & 0 \end{bmatrix} & \cdots & \begin{bmatrix} 0 & \cdots & 0 & h_{d_h}^{(0)} \\ \vdots & \ddots & \vdots & \vdots \\ 0 & \cdots & 0 & h_{d_h}^{(0)} \end{bmatrix} \end{bmatrix}
\end{aligned}$$

Hence

$$
\begin{aligned}
\frac{\partial \mathcal{L}^{(T)}}{\partial \mathbf{W}_{hh}} =& \frac{\partial \mathcal{L}^{(T)}}{\partial \hat{\mathbf{y}}^{(T)}} \frac{\partial \hat{\mathbf{y}}^{(T)}}{\partial \mathbf{p}^{(T)}} \frac{\partial \mathbf{p}^{(T)}}{\partial \mathbf{h}^{(T)}} \frac{\partial \mathbf{h}^{(T)}}{\partial \mathbf{W}_{hh}} \\
=& \begin{bmatrix} -\frac{y_1}{\hat{y}_1^{(T)}} & \cdots & -\frac{y_K}{\hat{y}_K^{(T)}} \end{bmatrix} \begin{bmatrix} \hat{y}_1^{(T)}\left(1-\hat{y}_1^{(T)}\right) & -\hat{y}_1^{(T)}\hat{y}_2^{(T)} & \cdots & -\hat{y}_1^{(T)}\hat{y}_K^{(T)} \\ -\hat{y}_2^{(T)}\hat{y}_1^{(T)} & \hat{y}_2^{(T)}\left(1-\hat{y}_2^{(T)}\right) & \cdots & -\hat{y}_2^{(T)}\hat{y}_K^{(T)} \\ \vdots & \ddots & \ddots & \vdots \\ -\hat{y}_K^{(T)}\hat{y}_1^{(T)} & \cdots & \cdots & \hat{y}_K^{(T)}\left(1-\hat{y}_K^{(T)}\right) \end{bmatrix} \mathbf{W}_{ph} \\
& \left(\prod_{t=2}^{T} diag\left(1-\left(\mathbf{h}^{(t)}\right)^2\right)\right) \left(\mathbf{W}_{hh}\right)^{T-1} diag\left(1-\left(\mathbf{h}^{(1)}\right)^2\right) \left[\begin{bmatrix} h_1^{(0)} & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ h_1^{(0)} & 0 & \cdots & 0 \end{bmatrix} \cdots \begin{bmatrix} 0 & \cdots & 0 & h_{d_h}^{(0)} \\ \vdots & \ddots & \vdots & \vdots \\ 0 & \cdots & 0 & h_{d_h}^{(0)} \end{bmatrix}\right]
\end{aligned}
$$

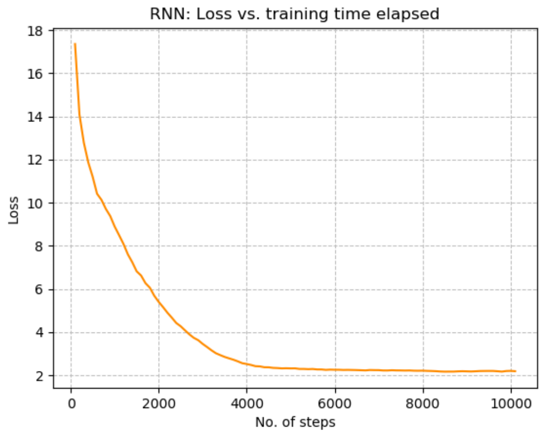
were

$$\frac{\partial \mathcal{L}^{(T)}}{\partial \mathbf{W}_{hh}} \in \mathbb{R}^{d_h \times d_h}$$

Regarding the temporal dependence of the gradients considered, as we can see from the derivations above, $\frac{\partial L^{(T)}}{\partial W_{ph}}$ only depended on variables in the current timestep T, however $\frac{\partial L^{(T)}}{\partial W_{hh}}$ also depends on all the previous hidden states. As we can see above, the analytical expansion of $\frac{\partial \mathcal{L}^{(T)}}{\partial \mathbf{W}_{hh}}$ contains the term $\left(\mathbf{W}_{hh}\right)^{T-1}$. If the values in $\mathbf{W}_{hh}$ are lower than 1, a resultant problem that can occur when training this recurrent network for a large number of timesteps is the so called "vanishing" gradient problem, were gradient in the early layers of the network becomes close to zero, slowing down or even completely halting the training process. Conversely, if the values in $\mathbf{W}_{hh}$ are greater than one, a large number of timesteps will result in the "exploding" gradient problem, were the gradient is amplified in each layer, so in the early layers of the network, huge gradients result in huge changes in parameter values after each parameter update. This results in perpetual oscillations, which does not allow the model to learn anything.

Question 1.2

For this section, I implemented the vanilla recurrent neural network as specified by the equations above, using the given default parameters, and using the RMSProp optimizer for tuning the weights, , and Cross-Entropy as my loss function. We use the function $torch.nn.utils.clip\_grad\_norm()$ after applying backward propagation to prevent exploding gradients by 'clipping' the norm of the gradients, to restrain the gradient values to a certain threshold. This essentially acts as a limit to the size of the updates of the parameters of every layer, ensuring that the parameter values don't change too much from their previous values. I also added the convergence condition $|\mathcal{L}^{(\tau+1)} - \mathcal{L}^{(\tau)}| > \epsilon$, were $\mathcal{L}$ is the training loss over the last 100 batches, τ is the current training step, and $\epsilon = 1e-6$. The loss and accuracy were computed by taking the average of the last 100 batches. The corresponding curves are illustrated below in Figure 1.



