Assignment 2: RNNs & GCNs

Ipek Ganiyusufoglu | 12225428
ipek.ganiyusufoglu@student.uva.nl

1 Vanilla RNN versus LSTM

1.1 RNN, Question 1.1

Below, all derivatives can be found. As we expands derivatives with the chain rule, we see that they share some of the derivatives that we have to calculate, hence I have shown all three derivations in the chain rule for the first one, then in the following two derivations of loss I left the parts already shown un-expanded.

$$\frac{\partial L^{(T)}}{\partial W_{ph}} = \frac{\partial L^{(T)}}{\partial \hat{y}_k^{(t)}} \frac{\partial \hat{y}_k^{(t)}}{\partial p_l^{(t)}} \frac{\partial p_l^{(t)}}{\partial W_{ph}} \tag{1}$$

$$= -y_k \frac{1}{\hat{y}_k^{(t)}} \frac{\partial \hat{y}_k^{(t)}}{\partial p_l^{(t)}} \frac{\partial p_l^{(t)}}{\partial W_{ph}} \tag{2}$$

$$\frac{\partial \hat{y}_k^{(t)}}{\partial p_l^{(t)}} = \frac{\partial}{\partial p_l^{(t)}} softmax(p_l^{(t)}) = \begin{cases} \hat{y}_{ki}^{(t)}(1-\hat{y}_{ki}^{(t)}) & \text{if } i == j \\ -\hat{y}_{kj}^{(t)}\hat{y}_{ki}^{(t)} & \text{else} \end{cases} \tag{3}$$

$$= \begin{bmatrix} \hat{y}_{ki}^{(t)}(1-\hat{y}_{ki}^{(t)}) & \dots & -\hat{y}_{kj}^{(t)}\hat{y}_{kn}^{(t)} \\ -\hat{y}_{k(j+1)}^{(t)}\hat{y}_{ki}^{(t)} & \ddots & \vdots \\ \vdots & & \vdots \\ -\hat{y}_{kn}^{(t)}\hat{y}_{ki}^{(t)} & \dots & \hat{y}_{kn}^{(t)}(1-\hat{y}_{kn}^{(t)}) \end{bmatrix} \tag{4}$$

$$\frac{\partial p_l^{(t)}}{\partial W_{ph}} = \frac{\partial}{\partial W_{ph}}(W_{ph}h_l^{(t)} + b_p) = h_l^{(t)} \tag{5}$$

$$\frac{\partial L^{(T)}}{\partial W_{ph}} = \frac{\partial L^{(T)}}{\partial \hat{y}_k^{(t)}} \frac{\partial \hat{y}_k^{(t)}}{\partial p_l^{(t)}} \frac{\partial p_l^{(t)}}{\partial W_{ph}} \quad \text{bringing it all together} \tag{6}$$

$$= -\frac{y_k}{\hat{y}_k^{(t)}} \begin{bmatrix} \hat{y}_{ki}^{(t)}(1-\hat{y}_{ki}^{(t)}) & \dots & -\hat{y}_{kj}^{(t)}\hat{y}_{kn}^{(t)} \\ \vdots & \ddots & \vdots \\ -\hat{y}_{kn}^{(t)}\hat{y}_{ki}^{(t)} & \dots & \hat{y}_{kn}^{(t)}(1-\hat{y}_{kn}^{(t)}) \end{bmatrix} h_l^{(t)} \tag{7}$$

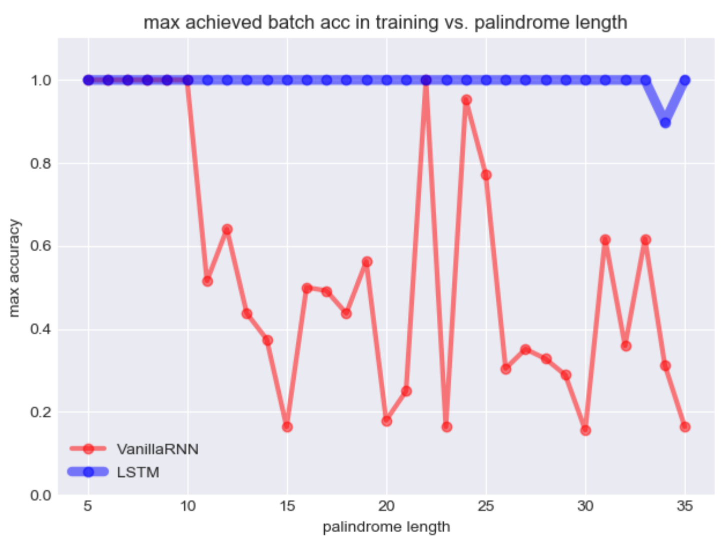

Figure 1: Maximum accuracy achieved for RNN and LSTM for different palindrome lengths.

