Deep Learning - Assignment 2

Davide Belli
11887532
University of Amsterdam
davide.belli@student.uva.nl

1 Vanilla RNN versus LSTM

1.1

$$\begin{aligned}\frac{\partial L^{(T)}}{\partial W_{ph}} &= \frac{\partial L^{(T)}}{\partial \hat{y}^{(T)}}\frac{\partial \hat{y}^{(T)}}{\partial p^{(T)}}\frac{\partial p^{(T)}}{\partial W_{ph}} \\ &= -\frac{y^{(T)}}{\hat{y}^{(T)}}\Big(diag(\hat{y}^{(T)}) - \hat{y}^{(T)}\hat{y}^{(T)T}\Big)h^{(T)} \\ &= (\hat{y}^{(T)} - y^{(T)})h^{(T)T}\end{aligned} \tag{1}$$

Where the last passage combines the Softmax and the Cross Entropy Loss derivatives by noticing that $\sum_i (y_i \hat{y} - y) = \hat{y} - y$ because the target vector y is in one hot encoding.

$$\begin{aligned}\frac{\partial L^{(T)}}{\partial W_{hh}} &= \frac{\partial L^{(T)}}{\partial \hat{y}^{(T)}}\frac{\partial \hat{y}^{(T)}}{\partial p^{(T)}}\frac{\partial p^{(T)}}{\partial h^{(T)}}\frac{\partial h^{(T)}}{\partial W_{hh}} \\ &= \sum_{k=0}^{T}\frac{\partial L^{(T)}}{\partial \hat{y}^{(T)}}\frac{\partial \hat{y}^{(T)}}{\partial p^{(T)}}\frac{\partial p^{(T)}}{\partial h^{(T)}}\frac{\partial h^{(T)}}{\partial h^{(k)}}\frac{\partial h^{(k)}}{\partial W_{hh}} \\ &= \frac{\partial L^{(T)}}{\partial \hat{y}^{(T)}}\frac{\partial \hat{y}^{(T)}}{\partial p^{(T)}}\frac{\partial p^{(T)}}{\partial h^{(T)}}\left(\sum_{k=0}^{T}\frac{\partial h^{(T)}}{\partial h^{(k)}}\frac{\partial h^{(k)}}{\partial W_{hh}}\right) \\ &= \frac{\partial L^{(T)}}{\partial \hat{y}^{(T)}}\frac{\partial \hat{y}^{(T)}}{\partial p^{(T)}}\frac{\partial p^{(T)}}{\partial h^{(T)}}\left(\sum_{k=0}^{T}\Big(\prod_{j=T}^{k+1}\frac{\partial h^{(j)}}{\partial h^{(j-1)}}\Big)\frac{\partial h^{(k)}}{\partial \tilde{h}^{(k)}}\frac{\partial \tilde{h}^{(k)}}{\partial W_{hh}}\right) \\ &= (\hat{y}^{(T)} - y^{(T)})W_{ph}\left(\sum_{k=0}^{T}\Big(\prod_{j=T}^{k+1}\Big(I - \tanh^2\left(diag(\tilde{h}^{(j)})\right)\Big)W_{hh}\Big)\Big(I - \tanh^2\left(diag(\tilde{h}^{(k)})\right)\Big)h^{(k-1)}\right) \\ &= (\hat{y}^{(T)} - y^{(T)})W_{ph}\left(\sum_{k=0}^{T}\Big(\prod_{j=T}^{k+1}\Big(I - diag^2(h^{(j)})\Big)W_{hh}\Big)(I - diag^2(h^{(k)}))h^{(k-1)}\right)\end{aligned} \tag{2}$$

with $\tilde{h}^{(t)} = W_{hx}x^{(t)} + W_{hh}h^{(t-1)} + b_h$
and $h^{(t)} = \tanh(\tilde{h}^{(t)})$

The difference between the two gradients is that the first is only dependent on the last layer in the sequence of LSTM cell in the unrolled model, while the latter is defined recursively over time due to the term $\sum_{k=0}^{T}\frac{\partial h^{(T)}}{\partial h^{(k)}}\frac{\partial h^{(k)}}{\partial W_{hh}}$. As a consequence, the latter gradient can be very unstable numerically, due to the long sequence of products in the partial derivative with respect to the hidden

32nd Conference on Neural Information Processing Systems (NIPS 2018), Montréal, Canada.

states corresponding to each element in the time sequence. This can result in two different behavior, namely the exploding gradient and vanishing gradient problems. As discussed during the lectures, the training of the network is very likely to diverge (in the first case) or to converge to a non-optima solution too early. To solve the exploding gradient problem in the RNN case, an option is to introduce the gradient clipping or rescaling, in order to keep the same direction of the partial derivatives but reducing the vector magnitude when it overpasses a certain threshold.

