Deep Learning Assignment 2 - RNNs and Graph Networks

Balint Hompot
12746452
Artificial Intelligence Msc.
University of Amsterdam
Nieuwe Achtergracht 166, 1018 WV Amsterdam
balint.hompot@student.uva.nl

November 29, 2019

1 Vanilla RNN vs LSTM

1.1 Gradients

Let us start with the gradient w.r.t. the output weights. We calculated the softmax with cross-entropy derivative in the first assignment, so I will use at as given there. $\ominus$ stands for elementwise minus and $\otimes$ for tensor product. First, let's express the gradient for the final time step:

$$\frac{\partial L^T}{\partial W_{ph}} = \frac{\partial L^T}{\partial p^T}\frac{\partial p^T}{\partial W_{ph}} = (y \ominus softmax(p^T)) \otimes h^T$$

We can see that this only depends on the output and the hidden activity at the given time step, so, assuming that we don't only use the output weights at the last time step (as some implementations do), we can get the full gradient by simply summing the gradients at the time steps:

$$\frac{\partial L}{\partial W_{ph}} = \Sigma_{t=1}^{T}(y \ominus softmax(p^t)) \otimes h^t$$

For the hidden-to-hidden layer we have:

$$\frac{\partial L^T}{\partial W_{hh}} = \frac{\partial L^T}{\partial p^T}\frac{\partial p^T}{\partial h^T}\frac{\partial h^T}{\partial W_{hh}}$$

Note that this is not complete, as the hidden activity depends on the previous hidden activity, so the partial of hidden activity w.r.t. the weights also depend on the previous hidden activity. For that we need to incorporate the gradient with respect to all the previous layers, by summing up the gradient up to all possible depths:

$$\frac{\partial L^T}{\partial W_{hh}} = \Sigma_{k=0}^{T} \frac{\partial L^T}{\partial p^T} \frac{\partial p^T}{\partial h^T} \frac{\partial h^T}{\partial h^k} \frac{\partial h^k}{\partial W_{hh}}$$

The partial derivative of the hidden activity w.r.t. some previous hidden activity is also recursive, as any hidden activity depends directly on the previous one. So the full expression is:

$$\frac{\partial L^T}{\partial W_{hh}} = \Sigma_{k=0}^{T} \frac{\partial L^T}{\partial p^T} \frac{\partial p^T}{\partial h^T} \Pi_{j=k+1}^{T} \frac{\partial h^j}{\partial h^{j-1}} \frac{\partial h^k}{\partial W_{hh}}$$

Now we need to express these partials in terms of quantities in the original equations. The first fraction is the same as in the case of the hidden weights. The second partial is:

$$\frac{\partial p^t}{\partial h^t} = W_{ph}$$

Then we have (using $tanh(x)' = 1 - tanh^2(x)$):

$$\Pi_{j=k+1}^{T} \frac{\partial h^j}{\partial h^{j-1}} = \Pi_{j=k+1}^{T} (1 - h^{j^2}) * W_{hh}$$

And for the final term:

$$\frac{\partial h^k}{\partial W_{hh}} = (1 - h^{k^2}) * h^{k-1}$$

So the full expression is:

$$\frac{\partial L^T}{\partial W_{hh}} = \Sigma_{k=0}^{T} \frac{\partial L^T}{\partial p^T} \frac{\partial p^T}{\partial h^T} \Pi_{j=k+1}^{T} \frac{\partial h^j}{\partial h^{j-1}} \frac{\partial h^k}{\partial W_{hh}} =$$
$$(y \ominus softmax(p^T)) \otimes h^T * W_{ph} * \Sigma_{k=1}^{T} \Pi_{j=k+1}^{T} (1 - h^{j^2}) * W_{hh} * (1 - h^{k^2}) * h^{k-1}$$

We can see that the main difference between the two derivatives is that the second one is recursive, its value depends on the activities at all time steps. This can give to several problems potentially, if we sequence length gets big: as the gradient depends on the recursive product of activities, there is a chance that the gradient vanishes by multiplying several times with a quantity close to 0, or, there is a chance that the gradients would explode, by multiplying many times by a factor with a norm bigger than 1. An other problem, if we calculate the outputs and the gradients at every time step is that we update the weights, but the hidden state that is passed to the next time step was calculated with the old weights, so the gradients at the next time step depend on the old weights, but update the new ones. This is many time a good approximation, but as the sequence gets longer, the inconsistencies stack up, making it harder to train.