$$\frac{\partial L^{(T)}}{\partial W_{hh}} = \frac{\partial L^{(T)}}{\partial \hat{y}_k^{(t)}} \frac{\partial \hat{y}_k^{(t)}}{\partial p_l^{(t)}} \frac{\partial p_l^{(t)}}{\partial h_m^{(t)}} \frac{\partial h_m^{(t)}}{\partial W_{hh}} \tag{8}$$

$$\frac{\partial p_l^{(t)}}{\partial h_m^{(t)}} = W_{ph} \tag{9}$$

$$\frac{\partial h_m^{(t)}}{\partial W_{hh}} = \frac{\partial}{\partial W_{hh}} \tanh\left(W_{hx}x_m^{(t)} + W_{hh}h_m^{(t-1)} + b_h\right) = (1-(h_m^{(t)})^2)h_m^{(t-1)} \tag{10}$$

$$\frac{\partial L^{(T)}}{\partial W_{hh}} = \frac{\partial L^{(T)}}{\partial \hat{y}_k^{(t)}} \frac{\partial \hat{y}_k^{(t)}}{\partial p_l^{(t)}} W_{ph}(1-(h_m^{(t)})^2)h_m^{(t-1)} \tag{11}$$

$$\frac{\partial L^{(T)}}{\partial W_{hx}} = \frac{\partial L^{(T)}}{\partial \hat{y}_k^{(t)}} \frac{\partial \hat{y}_k^{(t)}}{\partial p_l^{(t)}} \frac{\partial p_l^{(t)}}{\partial h_m^{(t)}} \frac{\partial h_m^{(t)}}{\partial W_{hx}} \tag{12}$$

$$\frac{\partial h_m^{(t)}}{\partial W_{hx}} = (1-(h_m^{(t)})^2)x_m^{(t)} \tag{13}$$

$$\frac{\partial L^{(T)}}{\partial W_{hx}} = \frac{\partial L^{(T)}}{\partial \hat{y}_k^{(t)}} \frac{\partial \hat{y}_k^{(t)}}{\partial p_l^{(t)}} \frac{\partial p_l^{(t)}}{\partial h_m^{(t)}}(1-(h_m^{(t)})^2)x_m^{(t)} \tag{14}$$

1.2 RNN, Question 1.2

Implemented as instructed.

1.3 RNN, Question 1.3

On figure 1, we see the how RNN models accuracy changes as the palindrome length increases. Clearly RNN is not able to maintain a good accuracy at all as the sequences get longer. This is because of RNNs incapability of capturing long-term dependencies, which LSTMs can handle.

1.4 RNN, Question 1.4

Mini-batch SGD is often the first optimizer we turn to when using neural networks, so far at least. It already has many advantages over GD, such as less variance in consecutive parameter updates, efficient matrix gradient computations etc. but it still has issues such as picking a proper learning rate, local minima and being unable to update parameters individually.

If our loss space has pathological curvatures etc, it is possible for m-SGD to oscillate and hence proceed very slowly, especially if we have a fixed and large learning rate. To over-come such challenging surfaces we can use methods that consider the bigger picture, e.g. remember previous updates. One such method is **momentum**, which updates weights with a moving average of the gradients of last couple of steps, reducing the stochasticity. This hinders oscillation as now the exponential average update is more general rather than local & robust, hence also converges faster.