1.2

See python code.

1.3

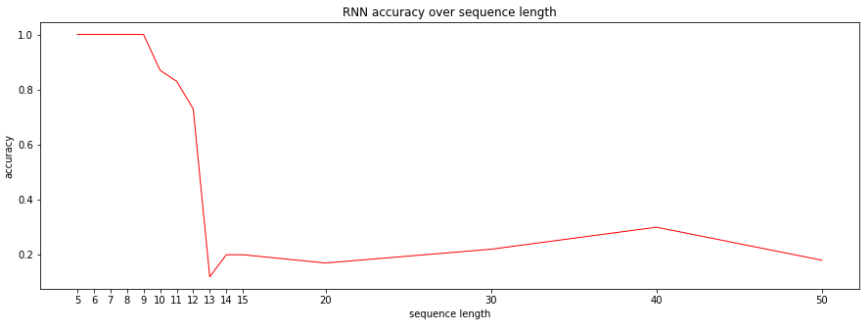

Figure 1: Accuracies of RNN over different sequence lengths.

In Fig. 1 we show how the RNN behaves when increasing the sequence length for input palindromes. In this experiment, we fix the training steps to $10,000$ and the learning rate to 0.001. We tried to run this experiment varying the batch size from 128 to 512 without seeing significant differences. The sequence length we tested are $\{5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 20, 30, 40, 50\}$. We run every experiment at least two times with different seeds to validate the results, and averaged the accuracies for our final plot. We notice that with input sequences longer than ten, the accuracy decrease drastically, performing similarly to random guessing. This is due to the intrinsic problem in RNNs due to which long term memory is easily lost after a number of iterations.

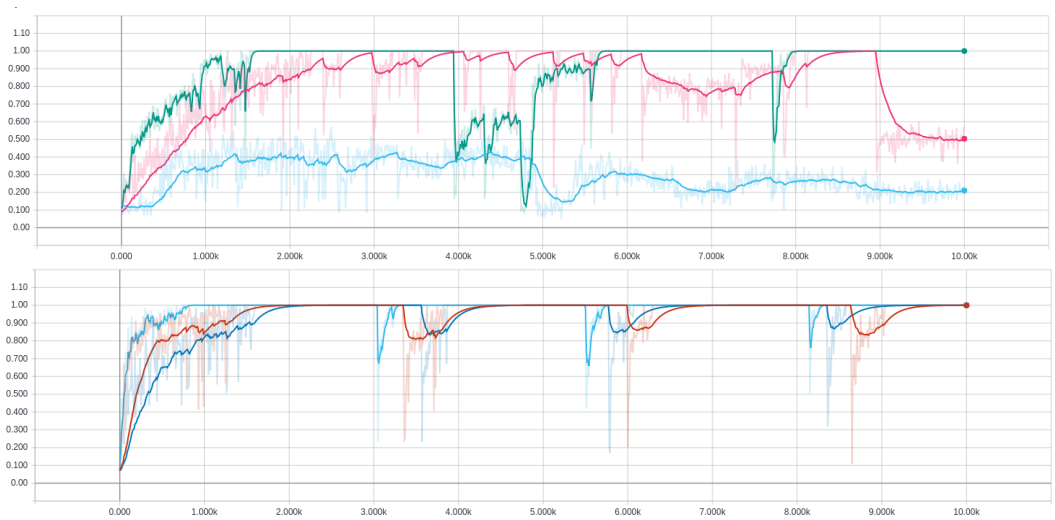

Figure 2: Comparing Normal and Orthogonal W_{hh} initialization for RNN.