1.2 RNN implementation

The model in vanilla_rnn.py implements a batch method for training a simple RNN (with the time steps being equal in a batch) with a set of input to hidden a set of hidden to hidden parameters, and a set of hidden to output parameters. All these implement bias as well. The output is only calculated and matched at the last time step (sequence-to-one implementation). In the forward step, we loop through the time steps, get the corresponding inputs, calculate the hidden activity by a matrix multiplication on the input and the previous hidden activity (which is set to ones in the beginning, so there only the input counts) and the corresponding biases. The output is calculated similarly at the end, no sigmoid is explicitly used, as it is included in the cross-entropy loss that we use for training.

1.3 RNN experiments

The implementation can be found in the vanilla_rnn file. For weight initialization, I used a normal distribution around 0 with 0.5 standard deviation to avoid getting stuck in the beginning. The mapping is performed as described, using one set of weights for input to hidden, one set for hidden to next hidden, and a separate for hidden to output layer. I run 4 experiment with increasing length of 5, 9, 13 and 17, using the default parameters. The results can be seen on figures 1, 2, 3, 4, respectively (note that in all of these graphs, step refers to training step). Note that instead of plotting the accuracies as a function of length, I chose to create separate plots for the different training processes. The reason is, that at the end of the training with such high number of training step the accuracies can be quite similar, and it is more informative to look at the training process as a whole, through training steps, but plotting all processes with different lengths would have been confusing with the rugged training curves. The accuracies refer to the batch accuracy at time step.

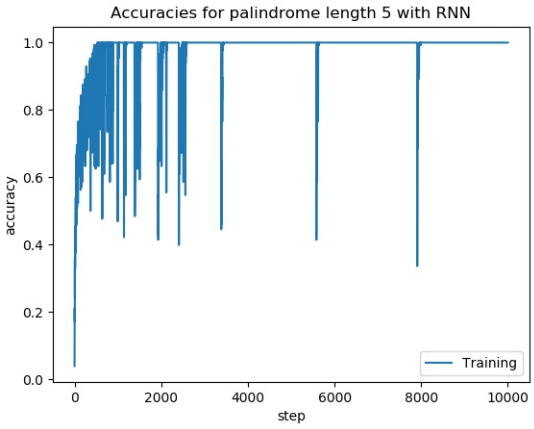

Figure 1: "RNN accuracy on palindrome length 5 with default parameters"

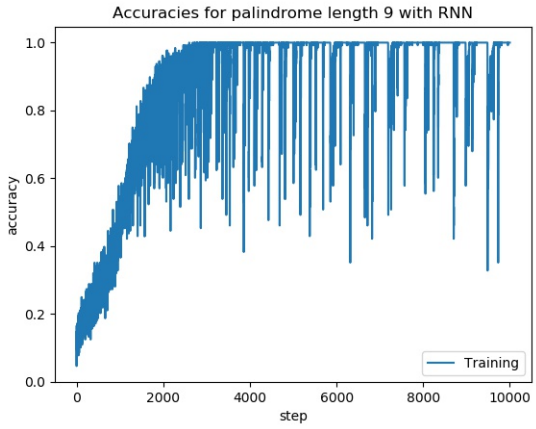

Figure 2: "RNN accuracy on palindrome length 9 with default parameters"

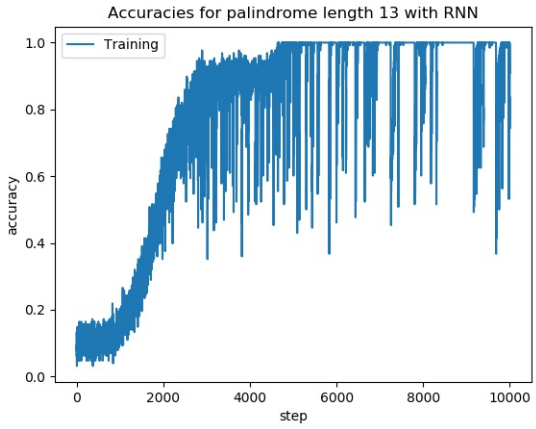

Figure 3: "RNN accuracy on palindrome length 13 with default parameters"

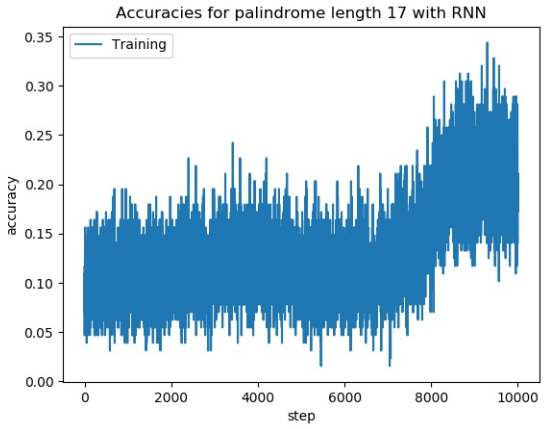

Figure 4: "RNN accuracy on palindrome length 17 with default parameters"

Wee can see, that for short sequences the RNN converges to perfect output quickly, and only drops from there rarely, but it gets harder as the seqeunce gets longer, and the mistakes become more frequent. Even for length 17, there is some learning at the end of the training, but the overall performance is poor, and it is safe to assume, that the model would be of no use for even longer sequences.