RMS-prop is a method that uses **adaptive learning rates**. As mentioned, as huge issue in a gradient update is picking a general learning rate to fit every update, while gradients vary greatly in size. We want large gradients to be tamed (noisy surface) while the smaller ones should be emphasized (e.g. escaping a plateau). We also want to update weight parameters individually. RMS-prop keeps a moving average over squared gradients for each weight, then the gradient is divided by the square of this (i.e. root-mean-square). Because of these adaptive updates RMS-prop is able to avoid false huge gradient updates and make the small ones more aggressively, hence converge faster.

Adam combines momentum and the idea of RMS-prop (adaptive momentum). Adam is found to be superior to RMS-prop in convergence. To summarize, smarter optimizers tackle the mentioned problems of a vanilla SGD algorithm and offer faster convergence.

I've read the following articles to help grasp the concepts better, links: RMS-Prop [3], Adam [4].

1.5 LSTM, Question 1.5

(a) 1) $f^{(t)}$ The **forget gate** regulates how much information of the cell state we are going to *forget*, as it can be seen in $c^{(t)}$ formulation its multiplied with $c^{(t-1)}$. It's a function of the input and the remembered hidden state. It can decide how much we forget of each number in $c^{(t-1)}$ via its sigmoid non-linearity, which outputs a number between 0 & 1, where 0 means we forget entirely.

2) $i^{(t)}$. The **input gate** regulates how much we update the cell state, and the $g^{(t)}$ **input modulation gate** new values to update it with, hence the multiplication of these together give us the new cell state update, which is also combined with how much we forget the previous one.

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

$i^{(t)}$ uses a sigmoid, as it weights every value by how much it should be updated with a value between 0 & 1. $g^{(t)}$ uses a $tanh$ which kind-of looks like a sigmoid stretched in y-dimension and maps to a range of -1 to 1. Sigmoid is preferred as a gating mechanism because of the range, however $tanh$ here used because of its derivative which is able to sustain a gradient for a longer range compared to other non-linearities, hence over-comes the infamous vanishing gradient problem of RNNs. It is also **centered** at zero which preferable especially when the data is **normalized** as well, as then they are aligned.

3) $o^{(t)}$ The output gate decides how much and what parts of the new cell-state $c^{(t)}$ we've just updated are we going to remember as the new hidden state $h^{(t)}$. Again as this is a gating/weighting mechanism, it uses a sigmoid. But before it is multiplied with the cell state, $c^{(t)}$ is put through a $tanh$ again to center the values at 0.

(b) The total number of trainable parameters is basically the sum of sizes of weights and biases, which is *independent* of the batch size and the sequence length as it just depends on the input length (number of features) in size. So the number of parameters is below, where h is the size of the hidden layer.

$$n \cdot \Big(4 \cdot (h \cdot i + h \cdot h + h \cdot 1) + d \cdot h + d \cdot 1\Big)$$

Here i is the input dimension, which would be equivalent to d (feature/class depth) if we were using one-hot representations for the input. n is the number of LSTM cells/units.

1.6 LSTM, Question 1.6

LSTM is implemented as instructed. I've used a efficient computation trick that we've seen in the NLP1 project, where we just initialize 2 weight matrices (i.e. linear layers), one for weights with the input dimension W_x, and the other for the ones with hidden layer dimensionW_h. Then these matrices are used as a whole in one matrix multiplication in the forward pass, then the result is split into 4 for each, giving as e.g. $W_{gx}, W_{ix}, W_{fx}, W_{ox}$. In the forward pass we simply return the output $p(t)$.

It was advised in the assignment to **adjust learning rate** as palindrome length increases. For this, I have tried different techniques. Firstly the simplest, I create a manual learning rate schedule across palindrome length, and initialize the optimizers of each palindrome length training with their own learning rate. This palindrome lr schedule increases a little for each length. The first time I tried this, I must have chosen values poorly since it didn't work for every length while some where already at perfect accuracy, so I got a zig-zag-gy plot. Hence I tried adding different stuff like an actual scheduler or manually changing learning rate during training based on loss. However eventually I found that with the right setting my initial method performs perfect for the lengths between 5-35, but would probably need to be adjusted for longer palindromes as after some point the learning rate becomes too high. So the final schedule across palindrome lengths are:

```
LR_SCHEDULE = [[0.01, 0.01] + 0.005*i for i in range 20][:35]
```

```
= [0.01 0.01 0.015 0.015 0.02 0.02 0.025 0.025 0.03 0.03 0.035 0.035 0.04
0.04 0.045 0.045 0.05 0.05 0.055 0.055 0.06 0.06 0.065 0.065 0.07 0.07
0.075 0.075 0.08 0.08 0.085 0.085 0.09 0.09 0.095]
```

Again, on figure 1, we can see that LSTM can maintain a (near) perfect accuracy, after a tiny parameter tweaking, regardless of the palindrome length it is faced with. This is due to the capability of LSTMs to **capture long term dependencies**, with RNNs can not do. This difference is clearly present in the plot, as RNNs performance gets worse or unreliable as palindrome length increases.

