Assignment 2: Recurrent Neural Networks

Dana Kianfar
11391014
University of Amsterdam
dana.kianfar@student.uva.nl

Abstract

In this report we explore two different recurrent neural network architectures, the standard recurrent neural network (RNN) and the long short-term memory network (LSTM). We compare and contrast their performance on a simple task of palindrome completion, where we demonstrate that the LSTM is capable of modelling long-term dependencies better than the RNN. We then train a two-layered LSTM character-level language model and use it to decode randomly-initialized sequences and observe interesting patterns that arise during training such a model.

1 Vanilla RNN vs. LSTM

1.1 RNN Gradients

We can simplify the loss function (at the last time step T) by using one-hot encoding for ground-truth labels $y^{(T)}$. Let c be the index of the true class for a training pair $(\mathbf{x}^{(T)}, \mathbf{y}^{(T)})$ at the last time step.

$$\mathcal{L}^{(T)} = -\sum_{k=1}^{K} y_k \log \hat{y_k}^{(T)} = y_c \log \hat{y_c}^{(T)} = -p_c^{(T)} - \log \sum_{k=1}^{K} \exp p_k^{(T)}$$

We can derive the gradient for $\mathbf{W}_{oh}$ as follows.

$$\frac{\partial \mathcal{L}^{(T)}}{\partial \mathbf{W}_{oh}} = \sum_{k=1}^{K} \frac{\partial \mathcal{L}^{(T)}}{\partial p_k^{(T)}} \cdot \frac{\partial p_k^{(T)}}{\partial \mathbf{W}_{oh}}$$

$$\frac{\partial \mathcal{L}^{(T)}}{\partial p_k^T} = -\delta_k^c + \frac{\exp p_k^{(T)}}{\sum_{j=1}^{K} \exp p_j^{(T)}} = \hat{y_k}^{(T)} - \delta_k^c$$

$$\implies \frac{\partial \mathcal{L}^{(T)}}{\partial \mathbf{p}^T} = \mathbf{y}^{(T)} - \hat{\mathbf{y}}^{(T)}$$

For simplicity, we define $\mathbf{0}$ as a column vector of zeros.

$$\frac{\partial p_k^{(T)}}{\partial \mathbf{W}_{oh_{ij}}} = \left[\frac{\partial p_k^{(T)}}{\partial \mathbf{W}_{oh_{ij}}}\right]_k = \left[\frac{\partial (\mathbf{W}_{oh(k:)} \cdot \mathbf{h}^{(T)} + b_{o_k})}{\partial \mathbf{W}_{oh_{ij}}}\right]_k = \left[h_j^{(T)} \delta_i^k\right]_k = \begin{bmatrix} \mathbf{0} \\ h_j^{(T)} \\ \mathbf{0} \end{bmatrix}$$

$$\implies \frac{\partial p_k^{(T)}}{\partial \mathbf{W}_{oh}} = \begin{bmatrix} \mathbf{0}^T \\ \vdots \\ \mathbf{h}^{(T)T} \\ \vdots \\ \mathbf{0}^T \end{bmatrix} \quad \text{(a matrix of the same size as } \mathbf{W}_{oh}\text{)}$$

30th Conference on Neural Information Processing Systems (NIPS 2016), Barcelona, Spain.

Putting the results together, we have the final form for $\frac{\partial \mathcal{L}^{(T)}}{\partial \mathbf{W}_{oh}}$.

$$\frac{\partial \mathcal{L}^{(T)}}{\partial \mathbf{W}_{oh}} = \sum_{k=1}^{K} \frac{\partial \mathcal{L}^{(T)}}{\partial p_k^{(T)}} \cdot \frac{\partial p_k^{(T)}}{\partial \mathbf{W}_{oh}} = \sum_{k=1}^{K} (\hat{y_k}^{(T)} - \delta_k^c) \cdot \begin{bmatrix} \mathbf{0}^T \\ \vdots \\ \mathbf{h}^{(T)^T} \\ \vdots \\ \mathbf{0}^T \end{bmatrix} = \left[\mathbf{y}^{(T)} - \hat{\mathbf{y}}^{(T)} \right] \cdot \mathbf{h}^{(T)^T}$$