1.4 Learning optimization

A common problem regarding the training of neural networks is that in such convex state spaces, stepping to find a minimum (that is, SGD) often drives us into local minima, or plateaus or saddle points (points where the gradient is, so simple SGD is trapped). For solving these issues, in practice we mostly use adaptive learning rates and/or momentum.
The idea of adaptive learning rate comes the observation, that on non-steep surfaces the learning is slow even if the direction is consistently good. So in the first solutions researchers introduced methods, that increase the learning rates, when the direction is consistent with the previous steps, and decrease, when the step was too big, so the direction becomes inconsistent with the previous directions. In batch methods, the direction is somewhat stochastic, so the described method does not work well, so we rather modulate at each time step the learning rate by a moving average of gradients, using the previous time steps. The idea is the same: the size of step should be modulated by how big our errors were in the previous steps. This method is called RMSprop, and it is very helpful in speeding up learning around plateaus and saddle points.
Momentum on the other hand is most important in avoiding local minima. It can be explained in an intuitive way: if we go down a hill we get trapped easily if we take one step at a time only to the good direction, but if we run, and use the momentum of the previous steps, we just run through the small pits on the

way. The momentum terms implement this idea: the weight update does not only depend on the current gradient, but also on the previous one(s) multiplied by some hyperparameter.
Most current popular optimization algorithms, like Adam implements both adaptive weights and momentum.

1.5 LSTM discussion

1.5.1 gates

An LSTM runs a state through a sequence containing information about the sequence as a whole, which modulates it's hidden state and output at any time step. The first gate is a forgetting gate, which basically decides (based on the current input, and the incoming hidden activation - this is true for all the gates, so I won't mention at the next ones) what information should be dropped from the state. The activation here is sigmoid, which maps the values between 0 and 1 with a steep step: this was chose as we multiply the state by this, and the values of 0 and 1 correspond to dropping this information, or keeping the information (with the values in-between for partial keep). We can imagine this as a non strictly binary true (1) or false (0) gate, where we ask the question, 'do I still need this information based on the incoming facts'?
The input at a time step passes through the input modulation gate. This is not a filter like the first one, it is rather a usual linear-aggregation with non-linearity function for better separability and learning, thus it uses tanh activation which is similar to sigmoid, but spans from -1 to 1. This is a good choice, as it can learn flexible non-linear feature mapping, but is still centered around 0 which helps with propagating the error in a longer sequence.
The input is then multiplied with the input gate values, using a sigmoid activation. The choice of values from 0 to 1 is similar to the forgetting gate, but the question now: 'based on the incoming data and hidden state, do I need to store this piece of information?', where the dimensions close to 0 will not really change the state, and the dimensions close to 1 have a great effect on the state.
Lastly, we have the output gate =, which is again a soft-binary gate, where we ask 'do i need to pass to the dimension value to the next, based on the incoming data?', as this output in most implementation is used as the hidden activation in the next time step. As such, this again uses sigmoid activation, note however, that the output goes through tanh nonlinearity for better learning, though that is not a gating mechanism.

1.5.2 parameters

The number of parameters is independent of the batch size or the time steps, as we reuse the parameters at each step, for each sequence. So for each of the 4 gates discussed above, we have to do a mapping from input (d) to hidden (n) and hidden (at t-1) to hidden (at t), which gives us 4*(n*d) parameters. As we also have biases for each gate, in total we have 4*(d*n+n) parameters, assuming that

the final output is just a hidden state and not a separate mapping. (Otherwise, we also have n*o+o params for the output mapping with output dimensions of o in the simple case, although here many more classification layers can be applies with their own parameters.)

1.6 LSTM implementation

The LSTM implementation is similar to the RNN implementation (also in weight initialization), but instead of one mapping, we perform 4 non-linear mapping (the gates, as discussed), and we do not only run the hidden state through time, but also the cell state. In this implementation, there is also an output mapping at the end of the sequence, which has it's own weights and biases, and takes the last hidden activity as input (this is not necessarily the best solution, as the later steps have more weight implicitly, but the cell state, that influences the hidden state should capture all the important aspects of the whole sequence). I tested the LSTM on the same 4 settings as the RNN. It's performance on sequence length of 5 can be seen on figure 5.