1.7 The Question hidden in the code

```
# QUESTION: what happens here and why?
torch.nn.utils.clip_grad_norm(model.parameters(), max_norm=config.max_norm)
```

This function is used to hinder very large, **exploding gradients**. It happens when the layers back-propagate a large loss, and each layer multiplies & amplifies this loss even more which means the gradients end up being too large as it approaches lower layers. Thus the lower layers get **huge weight updates** such that either the model is very unstable or doesn't learn at all. This is mostly a problem or deep or recurrent nns. This article [6] refers to some symptoms of this problem.

Apparently using LSTMs can be a solution to this problem, however with long sequences they may also end up with exploding gradients as well. So one of the solutions to this problem is using what was asked about, **gradient clipping**. The gradient magnitude is compared with a given threshold and clipped respectively if it exceeds the value. This should be done after back-propagation (`loss.backward()`) is done, and before we update the weights (`optimizer.step()`).

2 Recurrent Nets as Generative Model

2.1 Question 2.1

(a) Model initialization simply consists of the initialization of a two layer LSTM followed by a linear layer of size: hidden layer $\times$ vocabulary size (feature size).

(b) Well, by the time I'm writing this question, I already have temperature implemented, so instead of taking argmax (pure greedy), I will use a very tiny temperature $1e - 15$ so that it's very greedy. The code can reproduce samples for a lot of different temperature values given the model files.

The model is trained on **an Agatha Christie book**. To sum-up the training process, it can quickly get up to 40-50% accuracy then slowly achieves a final accuracy of around 60%, see figure 2. This whole process corresponds to $\approx$ 7 epochs and 40K steps. Since it gets to $\approx$ 40% it doesn't come as a surprise that below, in the first sample we already see kind-of coherent words being formed. But as we train more, and

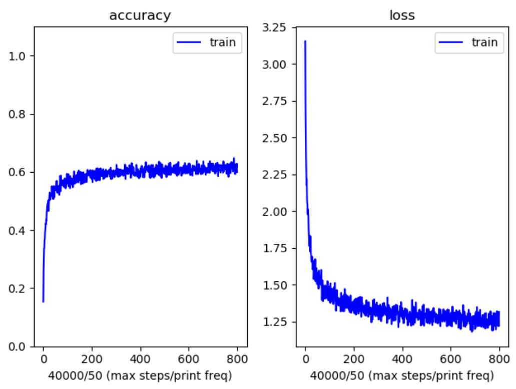

Figure 2: Part 2, training process for the Agatha Christie book.

improve the accuracy by a few percentages, we see that the sentence structure becomes more sensible, a specially the final sample can pass as an actual sentence if it were completed!

```
Step 08K/40K : "very shout a form of the last
Step 16K/40K : What the same with a strychnin
Step 24K/40K : Sou must be able to be a littl
Step 32K/40K : fe the coffee of the boudoir w
Step 40K/40K : Where we were suspected the st
```

(c) To sample from the model, we feed it a random character then from the probability distribution (i.e. softmax) of the output, we sample the next character, then now using this character and the hidden and cell states returned we can sample another, until we get the sequence length we want. In the actual *sampling* part, if we take the argmax of the softmax, we simply get the greedy choice.