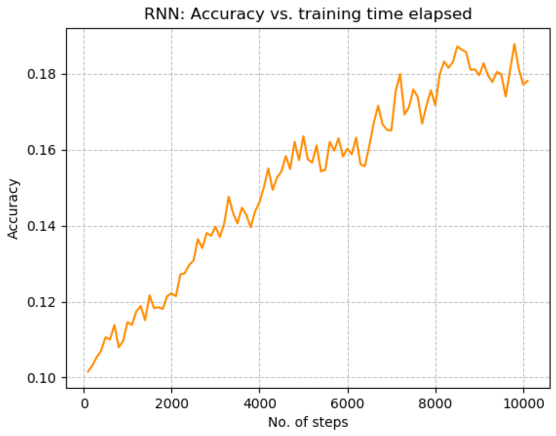
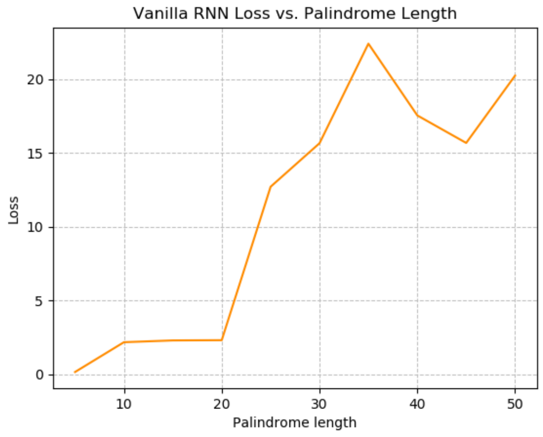


Figure 1: The Loss and Accuracy versus the training time elapsed for the Vanilla RNN, using the default parameters (sequence length = 10)

As we can see, using default parameters, the loss converges to 2.18 (2 d.p.), with a final accuracy of 0.18 (2 d.p.). As the training steps increase, we can see that the loss decreases and the accuracy increases, as the model makes less mistakes.

Question 1.3

To examine the memorization capability of the vanilla RNN, I started with training the net until corvergence on short palindromes of length $L = 5$, , and gradually increased L by a step of 5 untill we reached palindromes of length $L = 50$. The net was trained using the first $L - 1$ digits of each palindrome as input, and the task was to predict the last digit. Every other parameter of the model except the training rate was kept constant to assess the effect of changing only the sequence length on the performance of the model, The loss and accuracy for all sequence lengths tested are illustrated in Figure 2.



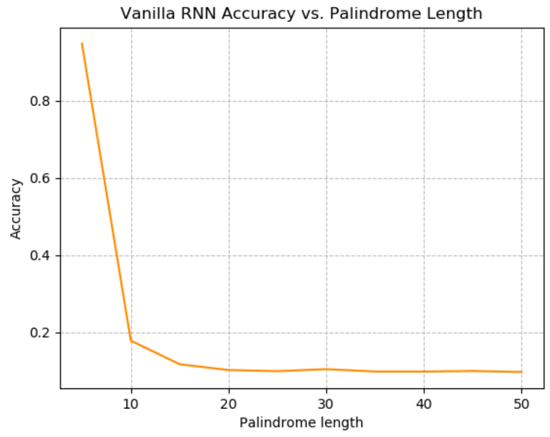


Figure 2: The Loss and Accuracy versus the palindrome length for the Vanilla RNN, using the default parameters

As expected, the loss increases, and the accuracy decreases as the sequence length increases, because of multiple multiplications of the same weight matrix (for each input), and because when long sequences are used, information from early inputs is lost as too much information has been aggregated after that without thorough modulation, a phenomenon known as ’forgetting’. We can see that after sequence lengths of size $L = 20$, the model is unable to learn anything.

Question 1.4

A common phenomenon when using neural network with deep, complex architectures is what is known as "pathological curvature" of the error topology. This is when the shape of the loss function we want to optimize is highly convoluted, making our task very difficult. Vanilla Stochastic Gradient Descent (SGD) does not perform well on such surfaces, as SGD updates in parameters in intricate areas of the error topology such as narrow ridges may cause our current error value to bounce across the parallel sides of the ridge instead of moving along a plane, reducing the rate at which we close in to the global optimum we are trying to reach, and even preventing us from reaching it, stopping our model from learning anything. This problem can be addressed by enhancing vanilla SGD with a technique called '**momentum**' or by using other SGD variants such as **RMSProp** and **Adam**, which prevent this oscillating behaviour from occurring, thereby improving learning. The intuition of these methods is described below.

• In the case of **momentum**, instead of using only the current step's gradient to guide the search, momentum also incorporates the decaying average of the gradients of the past steps, weighted by the 'coefficient of momentum' constant, to determine the direction of the update. The reason this is effective is because each update step is adapted to it's 'context', which we define as information retained from previous gradients, dampening sharp changes between the direction of the update of the current step, and that of previous steps. As a result, the oscillations in our search are suppressed, and we move faster towards the optima, building 'momentum'.