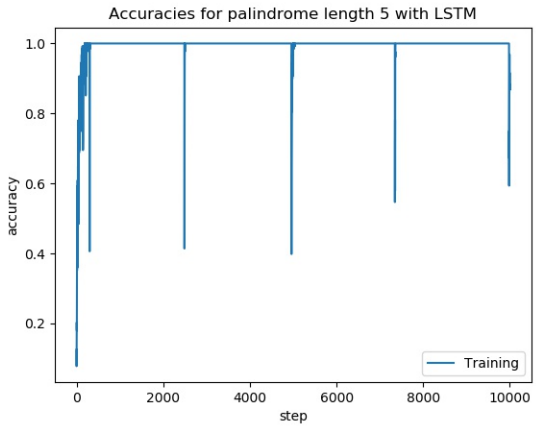

Figure 5: "LSTM accuracy on palindrome length 5 with default parameters"

The LSTM gets to a perfect score very fast, and only drops below that very rarely (even though these stand out on the graph) On 9 can be seen on figure 6.

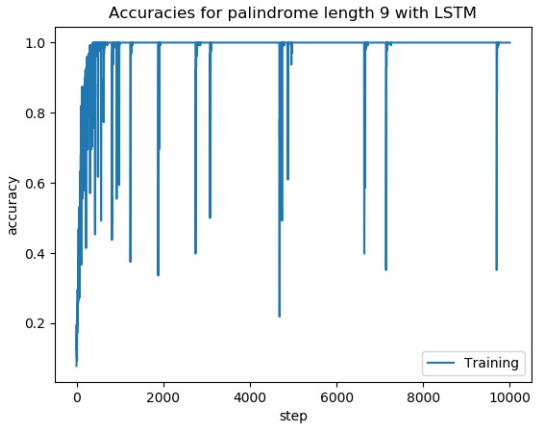

Figure 6: "LSTM accuracy on palindrome length 9 with default parameters"

On 13 can be seen on figure 7.

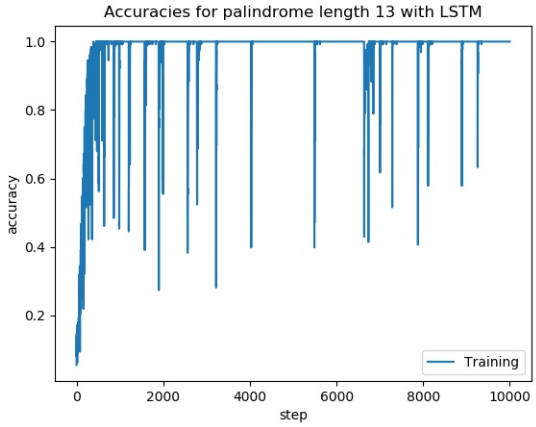

Figure 7: "LSTM accuracy on palindrome length 13 with default parameters"

On 17 can be seen on figure 8.

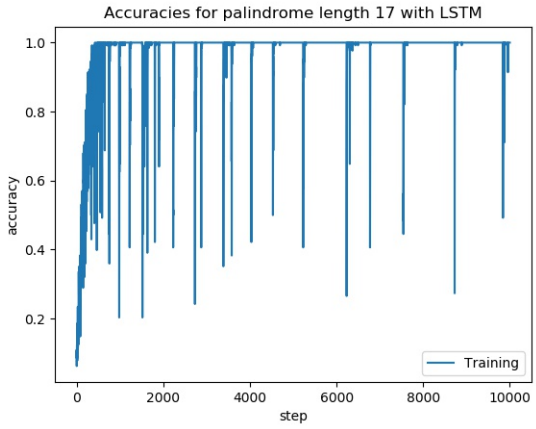

Figure 8: "LSTM accuracy on palindrome length 17 with default parameters"

We can see that the model is capable of generalizing much faster than the vanilla RNN on the same length, and handles longer dependencies much better thanks to the running cell state and the gating mechanisms. We can observe, that on these length, the LSTM gets to near perfect score easily, although it is visible, that with increasing the length, the convergence becomes slower, and the occasional drops for tricky inputs become more frequent. I even tested the architecture for sequence length of 40, and the result is still impressive, however visible worse than for the previous examples, as seen on figure 9.