After this experiment we also tried to adopt Orthogonal initialization of the weight matrix in the recurrent connection between hidden layers in RNNs. As proposed in [1], this initialization should drastically improve convergence time for this type of network. According to our experiments, this holds also in our case , with models converging after few iterations to an accuracy of 1.0 as shown in Fig. 2. In the first subplot from tensorboard we plot curves obtained with RNN using normal initialization of weights, in particular for sequence length of 5, 10, 20 (respectively in green, pink, light blue). In the second picture we show result for the same experiment using orthogonal initialization of W_{hh} matrix. In this case, sequence length of 5, 10, 20 are respectively colored in light blue, orange, blue. It is evident that orthogonal initialization helps the model to converge faster. It also looks like the local optima where the network converges is more stable than the one reached with random initialization. This can be seen considering as only in few cases over the whole training procedure the accuracy decreases when orthogonal initialization is used instead of normal.

1.4

Stochastic gradient descent is one of the most popular variants to perform gradient descent in a smarter way with respect to the basic batch update. The main issue solve by SGD is that it avoids redundant computations in the batch update using the whole datasets. Indeed, datapoints in large enough datasets tend to be similar or redundant with other ones. For this reason SGD usually learns much faster than batch gradient descent (it computes more, but much faster, update steps). In addition, SGD allows to leave local minima in the space of the objective function thanks to the stochastic component (the gradient changes drastically among different datapoints). By doing this, in combination with a slow decrease in learning rate, SGD is still granted to converge. Despite this, SGD can still be seen as a simpler approach to gradient descent if compared to other variants such as RMSprop and Adam. These variants, among many others, build upon SGD trying to solve some issues in order to further improve the optimization process.

One of the problems SGD is not able to deal with is the navigation in surfaces where the gradient is much steeper in a dimension with respect to the others. In a 3D example, this can be pictured in valley-shaped regions of space surrounding a local minima. In this situtation, SGD would bounce from one side of the local minima to the other in the dimension with steepest slope, but would move very slowly in all the other dimensions. This slow convergence issue can be solved considering the history of the update steps. Momentum is a method that includes this information in the computation of the update steps. This is achieved by summing to the update step the update at the previous step discounted by some value γ. As a result, SGD with momentum accelerates the movement in the relevant direction while also reducing the oscillation in the steepest dimension.

Another issue intrinsic in the original SGD method is that the decay in learning rate has to be defined a priori, without the possibility to adapt the decay steps on the fly at training time. Adagrad is another algorithm for gradient descent [2] that aims to solve this problem using an adaptive approach to learning rate decay. In more details, the updates vary their weight in relation to how frequent a feature is seen at training time. Every parameter has a different learning rate that is adapted dinamically at every update step. This is done by considering the previous gradients for that parameter to rescale a global learning rate. The global learning rate η is divided by the square root of the sum of squares of the gradients with respect to the particular parameter plus a smoothing term to avoid numerical errors. In practice, this update is optimized using a dot product between the vector of partial derivatives and the matrix of rescaled learning rates. RMSprop, proposed by Geoff Hinton [3], includes a slight modification on top of Adagrad to tackle the problem with the sum of gradients which is monotonically increasing, forcing the learning rate to become null possibly before the convergence to the local minima. To do this, the sum of past gradients is discounted expononentially (exponentially decaying average of squared gradients).

Finally, Adam algorithm aims to merge the benefits proposed by RMSprop with the idea of momentum. Adaptive Moment Estimation (ADAM for short), computes an exponentially decaying average of past squares gradients (to adapt the learning rate) but also include an exponentially decaying average

[1] https://arxiv.org/abs/1702.00071

[2] Duchi, J., Hazan, E., & Singer, Y. (2011). Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. Journal of Machine Learning Research, 12, 2121–2159. Retrieved from http://jmlr.org/papers/v12/duchi11a.html

[3] Lecture 6e of his Coursera Class: http://www.cs.toronto.edu/tijmen/csc321/slides/lecture_slides_lec6.pdf

of the gradients themselves, which acts similarly to the momentum approach. Numerically, Adam estimates the first and second moment of the gradients and remove their intrinsic bias towards zero by using two hyperparameters β_1, β_2. The unbiased estimates of the first and second moment are then substituted respectively to the gradients and the matrix of decaying average of past squared gradients in the update equations we previously saw in RMSprop.