For the gradient $\frac{\partial \mathcal{L}^{(T)}}{\partial \mathbf{W}_{hh}}$, we can express the chain rule as follows. Let d_H be the dimensionality of the hidden state, thus W_{hh} is a $d_H \times d_H$ matrix..

$$\frac{\partial \mathcal{L}^{(T)}}{\partial \mathbf{W}_{hh}} = \sum_{l=1}^{d_H} \frac{\partial \mathcal{L}^{(T)}}{\partial h_l^{(T)}} \frac{\partial h_l^{(T)}}{\partial \mathbf{W}_{hh}}$$

For the first term in the right-hand side, reusing results from the previous derivation, we know that

$$\frac{\partial \mathcal{L}^{(T)}}{\partial h_l^{(T)}} = \sum_{l=1}^{d_H} (\mathbf{W}_{oh}^T)_{l:} \cdot (\hat{y}^{(T)} - y^{(T)})$$

For the second term, we have the following.

$$\frac{\partial h_l^{(T)}}{\partial \mathbf{W}_{hh_{ij}}} = \frac{\partial \tanh(const. + \mathbf{W}_{hh_{l:}} \cdot \mathbf{h}^{(T-1)})}{\partial \mathbf{W}_{hh_{ij}}}$$

We know that $\frac{\partial \tanh(\mathbf{x})}{\partial \mathbf{x}} = (1 - \tanh^2(\mathbf{x}))$. We use the chain rule and product rule to achieve the following result.

$$\frac{\partial h_l^{(T)}}{\partial \mathbf{W}_{hh_{ij}}} = (1 - h_l^{(T)^2}) \sum_{m-1}^{d_H} \delta_{i,j}^{l,m} h_m^{(T-1)} + \mathbf{W}_{hh_{l:}} \cdot \frac{\partial h_m^{(T-1)}}{\partial \mathbf{W}_{hh_{ij}}}$$

Where $\delta_{i,j}^{l,m} = \delta_j^l \delta_j^m$. Putting all the results together, we have that

$$\frac{\partial \mathcal{L}^{(T)}}{\partial \mathbf{W}_{hh}} = \sum_{l=1}^{d_H} (\mathbf{W}_{oh}^T)_{l:} \cdot (\hat{y}^{(T)} - y^{(T)}) \left[(1 - h_l^{(T)^2}) \sum_{m-1}^{d_H} \delta_{i,j}^{l,m} h_m^{(T-1)} + \mathbf{W}_{hh_{l:}} \cdot \frac{\partial h_m^{(T-1)}}{\partial \mathbf{W}_{hh_{ij}}} \right]$$

Optimization Problems As demonstrated above, there is a temporal dependence between $\frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{W}_{hh}}$ and $\frac{\partial \mathbf{h}^{(t-1)}}{\partial \mathbf{W}_{hh}}$, i.e. a recursion. The base case of this recursion is $\frac{\partial \mathbf{h}^{(1)}}{\partial \mathbf{W}_{hh}}$, where we cannot compute the second summand $\frac{\partial h_m^{(0)}}{\partial \mathbf{W}_{hh_{ij}}}$ as the product rule no longer applies ($\mathbf{h}^{(0)}$ is randomly initialized). When performing backpropagation on an unrolled RNN, the gradients are computed via many matrix multiplications. Given that the tanh and sigmoid activations have small gradients for very large or small inputs at each timestep (i.e. neuron saturation), the product of the gradients at different timesteps can vanish very rapidly. As a result, the network cannot backpropagate error signals for long distances, and thus it cannot learn long-term dependencies.

In particular, we can observe that for a large number of timesteps, if the jacobian matrices produced by each individual term below has their largest eigenvalue smaller or larger than 1, we experience the vanishing or exploding gradients problem respectively.

$$\frac{\partial \mathbf{h}^{(T)}}{\partial \mathbf{h}^{(1)}} = \frac{\partial \mathbf{h}^{(T)}}{\partial \mathbf{h}^{(T-1)}} \frac{\partial \mathbf{h}^{(T-1)}}{\partial \mathbf{h}^{(T-2)}} \frac{\partial \mathbf{h}^{(T-2)}}{\partial \mathbf{h}^{(T-3)}} \cdots$$

1.2 Vanilla RNN

We implement a basic RNN model and test its performance on a palindrome prediction task. A network is trained unrolled for a fixed number of timesteps and trained to predict the last element of the palindrome. Therefore, the task is simply to remember the first element in the palindrome, and ignore all other input elements. This task is specifically designed to see how long the RNN can carry the information about the first element in the palindrome.