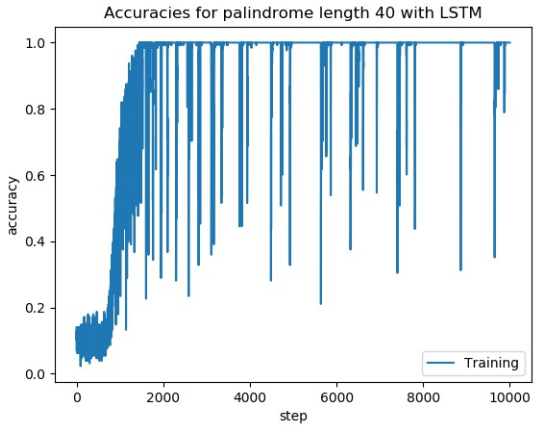

Figure 9: "LSTM accuracy on palindrome length 40 with default parameters"

1.7 Gradients over time

The script for checking the gradients is basically a modified version of the original train script. Each of the two models have a parameter store_hidden, which, when true, stores the hidden activity in an array with requiring gradients. The script modifies the original train such that it sets this parameter true, and then stores the gradients for these hidden activities. (An upside is, that the gradients can be checked after any number of train steps, we should just set the number of steps as usual. The default, as required, is after just one step). After retrieving the gradients, the magnitudes are calculated for each time step and plotted. For RNN, the initial state is shown on figure 10, for LSTM, on figure 11.

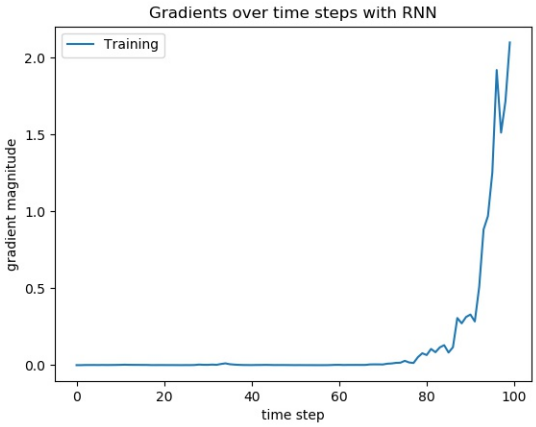

Figure 10: "Gradients over time before training - RNN"

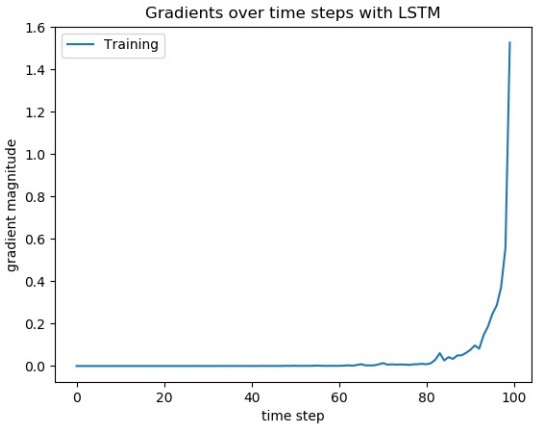

Figure 11: "Gradients over time before training - LSTM"

We can see here, that the RNN gradients are passed somewhat further on to the earlier time steps. This is probably due to the fact, that this is the untrained (randomly initialized) state, and the LSTM has much more parameters with small random values, so the gradient is more likely to vanish over time. However, as we start to train the model, the LSTM with the gating mechanisms is more fit to extract long term dependency, so the ratio should change after some learning. This is what we can observe on figure 12 and 13, which were plotted after performing 1000 training steps with default parameters. The difference is subtle, but the RNN gradients approach 0 sooner, and more abruptly, while LSTM gradients vanish more smoothly.

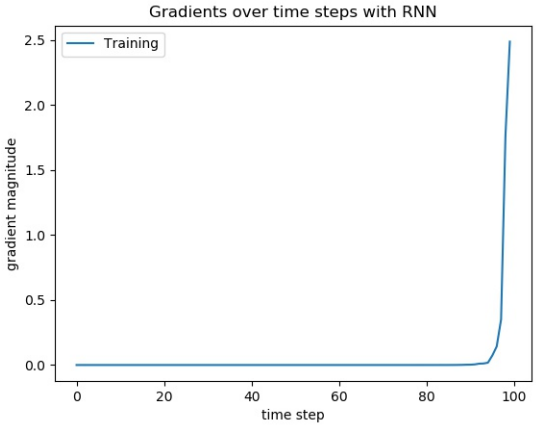

Figure 12: "Gradients over time after 1000 training steps - RNN"