In practice, the performances of more complex methods such as RMSprop and Adam are not guaranteed to perform better than SGD. Indeed, they strongly depend on the datasets and model themselves but also on the choice of hyperparameters such as the initial learning rate. For the case of RNNs we are working with, RMSprop is proven to be generally more effective since using the moving average of squared gradients as normalization term avoids exploding and vanishing gradients (through the rescaling in the update step we previously discussed)

1.5

(a)

The LSTM cells contains 4 different gates. Each of them is responsible for a different action in the process of combining the new input with the previous output to update the cell state of the LSTM. All of the gates take as inputs the previous output and the new input, multiplying them by weight matrices and adding a bias vector. The *forget* gate aims at deciding what information in the previous cell state is no longer useful and should be forgotten based on the new input. The non-linearity used in this layer is a sigmoid, since we want to decide how much information in the cell state to keep. When the value for an element in the forget output vector is 0, it means we want to remove the corresponding content from the cell state. An output of 1 means we want to keep all of the relative information in the cell state. For the same reason (gating between zero and one), a sigmoid non-linearity is also chosen for the input and output gates. The *input modulation gate* is used to generate a vector of new candidate values for the cell state. The tanh non-linearity is chosen for this gate, since it helps avoiding the vanishing gradient problem (its second derivative is far from zero for a longer range, compared to other non-linearities). The *input* gate uses a sigmoid non-linearity for the reason discussed previously. The purpose of this gate is to choose how important are the updates proposed in the new candidate vector, rescaling the values accordingly before combining the new information with the cell state after the forget step. Finally, the *output* gate is used to determine which information contained in the cell state is relevant for the current output, depending on the current input and the previous output. Once again, sigmoid is used to have values between zero and one in this gate.

(b)

The number of trainable parameters in LSTM is independent from the length of the input sequence T. All of the four gates use the same two type of weight matrices and one bias vector to combine the information from the input and previous output. The elements in the batch size are independent from each other. In particular, we have the following number of parameters for every gate (where $h \in \{g, i, f, x\}$):

$W_{hx} : n \times d$
$W_{hh} : n \times n$
$b_h : n$

Which, combining the 4 different gates, result in:

$N_{LSTM} = 4 * n * (1 + d + n)$

If we want to also consider the linear mapping to the output, then we should add:

$W_{oh} : o \times n$
$b_o : o$

$N_{linear} = o * (n + 1)$

1.6

In Fig. 3 we show how the LSTM behaves when increasing the sequence length for input palindromes. In this experiment, we fix the training steps to $10,000$ and the learning rate to 0.001. The sequence lengths we tested are $\{5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 20, 30, 40, 50\}$. We run every experiment

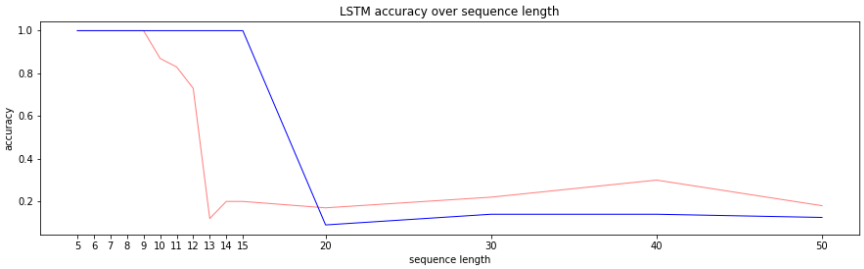

Figure 3: Accuracies of LSTM over different sequence lengths.

at least two times with different seeds to validate the results, and averaged the accuracies for our final plot. In comparison with Vanilla RNN we notice that LSTM is more performant with longer sequences, with accuracy reaching one for lengths up to 15 digits. This behavior is of course expected, since due to the structure of the LSTM cell the network is able to keep in memory information from time steps further back in time. One way to notice this is by considering how the cell state is updated at every time considering an element wise weighting (given by the gates outputs) of the new proposed cell state and the old cell state. On the other side, in Vanilla RNN cells the cell state is completely overwritten at every time step.

2 Modified LSTM Cell

All the following answers assume an $=$ is covering discontinuities between the 3 different parts of the proposed gate function.