• The **RMSProp** optimizer, similarly to SGD with momentum, is also used to restrict oscillations. However, the exponentially decaying average of past squared gradients (which weights temporally closer gradients more than gradients further away) is retained, and incorporated directly into the learning rate. The intuition is prioritizing recent information to adapt the learning rate will implicitly dampen oscillations occuring in successive updates by restricting the size of the updates when the the previous steps are too large, making the search more conservative. An additional difference from vanilla SGD with momentum is that the learning rate of each parameter is modulated separately.

• The **Adam** optimizer combines heuristics from both momentum and from RMSProp. For each parameter, it stores an exponentially decaying average of past gradients and of past squared gradients, similarly to momentum and RMSProp, respectively. The learning rate is then multiplied by the exponentially decaying average of past gradients, divided by the exponentially decaying average of past squared gradients. Intuitively, Adam optimizer works by controlling the momentum gain by reacting to the size of the previous steps in a process similar to friction. The greater the magnitude of previous updates; that is, the greater the 'momentum' of the search, the greater the friction, and the more conservative the search becomes. A drawback of the Adam optimizer is that when the loss curvature is simple, SGD performs better because it is less complicated.

1.3 Long-Short Term Network (LSTM) in PyTorch

The the LSTM's core part is formalized by the following equations

$$\mathbf{g}^{(t)} = \tanh\left(\mathbf{W}_{gx}\mathbf{x}^{(t)} + \mathbf{W}_{gh}\mathbf{h}^{(t-1)} + \mathbf{b}_g\right) \tag{5}$$

$$\mathbf{i}^{(t)} = \sigma\left(\mathbf{W}_{ix}\mathbf{x}^{(t)} + \mathbf{W}_{ih}\mathbf{h}^{(t-1)} + \mathbf{b}_i\right) \tag{6}$$

$$\mathbf{f}^{(t)} = \sigma\left(\mathbf{W}_{fx}\mathbf{x}^{(t)} + \mathbf{W}_{fh}\mathbf{h}^{(t-1)} + \mathbf{b}_f\right) \tag{7}$$

$$\mathbf{o}^{(t)} = \sigma\left(\mathbf{W}_{ox}\mathbf{x}^{(t)} + \mathbf{W}_{oh}\mathbf{h}^{(t-1)} + \mathbf{b}_o\right) \tag{8}$$

$$\mathbf{c}^{(t)} = \mathbf{g}^{(t)} \odot \mathbf{i}^{(t)} + \mathbf{c}^{(t-1)} \odot \mathbf{f}^{(t)} \tag{9}$$

$$\mathbf{h}^{(t)} = \tanh\left(\mathbf{c}^{(t)}\right) \odot \mathbf{o}^{(t)} \tag{10}$$

were $\mathbf{g}^{(t)}$ is the input modulation gate, $\mathbf{i}^{(t)}$ is the input gate, $\mathbf{f}^{(t)}$ the forget gate, $\mathbf{o}^{(t)}$ the output gate, $\mathbf{c}^{(t)}$ the cell state, and $\mathbf{h}^{(t)}$ the hidden state. $\odot$ is element-wise multiplication and $\sigma(\cdot)$ is the sigmoid function. The linear output mapping from the core unit is given by

$$\mathbf{p}^{(t)} = \mathbf{W}_{ph}\mathbf{h}^{(t)} + \mathbf{b}_p \tag{11}$$

$$\hat{\mathbf{y}}^{(t)} = \text{softmax}\left(\mathbf{p}^{(t)}\right) \tag{12}$$

Question 1.5

(a) The purpose of the input modulation gate $\mathbf{g}^{(t)}$, input gate $\mathbf{i}^{(t)}$, forget gate $\mathbf{f}^{(t)}$, and the output gate $\mathbf{o}^{(t)}$ that make part of the core part of LSTM's is discussed below.

- **The input modulation gate**, $\mathbf{g}^{(t)} = \tanh\left(\mathbf{W}_{gx}\mathbf{x}^{(t)} + \mathbf{W}_{gh}\mathbf{h}^{(t-1)} + \mathbf{b}_g\right)$

The input modulation gate combines information from the current input, and previous hidden state, extracted from the two weight matrices $\mathbf{W}_{gx}$, and $\mathbf{W}_{gh}$, respectively, and a $tanh$ non-linearity to map the result into a space centered over 0, between the values of -1 and 1. This preserves topological properties of the feature space, but transforms it into a more mathematically convenient form. The input modulation gate essentially computes the new information that is available to be incorporated into the cell-state.

- **The input gate**, $\mathbf{i}^{(t)} = \sigma\left(\mathbf{W}_{ix}\mathbf{x}^{(t)} + \mathbf{W}_{ih}\mathbf{h}^{(t-1)} + \mathbf{b}_i\right)$

The input gate uses a sigmoid non-linearity to compute a weight for each component, or feature of the information computed by the input modulation gate, by mapping the feature space in a space between 0 and 1. The relative weight basically encodes the corresponding relative importance of each component, which is int turn determined by the relative magnitude of each component computed by a linear combination of the current input and previous hidden state. So the input gate essentially determines what information outputted from the input modulation gate to incorporate in the previous cell state to update it into the current cell state

- **The forget gate**, $\mathbf{f}^{(t)} = \sigma\left(\mathbf{W}_{fx}\mathbf{x}^{(t)} + \mathbf{W}_{fh}\mathbf{h}^{(t-1)} + \mathbf{b}_f\right)$

The forget gate, similarly to the input gate, uses a sigmoid non-linearity to compute a weight for each component of the previous cell state. However, the relative weight encodes the corresponding relative insignificance. Again the relative insignificance is determined by the relative magnitude of the corresponding component of the vector formed by performing weighted combinations of the current input and previous state. Intuitively, the forget gate is used to select what information from the previous cell gate to remove, based on how useful it is.