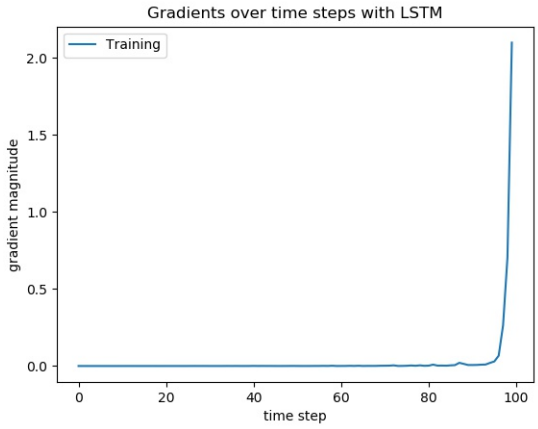

Figure 13: "Gradients over time after 1000 training steps - LSTM"

2 Generative RNN-s

2.1 PyTorch LSTM implementation

The pytorch implementation is based on the LSTM module of the library. This implements the 4 gates as discussed above, but unlike the manual implementation, it does not have an output calculation layer, but concatenates the hidden activity at each time step to create an output. Thus, the LSTM module takes the input and hidden size as parameters, but not the output dimensionality. As the input is the one-hot encoding of the character, the input dimensionality is the alphabet size (or vocab size as it is called). To generate the correct output -which should again be size of the alphabet, as we output probabilities for letters at a time - i applied a multilayer perceptron with ReLU nonlinearity to reduce the size of the hidden activity to alphabet size, with some learnable classification parameters. For initialization I used the hyperparamteres batch_size for the size of the training batch, seq_len to define the sequence length, and vocab_size to define the number of different possible characters, thus the dimensions of input and output. Besides, I added paramters num_layers and num_hidden with values 2 and 256, as defined in the assignment, and a temperature parameter that will be discussed later. Through the training, I also print out generated outputs at some time steps to see the progress as defined in the config. In the implementation, I used 2 LSTM layers stacked, the performance on the Grimm training data with the default parameters can be seen on figure 14 (note that the number of training steps is upper bounded by the length of the input, for Grimm it's 8440).

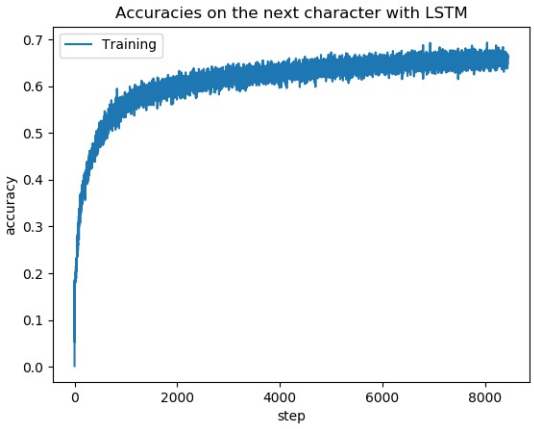

Figure 14: "LSTM accuracy on the Grimm tales with default parameters"

We can see that the performance approaches around 65 (quite rapidly) and seems to be still growing%. This result is very good, although not as high as for the palindromes, due to the high variability and complexity of real language, and the bigger number of potential outputs (10 vs 87), but it is way above chance level (which would be 1.149% with 87 characters).

2.2 Text generation on input

The text generation function can be found in the genText.py file and is called after the training is finished (given the generate_text flag is true). One change I made is that it prompts the user for the starting input (pay attention to this if you run on the cluster) and accepts strings of any length, not just single letters. The function converts the input into a sequence of one-hot encoding and runs a forward pass of the model with that. From the output, it extracts the next character based on the 'stochastic' parameter: if false, we simply select the maximum probability character, if true, we sample from the possible characters based on their probabilities. This selected character is the printed out, and appended to the input so in the next iteration the new character is generated knowing the previous one. The length of the generated sequence (so the number of iterations) is also given as a parameter. Here are examples of the generated texts after 2000, 4000, 6000 and 8000 and 8840 (the maximum) training steps, in this order:

with the words to the work and

the wood where the mother said

the wood and said: 'I will not

the king to himself: 'If I coul

the prize. 'You must take me an

We can observe, that in the beginning, only the most frequent words are extracted, and the spelling mistakes are more frequent (not in these examples), and later the learned vocabulary gets bigger, and the spelling mistakes become rare. The tale-like structures, such as quotations and dialogues also start to get generated. Although, the improvement is most obvious in the beginning, as the training is faster early in the process/
For most of my experiments however, I used length of 300. 30 characters is generally too short to tell how comprehensive a text is, and it is easier to look at the process on this length. An example generated Grimm-tale of length 300 characters with greedy sampling at each step looks as follows (starting with t):

the children were seen the poor man and the first time the first twig, 'I will give you the wild bear it in the window, and the wild man said: 'I will give you my head that I had not go to the table with the three servant to come to the town with the water to the castle where to be pope? His master w

Not surprisingly - as the learning sequence length was 30 characters - the model cannot generate consistent text over this length, and this would be the case for length over 30 in general, as the network is trained to extract dependencies on this term. Still, it shows good syntactical and stylistical ability, and sometimes coherent use of names and objects for a part. For sequences shorter than 30 characters, the model generates really comprehensive text, as the short dependencies are easier to hold and generate, however, these are too short to be of any use in most of the text generation tasks.

