Assignment 2
Recurrent Neural Networks and Graph Neural Networks

Francesco Dal Canton
f.dal.canton@student.uva.nl
12404225

1 Vanilla RNN versus LSTM

1.1 Toy Problem: Palindrome Numbers

1.2 Vanilla RNN in PyTorch

1.1 Throughout this assignment I use the notation x_T in place of $x^{(T)}$ used in the assignment text. I also make the choice to ignore the $tanh$ function, which would only appear as an additional ∂-term in each application of the chain rule.

That said, we can easily write the expression for the gradient of the loss w.r.t. the output weights as follows.

$$\begin{aligned}\frac{\partial \mathcal{L}_T}{\partial W_{ph}} &= \frac{\partial \mathcal{L}_T}{\partial \hat{y}_T}\frac{\partial \hat{y}_T}{\partial p_T}\frac{\partial p_T}{\partial W_{ph}} \\ &= \frac{\partial \mathcal{L}_T}{\partial \hat{y}_T}\frac{\partial \hat{y}_T}{\partial p_T} h_T^T\end{aligned}$$

Defining the gradient w.r.t the hidden weights is trickier. We can start by writing out the standard chain rule formulation, which we can compress by omitting parts that we've defined in the previous assignment (i.e. cross-entropy, softmax, etc.).

$$\begin{aligned}\frac{\partial \mathcal{L}_T}{\partial W_{hh}} &= \frac{\partial \mathcal{L}_T}{\partial \hat{y}_T}\frac{\partial \hat{y}_T}{\partial p_T}\frac{\partial p_T}{\partial h_T}\frac{\partial h_T}{\partial W_{hh}} \\ &= \frac{\partial \mathcal{L}_T}{\partial h_T}\frac{\partial h_T}{\partial W_{hh}}\end{aligned}$$

We notice that any h_t depends recursively on h_{t-1} which itself depends on W_{hh}. We can then define the gradient of the last hidden state w.r.t. its weights, and develop it using the product rule.

$$\begin{aligned}\frac{\partial h_T}{\partial W_{hh}} &= \frac{\partial \left[W_{hx}x_T + W_{hh}h_{T-1} + b_h\right]}{\partial W_{hh}} \\ &= \frac{\partial \left[W_{hh}h_{T-1}\right]}{\partial W_{hh}} \\ &= \frac{\partial W_{hh}}{\partial W_{hh}} h_{T-1} + W_{hh}\frac{\partial h_{T-1}}{\partial W_{hh}} \\ &= h_{T-1} + W_{hh}\frac{\partial h_{T-1}}{\partial W_{hh}}\end{aligned}$$

Where, at the end of the recursive chain, we reach the gradient of h_0 w.r.t. the hidden weights, which is 0. With this knowledge, we can expand the chain in order to compress it again.

$$\begin{aligned}\frac{\partial h_T}{\partial W_{hh}} &= h_{T-1} + W_{hh}\left(h_{T-2} + W_{hh}\left(h_{T-3} + W_{hh}\left(\dots\right)\right)\right) \\ &= h_{T-1} + W_{hh}h_{T-2} + W_{hh}^2 h_{T-3} + \dots + W_{hh}^{T-2} h_{T-(T-1)} \\ &= \sum_{i=1}^{T-1} W_{hh}^{i-1} h_{T-i}\end{aligned}$$

So that we can express the original gradient as:

$$\frac{\partial \mathcal{L}_T}{\partial W_{hh}} = \frac{\partial \mathcal{L}_T}{\partial h_T} \sum_{i=1}^{T-1} W_{hh}^{i-1} h_{T-i}$$

The most striking difference between the gradient of the loss w.r.t. the output weights and the one w.r.t. the hidden weights is that the latter involves a temporal dependency that extends all the way to the first time step of the cell's operation. In other words, after feeding the RNN a sequence of n inputs, in order to perform backpropagation we need to multiply the gradient by the hidden weight matrix a large number of times, which grows exponentially with the number of time steps.

This fact causes the well known problem of vanishing or exploding gradients. When performing an update over a large number of timesteps, the gradient that should pertain to the gradients further back in time either reduces to 0 or it increases to infinity, depending on whether the hidden weights are small or large. This makes it very hard to train such networks for long sequences, since those longer time dependencies can't be learned.

1.2 The implementation of the vanilla recurrent neural network was done as instructed.

In the `train.py` script, this line of code is used:

```
torch.nn.utils.clip_grad_norm_(model.parameters(),
                               max_norm=config.max_norm)
```

Note that `clip_grad_norm_` was used instead of `clip_grad_norm` since the latter is deprecated. This is a way to avoid the problem of exploding gradients. This function clips the maximum value of the norm of the gradients. The effect is that there is no risk of multiplying gradients of high magnitude over and over, thus leading to huge update steps that make it impossible for the network to learn. Note, however, that the vanishing gradient problem still exists since the gradients can have a very low norm.

1.3 After the first rounds of experimentation, I noticed sudden drops in training accuracy after convergence, which all quickly recovered. I attribute those to the exploding gradients, and tried using lower values for the maximum norm value used in the clipping of the gradients. After trying values of 10, 5, 2, 1, and 0.5 with a baseline sequence length of 5, however, I realised that except for slight variations the problem still occurred. As such I decided to stick with the original value of 10. All other parameters were kept at their default values.