1.3 Palindrome RNN

We test the performance of the RNN model across on the palindrome task with length $T \in \{5, 10, 30, 50\}$. Unless otherwise stated, we use the default parameter values provided for this exercise. For each experiment, we present the train accuracy and a legend which denotes the optimizer and learning rate. We limit each experiment to 4 different learning rates. We refer to each experiment by the palindrome length, i.e. $T = 5$ refers to the experiment where the palindrome length is 5.

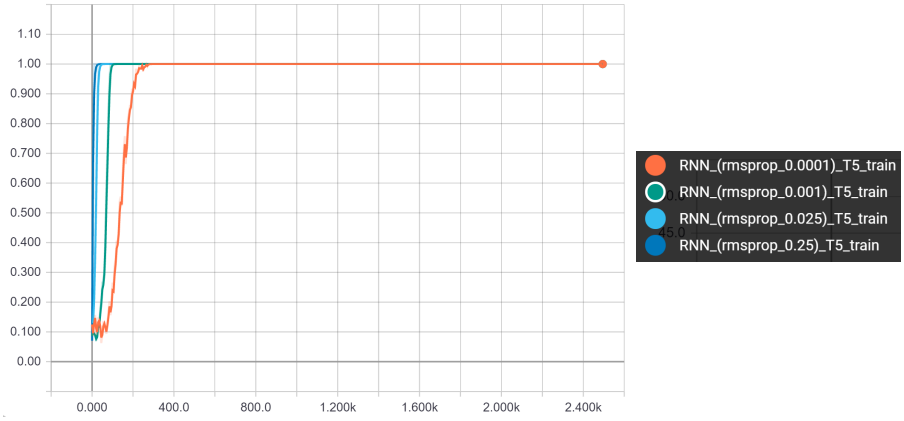

Figure 1: T=5 with RNN, optimized with RMSProp

As we observe in 1, the RNN is capable of learning palindromes of length 5 with ease. All models converge after 200 training steps and are not sensitive to the learning rate.

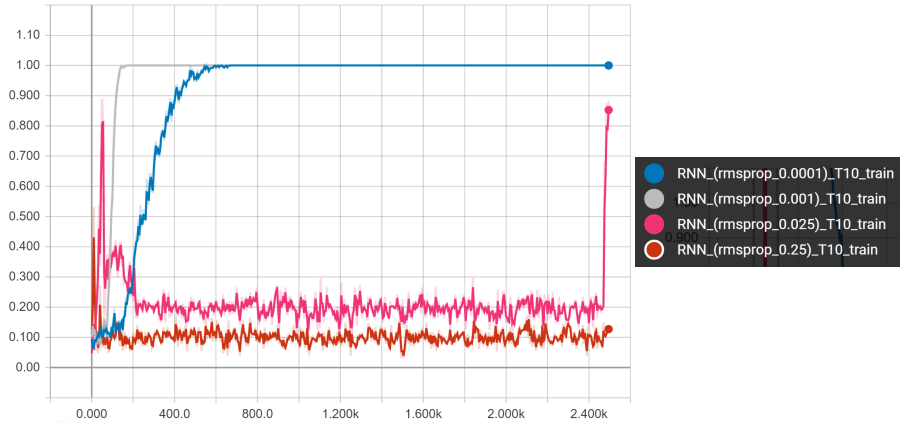

Figure 2: T=10 with RNN, optimized with RMSProp

Figure 2 presents good performance of the RNN on palindromes of length 10. We vary the learning rate and discover that the model is more sensitive to this parameter compared to the easier task of $T = 5$. In particular, large learning rates empirically cause the model to diverge. The best performance is obtained by a learning rate of $1e - 4$.