2.3 Sampling with temperature

The temperature is basically a constant factor to modulate the logits with. This does not change where the max of the output is, but it does change the distribution: if it is very small (close to 0), the logits approach 0 and become similar, so the distribution of probabilities (after softmax) is more flat. On the other hand, if it is large, it magnifies the differences between the logits, so the probability distribtion becomes close to a one-hot encoding of the max probability. This distribution difference is important if we use stochastic sampling (as discussed above): we select the next character based on their probabilities, not just the max at all steps. This also means, that with low temperature the distribution is flat, so we select more varied sequences, but we also make more mistakes, whereas with high temperature, we are getting close to max-sampling, meaning low amount of errors, but small variability. The code passes the temperature flag to the model as parameter, so the output is modulated with that, and we also need to set the text generation function to stochastic. Using character length of 300 for more informative output, with 0.5 temperature we generate tales like:

ter–it is such a large eyes and said: 'My hoes and willing, a very cruel where it your gooden to eat.' The young man began to flog by being to go. The king Hengeresses soon from tried, and strooper to drink in a kinddom: somewered: 'Just go for every time afraid of it, and he lived to be lided. The

With 1 (which is a factor of 1, so it's like not using temperature, but we are still using stochastic sampling, so this method is not the same as in the previous section) we get:

tter again.' The witch said he may nothing before the bear had here, through this and higher is accessed herself in the ground. Come, wheels, he knocked in these he gave it me for thy teams, dales, you will never combid to be true the Sun?s' replied she, 'and they had some copyoth exactly theref

With 2 an example is:

that he heard the cooking, whith his.' Huntsmen set up there, and a horse is this three dropped it off, when he heard a thing!' Hansel gathered the table-copy with the prince thought his twelve sime she given him. The day came all eat ying out on the roof, and it was quite innocent, you are shut out

These examples are in line with the analysis: with small temperature, we get more varied output with rare words, such as 'flog', but we also get more syntactic and spelling errors. With high temperature, the text gets less varied, but the number of mistakes is lower, and we get closer to the output of the previous section.

3 Graph Neural Networks

3.1 Graph Convolutional Network

3.1.1 description

Convolutional networks gained their popularity due to the ability of exploiting local connectivity: the idea, that e.g. in an image we don't have to look at full connectivity to extract patterns, but local patches can carry much information. The idea is the same for GCN-s: a local neighbourhood of graph vertices can carry much information about their nature. However a problem is, that the vertex neighbourhoods vary in size (unlike image patches). In order to fix that, we can represent an edge by a vector which denotes which other vectors it is connected to, with the dimensionality equal to the number of vertices (or we can learn a dense representation with e.g. a skip gram). Besides that, the vertex has to be added to it's own representation so the neighbourhood is complete (equation 16) and normalized by it's degree (eq. 15). In the end we have a matrix of representations of vertices in fixed size with local information, which can be a subject to regular mapping: it is multiplied by the incoming information,

then the learnable weights, and a non-linearity is applied on the top. Since this adjacency matrix is the same across layers, and the indices correspond to actual vertices, a multiplication by this matrix matrix can be seen as passing the message encoded in the other matrix (that is the incoming hidden activity) through the graph: the adjacent vertices, which have non-zero value receive the information of the multiplicand. This becomes the input of the next layer, so a stack of layers act as passing the information through the net. The passed message is modulated though by the normalization, the learnable weights and the nonlinearity.

3.1.2 drawbacks

In the discussed, simple form of GCN we only look at connectivity between edges. This can be potentially a problem, as many graphs contain important information inside the edges, e.g. about strength of connection, bandwidth, direction etc. A simple solution for this could to incorporate this information in the adjacency matrix: for weighted edges, we can simply weigh the matrix dimensions, and for directed edges, we only add the connection to the starting vertex (thus the matrix becomes non-symmetrical)

3.2 Graph operations

3.2.1 adjacency matrix

The non-normalized adjacency matrix encodes which vertex is neighboured by which, where each column vector correspond to one vertex, and each row correspond to a binary showing ig the vertex is connected to the one at the row's index. As eq. 16 shows, each vertex should be connected to itself.

$$\begin{matrix} 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 & 1 \end{matrix}$$

3.2.2 number of operations

As discussed above, one update step corresponds to passing the modulated information to the adjacent vertices (and itself), so the number of steps for the information in C to reach E is equal to the shortest distance between them, which is 3.