- **The output gate**, $\mathbf{o}^{(t)} = \sigma\left(\mathbf{W}_{ox}\mathbf{x}^{(t)} + \mathbf{W}_{oh}\mathbf{h}^{(t-1)} + \mathbf{b}_o\right)$

The output gate also uses a sigmoid non-linearity to compute the weight for each component of the updated cell state so that it selects the useful information from the current cell state under a $tanh$ non-linearity is incorporated in the hidden state.

(b) Given the LSTM cell as defined above with n number of units, an input sample $x \in \mathbb{R}^{Td}$ where T denotes the sequence length and d is the feature dimensionality, and a batch size of m, the total number of total number of trainable parameters for the LSTM cell is the number of all the parameters in the weight matrices $\mathbf{W}_{gx}, \mathbf{W}_{gh}, \mathbf{W}_{ix}, \mathbf{W}_{ih}, \mathbf{W}_{fx}, \mathbf{W}_{fh}, \mathbf{W}_{ox}, \mathbf{W}_{oh}$ and $\mathbf{W}_{ph}$, and biases $\mathbf{b}_g, \mathbf{b}_i, \mathbf{b}_f, \mathbf{b}_o$ and $\mathbf{b}_p$. The batch size m, or sequence T do not affect the number of parameters. Therefore, the formula for the total number of trainable parameters for the LSTM cell is given by

$$\begin{aligned}
\text{No. of parameters} &= \Big(\text{no. of parameters in LSTM's core part}\Big) + \Big(\text{no. of parameters in linear output mapping layer}\Big) \\
&= \Big(d_{in} \times \text{no. of hidden units} + \text{no. of hidden units} \times \text{no. of hidden units} + \text{no. of hidden units}\Big) \\
&\quad \times \text{no. of gates} + \Big(d_{out} \times \text{no. of hidden units} + d_{out}\Big) \\
&= \Big(d \times n + n \times n + n\Big) \times 4 + \Big(d_{out} \times n + d_{out}\Big)
\end{aligned}$$

were d_{in} and d_{out} are the feature dimensionality of the input and output, respectively.

Question 1.6

For this section, I implemented a one-layer LSTM network as specified by the equations above, using the given default parameters, and using the RMSProp optimizer for tuning the weights, and Cross-Entropy as my loss function. Again, I added the convergence condition $|\mathcal{L}^{(\tau+1)} - \mathcal{L}^{(\tau)}| > \epsilon$, were $\mathcal{L}$ is the training loss over the last 100 batches, τ is the current training step, and $\epsilon = 1e-6$. The loss and accuracy were computed by taking the average of the last 100 batches. The corresponding curves are illustrated below in Figure 3.

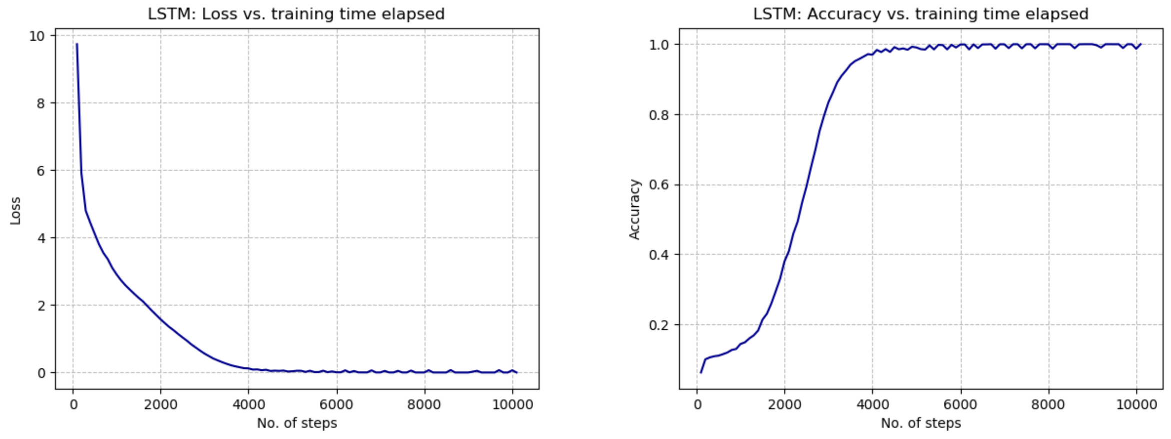


Figure 3: The Loss and Accuracy versus the training time elapsed for the one-layer LSTM, using the default parameters (sequence length = 10)

We can observe that our LSTM model is very good at learning pallindromes of length $L = 10$, learning to predict the last digit of the palindrome as early as 4000 training steps (loss = 0, accuracy = 1), compared to my Vanilla RNN model which could only reach an accuracy of 0.18 by the end of default max training steps of 10000. To examine the memorization capability of the one-layer LSTM, I followed the same protocol as I did for Question 1.3. I trained the model until corvergence on palindromes of length $L = \{5, 10, 15, 20, 25, 30, 35, 40, 45, 50\}$ using the first $L - 1$ digits of each palindrome as input to get the final digit as output. The loss and accuracy for all sequence lengths tested are illustrated in Figure 4, plotted on the same axis as the Vanilla RNN results for comparison.