2.1

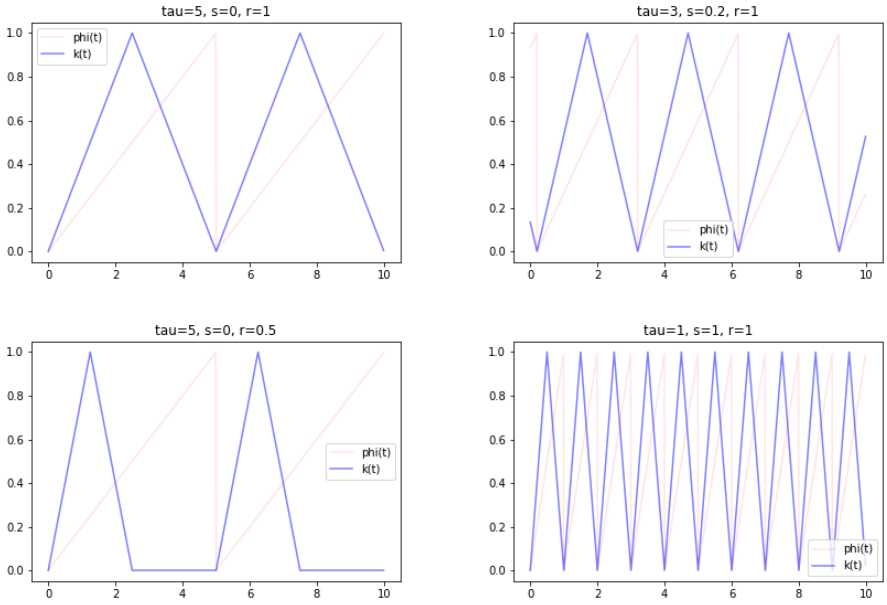

Figure 4: Behavior of the new gate over time in the extended LSTM when tuning the parameters τ, s and r_{on}.

In fig. 4 we show the behavior of the gate introduced in the extended LSTM cell, in function of the time step t. In all the plots we show the output of the gate function $k^{(t)}$ and we also draw in the background the relative function for $\phi^{(t)}$, depending on τ and s. Three parameters are used to tune the effect of the gate itself. The parameter s is used to shift the filter on the time axis without

modifying the shape of the gate function itself. It can be seen as regulating the translation of a function. The parameter τ regulates the period of the filter function over the time t. This is evident looking at the modulo operation in the definition of $\phi^{(t)}$. The function $\phi^{(t)}$ generated depending on t and parameters s and τ is a sawtooth-shaped wave. The function k over the input ϕ depends on the third parameter r_{on}, which describes what percentage of the period assumes a triangle shaped filter and what part is null. In our plots, we see how varying r_{on} from 1 to 0.5 changes the shape of the gate function (considering top left and bottom left pictures). If the parameter is larger than one, the gate function will not have a phase with null activation, but will be a transformed version of the sawtooth wave we see in ϕ. The most interesting case for this gate function is probably the one with r_{on} rangin between 0 and 1, since it creates interval of partially opening gates and interval of closed gates. The effect of tuning the shift value from 1 (null) to 0.2 is shown in the top right corner, while in the bottom right we plot the changes introduced by decreasing the period from 5 to 1.

2.2

Considering the explanation of the gate function behavior we just provided and the way it's applied to in the LSTM cell, we can discuss its behavior in the training phase of the network. The new gate function is applied twice in the cell: before the cell update and before the cell output. In particular, we see that for high values of the gate $k^{(t)}$ the new proposed values for the cell state $\tilde{c}$ and the output $\tilde{h}$ are more important in the weighted sum than the values from the previous time step. On the contrary, if the gate values are zeros, the LSTM cell is not influenced by the new inputs and both its internal state and its output are kept the same as the previous time step. Intuitively, this means that the cell state is only updated in a part of the period τ depending on the value of the parameter r_{on}. In particular, the maximum effect of the new input over the cell state update happens in the central part of the wave we show in our plots, gradually decreasing towards the inactive/blocking moments in the wave period (where the gate function assumes value zero). This gate can be useful in a different range of scenarios when we want to only consider input at periodical time steps. For example, this gate can be used in case of noisy input, where the noise itself appear periodically at already known time steps in the sequence. In this setting, we can use the extended cell to avoid being influenced by misleading inputs, keeping its state constant until the noisy part of the sequence is finished. The same idea is also true for similar cases, generally when we want to learn structures in a input sequence with a known periodical behavior. Another example of use case is the one of frequency discrimination. From a practical point of view, we may try this task in applications such as BPM (Beats Per Minute) or instrument detection in music tracks. Hopefully, the network with this kind of gate will be able to adapt to periodic waves like the one produced by drums and basses (assuming that the music tracks are characterized by a repetitive fixed frequnce) in sound tracks and learn their frequencies for such goals.