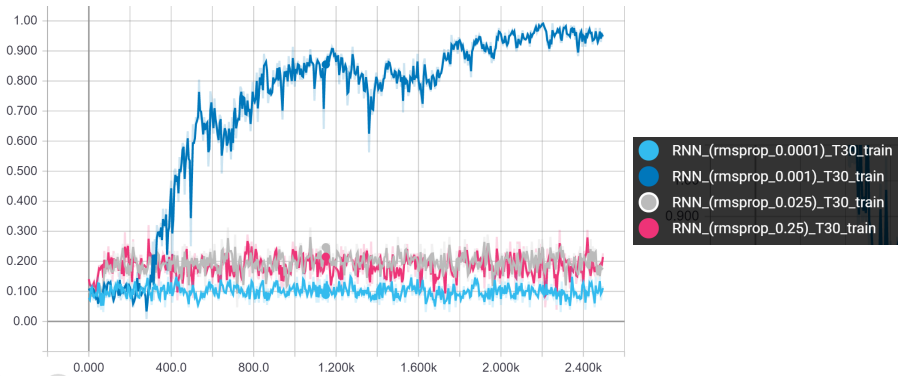

Figure 3: T=30 with RNN, optimized with RMSProp

From Figures 3 and 4 we observe that the RNN struggles to model long-term dependencies past 30 time steps when optimized with RMSProp. When $T = 50$, our parameters did not find a model that converged to an accuracy above past 50%.

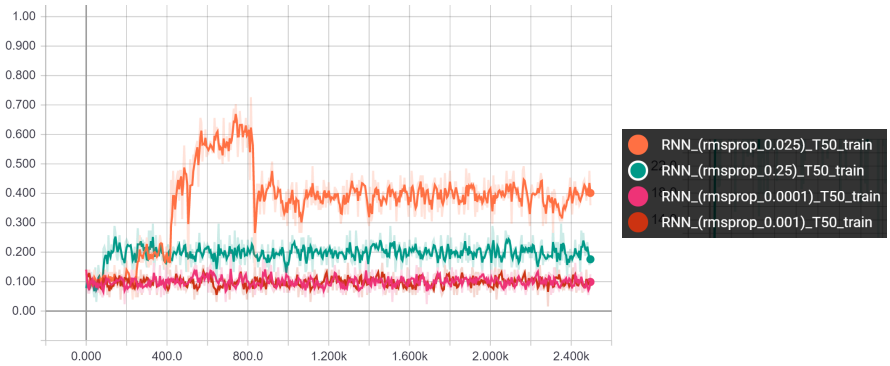

Figure 4: T=50 with RNN, optimized with RMSProp

It should be noted that under different (hyper-)parameter and optimizer settings, this performance can be improved. However, for longer time steps, it becomes increasingly difficult to find a parameter setting that provides desirable results and stable training behavior. We further note that for this task, a validation or test set was not used as we have access to the true distribution of the data. Therefore, by principle we are not overfitting on the regularities of any particular training set. For small palindrome lengths, it is possible for an RNN to simply memorize all palindrome sequences. For this reason, the hidden state of our networks are small (set to the default parameters).

1.4 Stochastic Gradient Descent Variants

A good learning rate results in fast convergence and good local optima. Vanilla stochastic gradient descent uses a fixed learning rate η to update the model parameters. Hand-tuning η is time-consuming and often non-trivial. Different models, datasets and training parameter settings require different values of η for stable learning and convergence. Small values of η slow convergence, and conversely large values result in a divergent training regime, especially on highly non-convex loss surfaces. Learning rate schedules can improve training, where we anneal the learning rate from large to small as training progresses to achieve faster convergence and better local optima. However, schedules are still an inflexible design choice as they do not adapt to the dataset or the task at hand, and are indeed a fixed choice.

Different variants on stochastic gradient descent use local information to tune the learning rate automatically during training. Two commonly used variants are Adam and RMSProp. These methods use momentum and adaptive learning rates to tune the learning rate during training.