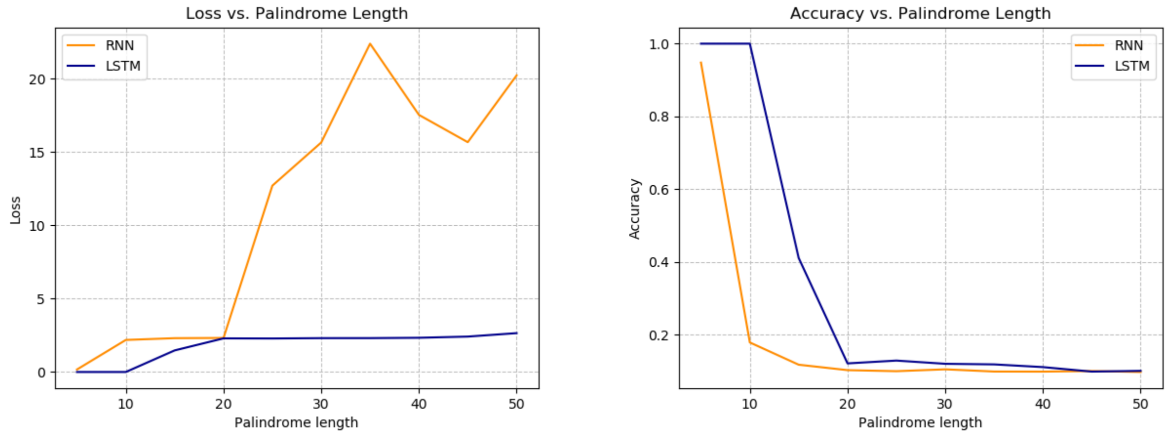


Figure 4: The Loss and Accuracy versus the palindrome length for the Vanilla RNN (orange), and LSTM (blue) implementations, using the default parameters

The results above clearly show my LSTM model outperforming my Vanilla RNN, with peak performance until palindrome lengths of 10. The performance of the Vanilla RNN decreases dramatically after palindromes of size 5, wereas LSTM shows a similar sharp decrease in performance after sequences of size $L = 10$. The LSTM outperforming the Vanilla RNN was expected, as the use of gates to control information flow in and out of the hidden state (LSTM) does a better job at preserving information than no modulation (Vanilla RNN), delaying the onset of forgetting, and also keeping the cell state and hidden state separate stops the problem of vanishing gradients to an extend.

2 Recurrent Nets as Generative Model

Question 2.1

(a) For this section, I implemented a two-layer LSTM network, using the given default parameters, and using the RMSProp optimizer for tuning the weights. I defined the total loss as average of cross-entropy loss over all timesteps, which is performed automatically by the $torch.nn.CrossEntropyLoss()$ function. I trained the model on sentences of length $L = 30$ from the book "Grimms' Fairy Tales", to predict the next character in a sentence. As in all my previous implementations, I added the convergence condition $|\mathcal{L}^{(\tau+1)} - \mathcal{L}^{(\tau)}| > \epsilon$, were $\mathcal{L}$ is the training loss over the last 100 batches, τ is the current training step, and $\epsilon = 1e - 6$. Again, the loss and accuracy were computed by taking the average of the last 100 batches. The corresponding curves are illustrated below in Figure 5.

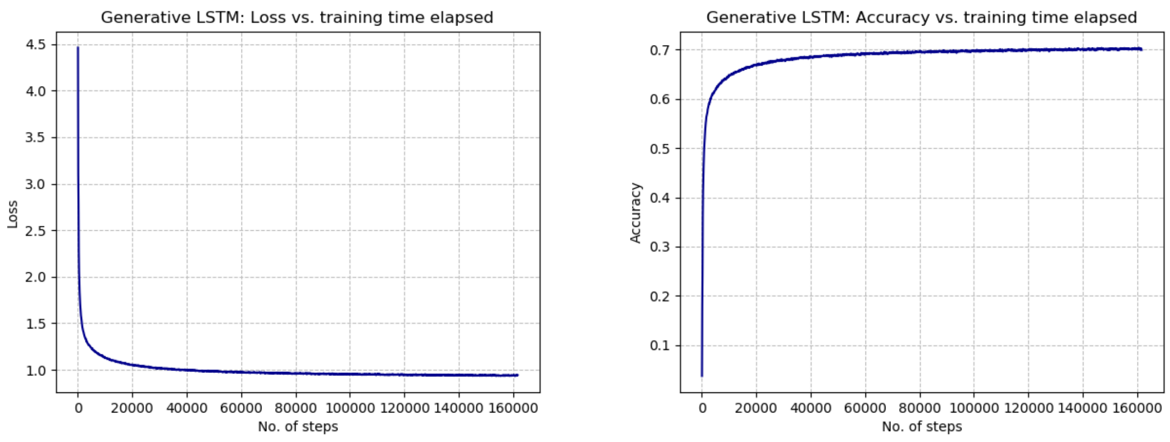


Figure 5: The Loss and Accuracy versus the training time elapsed for the two-layer LSTM network implementation, using the default parameters, over 20 epochs (1 epoch $\approx$ 18000 steps)

As we can see, the two-layered LSTM performance improves in a pattern similar to what we have observed for the one-layered LSTM. The loss converges to 0.94 (2 d.p.), and the final accuracy is 0.70 (2 d.p.).