The strategy used to detect convergence is as follows: each 200 training steps, the average accuracy is computed for the last 200 steps. When the absolute difference between the current average accuracy and the previous average accuracy is smaller than 10^{-3}, we say that the model has converged, and store the last average accuracy. The tested input lengths were in the range $[5, 35]$. The result is shown in Figure 1.

1.4 The advantage of RMSprop is the usage of an adaptive learning rate. This element comes into play as the parameter update is calculated using a running average of the previous squared gradients, weighted according to the decay hyper-parameter. The result is that each new gradient is normalized based on the norms of the current and previous gradients. This leads to a sensible way of reducing the damage caused by exploding gradients.

Adam works in a similar way, but it also incorporates a momentum term. This is done by computing the next gradient update as a running weighted average of the previous gradients,

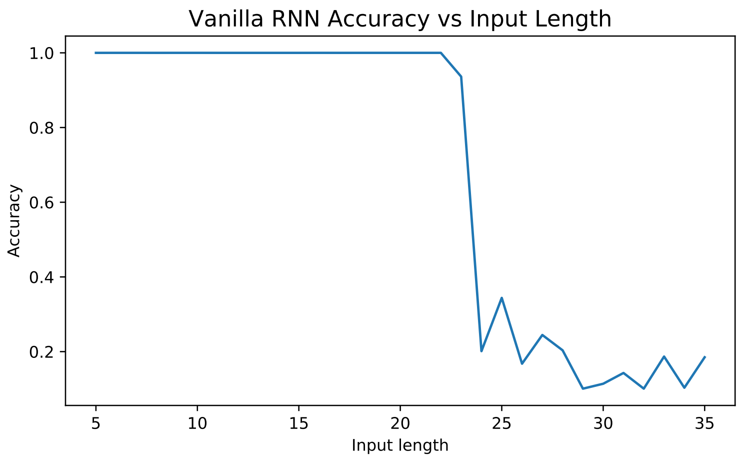

Figure 1: Average accuracy of the Vanilla RNN model in the last 200 training steps plotted against input length.

each of which is normalized in a way similar to that which RMSprop uses. The effect is that each parameter update is influenced by the gradients that preceded it: if the gradient is descending a steep hill it will accelerate, while if it is traversing a mostly flat region with uncertain topology it will slow down.

Both of these strategies are beneficial towards training, and they surpass SGD in training speed and accuracy.

1.3 Long-Short Term Network (LSTM) in PyTorch

1.5 **(a)** The input modulation gate $\mathbf{g}^{(t)}$ determines how each feature of the input affects the hidden state of the LSTM cell given the previous hidden state. Since the nonlinearity used in this gate is the $tanh$ function, each output feature will be in the range $(-1, 1)$, which makes sense since it normalizes the effect that each variable may have on the cell state, while also centering the activations around 0.

The input gate $\mathbf{i}^{(t)}$ determines how much influence each feature update, computed using $\mathbf{g}^{(t)}$, should affect the cell state. The softmax nonlinearity is a good choice for this purpose since its value is in the range $(0, 1)$. This means that when $\mathbf{i}^{(t)}$ is multiplied with $\mathbf{g}^{(t)}$, the magnitude of each feature update is modulated, but it can never exceed the original range of $(-1, 1)$.

The forget gate $\mathbf{f}^{(t)}$ is similar in nature to the input gate: it modulates the magnitude of the cell state as it flows through the cell, thus controlling which features are forgotten and which ones are left unchanged. The nonlinearity used in this gate is again the softmax function. For the same reason as the input gate, multiplying each feature of the cell state with a number in the range $(0, 1)$ makes it so that each feature is reduced in magnitude if it should be forgotten, and left unchanged if not (with all the steps inbetween).

The output gate $\mathbf{o}^{(t)}$ works once again in a similar way: it controls which parts of the cell state are forgotten, so that the rest is both given as the cell output and as the hidden state for the next time step. This gate works with the well chosen sigmoid nonlinearity according to the same principle as described in the previous two paragraphs. The cell state, after being passed through a $tanh$ function, is filtered by modulating the magnitude of each filter.

(b) In calculating the number of trainable parameters, both T and m don't come into play, since the parameters are shared across time steps and batch samples. To compute the number of parameters, it suffices to sum the number of parameters of each weight matrix used on the input ($n \times d$), of each weight matrix used on the hidden and cell states ($n \times n$), and of each bias vector (n). The result is:

$$\begin{aligned} N &= 4(n \times d) + 5(n \times n) + 5n \\ &= 5n^2 + 9n + 4d \end{aligned}$$

1.6 The LSTM network was implemented as instructed. The only difference was, instead of summing the linear transformations of input and previous hidden state, the two were concatenated and linearly transformed with a combined weight matrix.

The experimental setup for evaluating this model was identical to that used in question **1.3**. The results are shown in Figure **??**.

Run the same experiment as for the RNN

2 Recurrent Nets as Generative Model

2.1 **(a)**

(b)

(c)

2.2

3 Graph Neural Networks

3.1 GCN Forward Layer

3.1 **(a)**

(b)

3.2 Applications of GNNs

3.2

3.3 Comparing and Combining GNNs and RNNs

3.3 **(a)**

(b)