To balance out the choice between greedy and random, we add the parameter of temperature. We multiply the output softmax distribution with β the reciprocal of temperature, and sample (multinomial) as usual after that. This multiplication means we are playing with how sharp the distribution peak is, meaning the larger β is (or smaller the temperature is) the more greedy (sharper) our choice is. So if we test out temperatures $[1e-15, 0.25, 0.5, 0.75, 1.0, 1.5, 2.0]$, this selection ranges from very greedy to quite random. See results for sampling mid-training for the Agatha Christie book and sequence length 30. Providing results for two different random seeds.

```
Temperature:  1e-15 :  ê the poison to my surprise, a
Temperature:  0.25 :  /ng on the table at Styles, si
Temperature:  0.5 :  '  looking to the early committ
Temperature:  0.75 :  ?t at the bench taken as the p
Temperature:  1.0 :  êto's myself.  This had been ac
Temperature:  1.5 :  (se of them.\n\nShe found ofd.''\n
Temperature:  2.0 :  4eRge\nols''-net, hauride their

Temperature:  1e-15 :  Shought that it was a strangle
Temperature:  0.25 :  ``at any other way.  There was a
Temperature:  0.5 :  * in the long window, and stry
Temperature:  0.75 :  Dilat spirchly at\nStyles, star
Temperature:  1.0 :  `try, it was consoled them.  Fi
Temperature:  1.5 :  #re of relum\nare\nactively\nriqu
Temperature:  2.0 :  me, a\npervoum.  What cas redelo
```

We see that a little randomness (0.25 to 0.75) gives a nice variety to the generation, although it might not seem as well structured as the most greedy sample ($1e-15$). But I also think that it might be because of the unfortunate first random character that they got, e.g. what would follow * in the book? What about a forward slash? The most random values (1.5 to 2) seem to get too random, although there's still quite a bit of sensible words in there.

2.2 BONUS Question 2.2

To complete sentences with our model, we take a sentence to complete, then feed the whole thing forward in one-hot encoding. We keep the hidden and cell state returned and use those and the final character of the input sequence to start of the generation as we did before to sample completely new sentences. The difference is like mentioned, now we are already starting with a known character and states. Since I made my model learn an Agatha Christie book I made up some related sentence beginnings, and also used the fairy-related one. These sentences are generated with temperature 0.25, and 70 new characters are generated.

Sleeping beauty isessing a man and she had been admittedly that it was a strychnine at h
Murderer was an excellent for the moment. It was a large aparting the the moment,
Poirot used to much out of the boudoir, and was a really in the boudoir. And the poi
Of course it is the lawn with a man was a small provision of the particular sharply.
His mustasche was a sharp and his stepmother's manner was not there is a solicited the
Oh my! Now, if it is a smast seemed to see his stepmother should be the hall
Poirot inspected theve to say that I had a man is a strychnine for the tragedy was not som
The crime scene iself. As I was some particular scres of Mr. Inglethorp, and that it was
He said "Detective! The strength of Mr. Inglethorp was now any other was at the time at t

As seen above, the sentences are pretty good and fun to read! I tried to choose some sentences that might trigger some sort of a crime related sentence or perhaps a conversation. The sentences seem rational in the sense that they just look like the product of bad English education.

I also saw that certain patterns can trigger specific things, e.g. a new line can trigger a Chapter heading, which makes sense. And when I was trying to generate this I got another nice sentence that I can't resist not including. This example shows that the model somewhat understands quoting, and that *exclamation* needing reactions may go together. It also seems to carry a tone of story telling. Then in the following one we kick it off with two newlines, which triggers a chapter beginning and name.

> He said "Detective!\n
> “Ah!” I said at his shoulder. I was a sharp in any other man who was n
>
> \n\n \n\n\n\n CHAPTER I. THE WARRATHE POSSIDED OF POY PoIR EJ THE TRECEDEN OR CI

Let's try longer one just for fun.

> He said "Detective!
> “You are not know it?”\n\n“Yes, sir, I should be a little mantelpiece of the mantelpiece. I was not a little ideas. And I had an idea of the post-mortem.

Finally, I tried the model on whats-app data, which works great as well, seems to capture the things we most talk about, or completes people's names as messages.

2.3 Question 2.2

3 GNNs

3.1 Question 3.1

(a) As explained, A contains a mapping between edges (adjacency information), so if $edge_1$ out of N has a edge going into $edge_2$, A_{12} would be one. We can draw directed, or not directed, any graph using this matrix, as it has the structure of the graph encoded in this way. By using this matrix and the embedding/features of the nodes we get $\hat{A}$. As we multiply the input activations with the adjacency matrix, we are weighting the activations by the adjacency feature captured in $\hat{A}$. Thus the edge and node information are captured by all activations and can be propagated through layers. But note that one layer (or one call of such a layer) would only posses direct edge relationships, i.e. first degree neighbours.