(b) For this part of the assignment, I used the network described above to generate new sentences of length $L = 30$ every 100 steps of training (each step is training on a batch of 64 sentences of length $L = 30$). This was done by randomly picking a first letter from the current vocabulary, performing a forward pass using the current state of the trained model, and sampling from the probability distribution over the vocabulary from the output of the softmax layer the character with the greatest probability. After sampling the next character, this was used as input for the forward pass of the trained model until a string of 30 characters was complete. 5 text samples generated by the network over intervals of 500 training steps are shown below in Table 1

Training Step	Generated Sentence	Training Accuracy
0	"t "	0.038
500	"@ the said the said the said th"	0.556
1000	"6 and the was the was the was t"	0.596
1500	"Jo the wine was the wolf was th"	0.615
2000	" Fout and the king when he had b "	0.628

Table 1: Qualitive results of generative two-layered LSTM model. Training accuracy (shown to 3 s.f.) is computed using the average accuracy of the previous 100 training steps.

As we can see, as the training steps increase, the model starts becoming more articulate. At the first 500 steps we can see the word "the" is generated in high frequency. This is probably because it is used in analogously high frequency in the text, so it is one of the first words our model learns. The subsequent generated sequences in the table start becoming more complicated in both vobulary and semantics, as the model initially learns long-term dependencies between characters, therefore learning words, and then learning even longer distance dependencies, therefore learning sentences, by capturing sequential patterns as complex as syntactic rules, such as using the correct pronouns or a consistent tense, and compositional semantics.

(c) For this part of the assignment, I extended my current model through the addition of a temperature parameter T that controls the balance of the sampling strategy from the probability distribution over the vocabulary generated at the output of the softmax layer, described in above. This was implemented by changing equation (12) as follows

$$\hat{\mathbf{y}}^{(t)} = \mathrm{softmax}\left(\mathbf{p}^{(t)}\right) \;\rightarrow\; \hat{\mathbf{y}}^{(t)} = \mathrm{softmax}\left(\frac{\mathbf{p}^{(t)}}{T}\right)$$

were $\hat{\mathbf{y}}^{(t)}$ is the propability distribution over the characters of our vocabulary at training step t, and T is the temperature parameter (Goodfellow et al., 2016). Increasing the value of the T parameter dampens the shape of the propability distribution, rendering the differences between the probability densities of each character less pronounced. Therefore, our sampling strategy shifts from fully-greedy when $T \approx 0$, to fully-random when $T \approx \infty$. Sentences generated using the fully trained 2-layered LSTM generative model for $T = \{0.5, 1, 2\}$, and multinomial sampling ($torch.multinomial()$ function) are shown below in Table 2

Temperature	Sentence 1	Sentence 2	Sentence 3
0.5	"As soon as the hall on the grou"	"Yes, what a spoking, but she sa"	"; then she saw the old king was"
1	"Comranced, and for hunger or no"	"'I have thus beautiful forest t"	"I like this married the prize;"
2	"'Ut as if shoee.' The wlogs wer"	"YOU. Snowdrer to one, has voice"	"Dlancencledival yal on you!now"

Table 2: Qualitive results of the generative two-layered LSTM model when using different values for temperature T.

As we can see in the table above, using a low value for temperature decreases ambiguidy and results in generating coherent sentences, as the slope of the propability distribution $\hat{\mathbf{y}}^{(t)}$ becomes more curved, decreasing the chance of random characters being sampled. As the value of T increases, the sentences become semantically compromised. We can see when $T = 1$, the we start observing spelling mistakes, even though most words are correct, and word combinations start not making sense. This is because even though $\frac{\mathbf{p}^{(t)}}{T} = \mathbf{p}^{(t)}$, we are using multinomial sampling, to introduce some degree of stochasticity, in order to make our sentences more variable to increase our examples of how temperature affects generated sentences. Finally, using a temperature of $T > 1$ makes sentences unintelligable, as we can see when $T = 2$, were now sentences are complete nonsense, loosing any compositional semantics, and most words are gibberish, indicating that the slope of $\hat{\mathbf{y}}^{(t)}$ is now quite uniform, so basically, the characters sampled are almost completely random, and are not subject to any of the sequential patterns learned by the model.