2.3

The effects of the parameters τ, s and r_{on} on the gate function are the ones we already discussed in Sec. 2.1 when commenting the plots. Considering if those parameters can be learned or not, we know that to optimize the loss of our neural network, all the trainable parameters are updated in order to maximize the throughput of useful information from the input, adapting to itself. Since in this case the parameters are used to described a wave, I believe that they can be learned given an appropriate periodic input (ex: frequency discrimination tasks). In addition, I think that if we let only one of the parameters to be trained constraining the others to be fixed, our network will learn to adapt to the input signal in only one of its aspects, (either the period, the shift or the active sampling time range over the period), and this would be a simpler task than learning all three parameters combined.

3 Recurrent Nets as Generative Model

3.1

3.1.a

See python code.

3.1.b

For this experiment, we train our LSTM to generate text training on the book *Anna Karenina* by Leo Tolstoj. The model is trained using the default hyperparameter configuration. The characters represented in one-hot encoding vectors having the size of the vocabulary of characters occurring in the corpus. The LSTM is trained for 100k iterations, and every 100 we randomly sample a starting character from the vocabulary, from which to start the generation of text sequences of length $T = 30$. Here we show examples of generated text at different time steps during the training.

t_{100}: "**1** t t t t t t t t t t t t t t"

t_{200}: "**z**e the the the the the the the the "

$t_{1,000}$: "**I** and the same and she was she "

$t_{10,000}$: "**e** was a strange the conversati"

$t_{100,000}$: "**B**ut I can't be a man whose pr"

After only 100 iterations, the LSTM still hasn't learned patterns occurring in sequences (i.e. existing words), but it seems like it noticed that spaces are very frequent in the dataset, and there are never 2 spaces in a row. Thus, it generates one of the most frequent letters ("t") after every space. After 200 iterations, the model learned the most frequent word ("the"), and outputs a sequence of repeated words. This is because the LSTM is starting to learn sequences of length 3/4, but it doesn't know yet that repeating the word is wrong. After 1000 iterations the LSTM start to generate sequences of text that locally make sense (e.g. articles before nouns, conjunctions, verbs after nouns), but the sentence as a whole is still incoherent semantically. After 10k iterations the model learns longer and more rare words, and the sentence start to make sense as a whole. After 100k iterations the model doesn't change drastically anymore, but, overall, it can now handle rare characters like capital letters and apostrophes plus syntactically complex sentences are generated.

This experiment was also repeated on other datasets, noticing how the LSTM learns different patterns characteristic of a particular corpus (dialogues included in quotes, paragraph titles in capital letters, html and latex structures). We show some of these result in the Bonus Task.

3.1.c

Let's now consider how greedy sampling can be changed into a trade-off between random and greedy sampling by including a *temperature* parameter T. For $T = 0$ the randomness collapses to a greedy choice (argmax of the softmax), while at $T = \infty$ the sampling becomes random (samples are drawn from a uniform distribution over the vocabulary tokens). With greedy sampling, the generated sentences often include the most frequent words or combination of words, for example in

*"**R**ussian and the same time with"*
*"**h**e said, and with a smile of th"*
*"**q**uestion of the same time the s"*

we notice that words like *same*, *with*, *the*, *and* are occurring multiple times. In particular, the model is good at handling the random character at the beginning of the sentence and continuing with a meaningful word, but then tends to collapse towards the most frequent sequences. If we include some randomness in the character sampling setting $T = 0.5$, the model greatly improves in variety of the generated text. Example outputs are:

*"**h**e portrait of the country bec"*
*"**e** discussing of the contrary, a"*
*"**I** have worn many of the stairs,"*

If we increase more the temperature at $T = 1$, the randomness in the sampling is increased again, and although some sentences are grammatically correct and semantically meaningful, many of them include random mistakes that can be easily spotted, making the whole output seem not realistic. For example:

*"**K**itty excepted, you irrused for"*
*"**D**olly, and at '\n' that simple alrea"*
*"**L**evin would have anticipated by"*

Finally, we try text generation with a temperature of $T = 2$, which leads to an exaggerated noise in the sampling, where words themselves start to lose meaning. As an example:

*"**f**ea, make out covered: devoie s"*
*"**u**itase, emanto givably up,sâ?"*
*"**L**eft ereieful-laksâs volkaruâ"*

3.2

3.2.1 Generating paragraphs using different book styles

In this experiment, we train multiple copies of our model on different books. Our idea is that the LSTM will patterns from different writing styles and, when asked to generate text continuing from a fixed sentence, they will output content matching with their vocabulary and writing style. To this purpose, we consider the following corpus for training:

- **Anna Karenina**: novel in modern English including some dialogues
- **Shakespeare works**: a 100k characters sample of Shakespeare's works proposed in Karpathy's, poetic style blog[4].
- **The Republic**: the second longest work from Plato, discussing philosophy, law and society
- **The Odissey** epich book from Omero, in an antique English

For all of the four models we fixed some sentences and tried to continue generating text from them. Among them, we report results from one with an "open end", such it could result in meaningful sentences for different types of text. The temperature in this experiment was set to $T = 0.5$ to enforce some randomness in the results. In all of the paragraphs generated below we recognize different writing styles and vocabulary. In Shakespeare, we also see that the model learns to use line breaks to separate utterances and to use capitals for the name of the characters. Overall, we see how every paragraph as a whole often don't make too much sense. This is explainable with the fact that the model only learns relationships in phrases at most 30 characters long, and since the generated paragraph are long 200 characters, they start losing their meaning.

Anna Karenina
***He did not really like** the first to the peasants were standing at the door. He saw the club, and his book was a girl who had always been at the same time and had so much as the carriage with one of the conversation with*

Shakespeare works
***He did not really like** a friend, I will have my*
About my heart with the saltiness meel, but weived thou deeds.
BROTUD:
With days,
Mifte ours, you will make the marriage, sir.
BOYET:
Ay, Sir John! and this was so many as t

The Republic
***He did not really like** to fall under the performer of all the objects, and the souls of the soul, and are all objects against any one who is not the best of all, but there is no difficulty in the same thing is concerned wit*

The Odissey
***He did not really like** to speak of the house. Telemachus spoken of for its bed, for I have never seen the great stringing of the court, and something for myself to the same survain it will be reaching this way, but they may*

3.2.2 Generating LaTeXand HTML sequences

To experiment with something different from text generation, we also tried to predict hyper-textual or type-setting content such as LaTeXand HTML. To test the capabilities of our model, in this case, we

[4]http://karpathy.github.io/2015/05/21/rnn-effectiveness/

ask to continue generating sequences after a specific input using LaTeXor HTML syntaxes. Notice that in both cases the syntax is much more strict than the one in free english text. For example, we know that "\begin{" is correct in LaTeX, while "\bega..." shouldn't be generated at all. For this reason, we consider a temperature of $T = 0$ in this experiment, trying to avoid random noise in the generation of output.

LaTeX

```
\begin{DPalign*}
    \lintertext{\rlap{\indent Then show that we know that it is a maximum.}
\end{DPalign*}

The last term is
$\dfrac{dy}{dx} = \dfrac{1}{\sqrt{\theta^5}} - \dfrac{1}{\sqrt{\theta^5}} - \dfrac{1}
```

HTML

```
<p><font size="2">The structure of a response is signon information aggregate</font></td>
  </tr>
  <tr>
    <td width="180"><b><font size="1">&lt;STATE&gt;</font></b></td>
    <td width="366"><font size="2">Custom
```

These results seem extremely realistic, with beginning and closing tags for commands being correctly placed. This is probably beacus of the syntax in the corpora being much more rigid than the one in free text.