3.3 Applications

Graph networks can be applied to any classification or prediction problem where the inherent structure of information is stored in graphs. Based on towards data science blog-post [1] and some intuition, some of the most suitable fields include:

- Chemistry: the bonds between atoms and ions provide graph-like structures, and with GNN-s one can predict properties of structures, that can be useful in e.g. pharmaceutical research [2].
- Optimization: road maps and similar structures are often represented as graphs, where the edges are the roads. GNN-s provide a new technique to solve e.g. shortest path problems on these [3].
- Image classification: GNN-s extend the CNN-based image classification by generalizing for unseen classes, using graph representations of semantic connections of (both seen and unseen) classes. The trained CNN with the semantic graph and a GNN can label correctly classes, that were never seen by the CNN before [4].
- Text categorization: the citations in papers, or the links in web documents help us to construct graph representations of how these texts are related to each other. With GNN-s, we can categorize these texts based on their connectivity [6].

3.4 GNN-s and RNN-s

3.4.1 comparison

RNN-s take one step of an input sequence at a time, and work on it in combination with the processed information accumulated from the previous time steps. In a sense, a time sequence data can be seen as a (directed) graph where each time step is connected only to the next one. With this analogy, RNN-s can be seen as special GNN-s, where the length of the sequence is analogous of the layer depth of the GNN, as in one pass, we propagate information to the neighbouring nodes in GNN, that is, the next time step in RNN. There are however important differences in the two architectures: in sequnces, the direct connection between the nodes is limited to the next one, while in GNN-s we can hand-craft any sort of connectivity and still use the method (while the indirect information transition is learnable in both). This also defines the applicability: RNN-s work best for time series data, where the connection is known to be sequential, while GNN-s are applied when the structure of the data is known with variable local connectivity, and these variations have to be included, e.g. when analyzing text references, website links, groups of people etc. The data for RNN can then be represented as time series with fixed dimensions, and the data for GNN is rather a set of vertices, and their (variable sized) edges.
On the other hand, RNN-s are more flexible in update-pass length: the same architecture can work with any sequence length, just has to unroll at the end, while GNN-s are more like classical feedforward nets, where the architecture depth is fixed as a hyperparameter, so in the analogy, we predefine how many time we propagate information to the neighbours. This makes RNN-s more applicable for sequences of variable length (e.g. EEG signal analysis).
An other important difference is, that, as mentioned, GNN-s behave like feed-forward nets, so they also produce fixed size outputs, which can make them

useful in classification and prediction tasks (as discussed in 3.3), or they can be also seen as unified representation of a variable size graph. RNN-s on the other hand can produce output at any time step, making them applicable for sequence-to-sequence tasks, such as machine translation.

3.4.2 combination

From the above discussion, it can be seen how the two can work in combination: as GNN are mostly useful in graph analysis, while RNN-s in time series data, the combination can be used to analyse how graph structures unfold in time. The GNN controls the input and generates a unified representation for the RNN, and the RNN works on these inputs to reveal the patterns in time. This can be particularly useful for analyzing e.g. the course of interaction of chemicals (that hold graph-like structures), how the trends in website linking look, and many other areas, where the connection inside the data changes over time. An example architecture for such problems has been proposed this year [5].

References

[1] *Applications of Graph Neural Networks*. `https://towardsdatascience.com/https-medium-com-aishwaryajadhav-applications-of-graph-neural-networks-1420576be574`. Accessed: 2019-11-28.

[2] Connor W. Coley et al. "A graph-convolutional neural network model for the prediction of chemical reactivity". In: *Chem. Sci.* 10 (2 2019), pp. 370–377. DOI: `10.1039/C8SC04228D`. URL: `http://dx.doi.org/10.1039/C8SC04228D`.

[3] Chaitanya Joshi, Thomas Laurent, and Xavier Bresson. "An Efficient Graph Convolutional Network Technique for the Travelling Salesman Problem". In: June 2019.

[4] Michael Kampffmeyer et al. "Rethinking Knowledge Graph Propagation for Zero-Shot Learning". In: *CVPR*. 2018.

[5] Luana Ruiz, Fernando Gama, and Alejandro Ribeiro. "Gated Graph Convolutional Recurrent Neural Networks". In: Mar. 2019.

[6] Liang Yao, Chengsheng Mao, and Yuan Luo. "Graph Convolutional Networks for Text Classification". In: Sept. 2018.