Question 2.2

For this question part, I made my model generate sentences of size $L = 500$ using the given default parameters, and $T = 0.5$. An example of the generated sentences is shown below:

"'I will soon learn what he could not get in'; so they saw a beautiful court were pocket off the mill went the beautiful princess, you may do it.' But the silvery looked at the door, and the soldier was cleaned on the other side of his fire, that the seven hair benomissing at home, and the spindle which he had walked for his son, and said, 'Let me live, and I shall not think of years of an inn, ordered with all his breth straw till the fifth half. When the dance beside.' Then the servant was so"

As we can see, the sentences are syntactically sound, even using quotes for dialogue. However, even if short parts of the sentences make sense, when looking at a whole sentence or paragraph, we can see that the model is not coherent, and it almost seems like it forgets the subject of the text periodically, similarly to a dementia patient. This illustrates the limits of the learning capacity of the model. Even though it impressively captures various levels of patterns and rules in natural language, the model capabilities are limited at the macroscopic level. I would imagine that tuning model parameters such as temperature value, dropout, weight decay etc. could slightly improve results, however, the macroscopic problem would still remain.

3 Graph Neural Networks

3.1 GCN Forward Layer

Question 3.1

(a) The propagation rule for a layer in the Graph Convolutional Network architecture is given by

$$H^{(l+1)} = \sigma\left(\hat{A}H^{(l)}W^{(l)}\right) \tag{13}$$

were $\sigma(.)$ is the activation function, $H^{(l)}$ is the $N \times d$ matrix of activations in the l-th layer, W is a learnable $d^{(l)} \times d^{(l+1)}$ matrix, and $\hat{A}$ is the transformed adjacency matrix, computed by

$$\hat{A} = \tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}} \tag{14}$$

$$\tilde{A} = A + I_N \tag{15}$$

$$\tilde{D}_{ii} = \sum_j \tilde{A}_{ij} \tag{16}$$

Here, $\tilde{D}$ is a diagonal degree matrix used for normalization, and $\tilde{A}$ is the $N \times N$ adjacency matrix A with added self loops, by adding the identity matrix of size N, I_N.

Now, the structural information in the graph data is encoded in the adjacency matrix A, which describes the graph structure by indicating the connections of each node in a binary fashion. The input of the current layer; that is, the activations of the previous layer $H^{(l)}$, can be seen as a feature description for each node of the graph. By multiplying A with $H^{(l)}$, for each node, A 'selects' the corresponding neighboring nodes, and the new representation of each node would be the sum of the features of it's neighbours. In order to ensure that the updated representation of each node retains its own features too, we enforce self-loops in A by adding the identity matrix to A, to get $\tilde{A}$, thus, the new representation of each node is a mixture of the features of all its direct connections.

Furthermore, symmetric normalization is performed on $\tilde{A}$ to give $\hat{A} = \tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}$, ensuring that each row of the transformed adjacency matrix $\hat{A}$ sums up to one. This makes the updated representation of each node more concise, and prevents the feature vectors in the updated representation of each node from completely changing scale (which would disrupt optimization by SGD as it is sensitive to scale).

W allows us to learn to weigh the features evaluated from $\hat{A}H^{(l)}$ according to usefulness. However, since the W is shared across nodes, parameter learning occurs in a holistic fashion, and therefore depends on the overall structure of the graph, which in turn affects the representation of each node. Therefore each GCN layer exploits the structural information in the graph data by updates the feature description, or otherwise, embedding of each node through incorporating information from its direct connections, and learning a weighting scheme for each feature that depends on all the direct connections of all the nodes.

Finally, the activation function $\sigma(.)$ acts to transcribe the updated feature descriptions of the nodes in a more mathematically convenient space (i.e. establish a non-linear gradient signal to be exploited for training), to be used as input to the next layer, which can be interpreted as performing message passing over the graph, since in the next layer, each node willr receive information on higher degree connections (the connections of their connections) (Kipf and Welling, 2016).

(b) To propagate information from nodes 3 connections away (3rd degree connections) to each node's embedding, we would need at most 3 stacked GCN layers, as at each step, each node receives information from its direct connections, as described above. Therefore if we zoom in at a specific node, at the first GCN layer it will have information only nodes 1 connection away, and these nodes in turn will have information from only their direct connections. Then at the second GCN layer, our node will be updated by receiving the new information acquired by its direct connections, which consists of information from their respective direct connections. And if we go to the third layer, our node will now receive information from 3rd degree connections.

We could also propagate information from 3rd degree connections with less layers if we adjust the adjacency matrix $\hat{A}$ to include secondary or tertiary connections instead of only direct connections.

3.2 Applications of GNNs

Question 3.2

GNN's are applicable to any dataset that has the form of a graph or a network. Some examples of real-world applications of GNN's are

• **Knowledge graphs**. For example Hamaguchi et al. (2017) use graph neural networks to transfer knowledge for out-of-knowledge-base entities; that is, use elements from information encountered during training time to infer new information without retraining.

• Modelling **protein-interaction networks**. Fout et al. (2017) use a graph convolution approach to predict protein interfaces

• Modelling **physical systems**, as explored by Battaglia et al. (2016), who utilize a graph neural network to build a general-purpose learnable physics engine used to reason about objects and relations in complex physical systems.

• **Text classification**, such as the paper by Yao et al. (2019), were a graph convolutional network is used to classify tet without any external word embeddings or knowledge.