As you have more layers you will be propagated nested relationships of edges, or whatever information is stored on your graph, hence message passing can simply be done in this way, with however many layers you may need.

(b)

$$A \xrightarrow{hop1} B \xrightarrow{hop2} C \xrightarrow{hop3} D$$

$\hat{A}$ has both the feature and the adjacency information within itself. So let's say we are at the first node, we compute $\hat{A}$ using the features $\tilde{A}$ and the adjacency mapping A. When we compute the activation $H^{(l)}$ (assuming current layer was $l-1$ or 0) using $\hat{A}$ we know now that the embedding information of two 1^{st} degree neighbours are hidden within the activations. Now when we move onto the next layer, we are multiplying $\hat{A}$ with these activations $H^{(l)}$, meaning on top of the hidden neighbour information we propagated from the last layer, we are adding another layer of 1^{st} degree neighbour info, which ends up covering 2 hop neighbours. To give an example, above the first layer captures A to B, the second adds B to C on top of the activation of A to B, meaning now the activation contains both hops 1 and 2. So for propagating information of all 3 hops, **you would need 3 layers**.

3.2 Question 3.2 - Real World Applications

- Physical/Physics systems (**interaction networks**): Inspired from how humans think about the real world in through modelling objects and their interactions as a graph. Can be used to reason/make predictions about interactions.
- **Image** classification: as explained in the following question, the neighbouring relations of pixels can be modeled and learned as a graph. This seems to be especially appealing in zero-shot learning.
- **Text** (e.g. **sentiment classification**), where we have the luxury of using annotated word-tree's which model the function and hierarchy of words.
- Others. Combinatorial Optimization, Graph Generation, Machine Translation, Visual Question Answering etc.

Link to helper sources: article [1], paper [2] with a list of numerous applications.

3.3 Question 3.3

(a) If you have a graph or tree structured data I would expect GNNs to perform better, as they would be able to learn from the structure. For instance if you have text data, and the data is annotated such that the **parse-tree** is provided, a tree-structured model, e.g. GNN should perform better. If you simply use a RNN and learn the sentences as just sequences you are not learning different hierarchical relationships the words have with each other, hence not the true semantics of the data. One could argue that the tree structure could be Incorporated into a sequence form by some ordering, and feature adding but then this would be much harder to do if we have more complex sentences, or to be general more complex graphs with loops and (bi)directional edges etc. To sum, GNNs would be more expressive if we have **inherently graph structured data**.

If we have **sequential** data like a **time-series**, or you will make predictions based on the sequential behavior such that you would have to make an additional effort to put data into a graph-like form, then the data is not really inherently graph-like and capturing such a relationship wouldn't be any help, could even cause model to learn false semantics. For instance, speech or audio data would be more of an RNN-expertise, or future predictions based on time-series data etc.

(b) First thing that comes to my mind is wrapping an RNN cell in a GNN cell, to elaborate, basically incorporating the adjacency matrix into every activation in an RNN. Let's peak into what people have already tried.

I see that people have clearly attempted this. In this paper [5], Seo et al. propose GCRNs, which combine GCNs and any RNN type network into one. They propose two models, where in the first they simply stack a GCN and an LSTM, and in the second they generalize a convLSTM to graphs by replacing euclidean convolutions with graph convolutions.

References

[1] Medium article, Applications of Graph Neural Networks, link

[2] Graph Neural Networks: A Review of Methods and Applications, Zhou et al. 2018, arXiv:1812.08434 [cs.LG].

[3] Medium article, Understanding RMSprop - faster neural network learning, link

[4] Types of Optimization Algorithms used in Neural Networks and Ways to Optimize Gradient Descent, link.

[5] Structured sequence modeling with graph convolutional recurrent networks. Y Seo, M Defferrard, P Vandergheynst, X Bresson - International Conference on Neural Information, 2018

[6] Article, A Gentle Introduction to Exploding Gradients in Neural Networks, link