3.3 Comparing and Combining GNNs and RNNs

Question 3.3

(a) RNN-based models are well suited for sequential data because for each new input, the RNN aggregates information from the input into a hidden representation that will capture patterns unfolding over timesteps. Therefore, RNNs can capture long-term dependencies over specific sequences. In the case of Vanilla RNNs, the same weights are used to combine the new input with the current representation, therefore, the same network can be used on arbitrary sequence lengths without increasing the number of parameters. However, as we have seen before, weight sharing comes with the cost of making training harder due to vanishing or exploding gradients, and due to 'forgetting' as explained in section 1. LSTMs address the problem of 'forgetting' and the problem of vanishing gradients by including gated modules in each NN layer that exert better control at what information is preserved over longer sequences, however, this is at the cost of increasing model parameters.

Similarly to RNN-based models, GNNs also make use of weight-sharing which reduces the computational cost, and makes the model more flexible towards input of different sizes. GNNs, however, explicitly model the relationship between the objects in the input by collectively aggregating information from graph structure. As we saw previously, GNNs utilize garph structure to guide propagation, updating the hidden state of nodes by a weighted sum of their neighbours states. Again, this allows GNN's to capture long-distance dependencies, but also makes them invariant to the input order of the nodes, which is a particularly useful property when dealing with graph structured data, since the nodes in a graph don't exhibit a natural order. RNNs on the other hand would stack the states of nodes in a specific order, that could bias the new representation of nodes towards the latest node visited. Consequently, in order for RNNs to accurately model a graph, all the possible orders should be traversed, making the task much more inefficient. Therefore, GNNs would be preferred over RNN-based models on datasets such as knowledge bases, genomics, medical ontologies etc.

When it comes to natural language modelling, RNN-based models would outperform GNN models, as both written sentences or speech exhibit temporal dynamic behaviour. However, sentences can also be represented using dependency trees, that capture the syntactic properties of sentence structure in a graph format (Tai et al., 2015), blurring the line between which model family would be the clear winner in such tasks. Using these representations, I would expect GNNs to show some good results, however, not as good as LSTMs, as the specific order of a sentence is very important for sentence semantics.

In the case of images, using purely RNN-based models would not give a very good result, as representing static images as sequences of pixels would not be very useful, since important information about the overall spatial layout of the image would be lost. As we have learned by the success of Convolutional neural networks, it is important to pull information together from local patches of images and to preserve the overall spatial structure of the image, by capturing hierarchical features. GNNs have a multi-layer structure, which is critical when it comes to dealing with hierarchical patterns, allowing us to capture features of various sizes. Therefore, I would expect GNNs to preserve spatial information better than RNNs. When dealing with videos, however, even though RNN-based models cannot be used directly on videos for the same reason as for images, combining these with convolutional networks to represent frames would allow us to capture temporal dependencies between the frames, and would outperform GNNs or CNN with GNN combinations

In conclusion, both RNNs and GNNs capture long term dependencies, but the former is better at learning to recognize patterns occuring in specific sequences of data, wereas the latter is better at learning patterns in our data that do not lie in a euclidean space; that is, do not necessarily have a specific unidirectional order. Therefore, RNNs would be more expressive to dynamic unidirectional sequences, and GNNs to data displaying nested structures, as analysed above (Zhou et al., 2018).

(b) GNNs and RNNs models could be combined by using GNNs to map graph structures into low dimensional embeddings that represent the graph nodes, and accomodate information from both the individual graph nodes and their neigbours, and then using RNNs to model changes in these embeddings over time. For example, if we want to model the behaviour of the structure of a protein during a chemical reaction (for example if a protein is a digestive enzyme and it reacts with a substrate). We can have a graph representing protein structure, with the graph nodes representing atoms, and using the GNNs, we could capture representations for the different atoms in their role as being part of a molecular structure, by aggregating information on their neighbouring atoms. Subsequently, we can use RNNs to model the conformational changes ; that is, the structural changes that occur in these molecules when the protein engages in some activity. I could not find anything related in the literature, however, Jin and F. JaJa (2018) use a similar approach, showing that it is effective.

References

Battaglia, P., Pascanu, R., Lai, M., Rezende, D. J., and kavukcuoglu, K. (2016). Interaction networks for learning about objects, relations and physics. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*, NIPS'16, pages 4509–4517, USA. Curran Associates Inc.

Fout, A., Byrd, J., Shariat, B., and Ben-Hur, A. (2017). Protein interface prediction using graph convolutional networks. In Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., and Garnett, R., editors, *Advances in Neural Information Processing Systems 30*, pages 6530–6539. Curran Associates, Inc.

Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press. `http://www.deeplearningbook.org`.

Hamaguchi, T., Oiwa, H., Shimbo, M., and Matsumoto, Y. (2017). Knowledge transfer for out-of-knowledge-base entities: A graph neural network approach. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, IJCAI'17, pages 1802–1808. AAAI Press.

Jin, Y. and F. JaJa, J. (2018). Learning graph-level representations with gated recurrent neural networks.

Kipf, T. N. and Welling, M. (2016). Semi-supervised classification with graph convolutional networks. *CoRR*, abs/1609.02907.

Tai, K. S., Socher, R., and Manning, C. D. (2015). Improved semantic representations from tree-structured long short-term memory networks. *CoRR*, abs/1503.00075.

Yao, L., Mao, C., and Luo, Y. (2019). Graph convolutional networks for text classification. *CoRR*, abs/1809.05679.

Zhou, J., Cui, G., Zhang, Z., Yang, C., Liu, Z., and Sun, M. (2018). Graph neural networks: A review of methods and applications. *CoRR*, abs/1812.08434.