Momentum Simply put, momentum accelerates learning by using information from the previous training setip update as a short-term memory. By adding a portion of the last gradient update to the current update, dimensions whose gradients continue to increase or decrease in the same (or similar) directions receive a stronger update. Dimensions whose gradients from the last update and the current update have changed direction by a large enough margin (sensitivity to such fluctuations is controlled by a hyperparameter), receive a weaker (i.e. lower magnitude) update than they would have under SGD. This enables smoother learning and makes the learning more robust to noisy gradient updates from stochastic mini-batches and non-convex surfaces. Momentum is commonly related to a ball accelerating as it rolls down a hill. As the ball rolls further down, its acceleration increases as its velocity vector (gradient of the acceleration) is consistently pointing towards the bottom of the hill. Upon arriving at the bottom of the hill, however, the velocity vector changes direction and the ball slows down gradually.

Adaptive Learning Rates This method uses a learning rate per parameter. This method provides the benefit of adapting to each dimension of the parameter space and tuning it individually. When the data has steep curves on some dimensions of the loss surface and not on the others, using a fixed η can result in oscillation. Using adaptive learning rates, the η for each parameter can be influenced by previous updates to that specific parameter, and provide better optimization for model parameters.

The benefits of methods such as momentum and adaptive learning rate are that they incorporate local information to adapt the learning to the specific task and dataset at hand. This results in less hand-tuning and better learning performance.

1.5 LSTM

LSTMs are specifically designed to improve upon the long-distance dependency modelling shortcomings of RNNs. This is performed via four gating mechanisms that, in a principled way, control what information to remember, to delete and to output at any particular step. This approach stands in contrast to vanilla RNNs, which expose their hidden state at every step with no regards towards controlling the information contained in its hidden state. Concretely, the LSTM has two hidden states, the cell state c and the "exposed" hidden state h. At each timestep, h is exposed as the output of the LSTM, while c acts as a conveyer belt along which information is passed through timesteps, but it is never directly exposed.

Gating Mechanisms

All gates are fully connected layers with bias terms. The weights and biases of these gates persist throughout the timesteps (between each update). The activation functions are either the sigmoid or the hyperbolic tangent.

Input Gate This gate controls what information to add to the cell state c_t of the LSTM unit at any given time step t. "New" information arrives to the LSTM unit via the recurring hidden state h_{t-1} from the past step, and the current input x_t. The input gate, which is a fully-connected MLP with a sigmoid activation, is responsible for choosing which information is relevant to the current and future time steps. The use of the sigmoid activation, which provides outputs in $[0, 1]$, provides a coefficient for the preactivations of the input gate. An very small activation (close to zero) for any value indicates that it should not be added to the current cell state, whereas a large activation (close to 1) indicates that the value should be added as-is to the cell state.

Input Modulation Gate Working in conjunction with the input gate, this gate prepares a corresponding candidate value for each input value (h_{t-1} and x_t). Supposedly, this gate processes in inputs such that they are consistent with the current cell state and relevant to the current time step. The activation function used here is the hyperbolic tangent. This allows the gate to output values between $[-1, 1]$ which can be added to the cell state to increase or decrease any specific value in the cell state. The candidates are cross-multiplied with the output of the input gate, which regulates the amount by which each candidate value contributes to the cell state.

Forget Gate This gate is in charge of discarding information that is not useful for the current or future timesteps. This decision is made by observing the inputs to the LSTM unit (h_{t-1} and x_t),

which are from time steps $t-1$ and t respectively, and deciding which cell state values from the previous time step c_{t-1} should be replaced (i.e. forgotten). This layer, similar to the input gate, has a sigmoid activation which allows it to output proportions by which the cell states should be allowed to persist. The previous cell state c_{t-1} is cross-multiplied by the output of this gate.

Output Gate This gate is simply responsible for outputting relevant information for the current timestep. It acts as a filter on top of the cell state by processes the inputs to the LSTM cell, and performing the sigmoidal activation. The activations of the output gate are proportions by which each value in the cell state should be transferred to the hidden state. The final output of the LSTM unit is its "exposed" hidden unit at that time step, h_t, which is obtained by a cross product of the current cell state c_t and the output of the output gate.

Parameter size

Each gate is composed of a weight matrix of size $(d_H + d_X) \times d_H$ and bias units of size d_H, where d_H and d_X are respectively the dimensionality of the hidden state and input vectors at each time step. Therefore, the total number of paramters for all four gates are computed as follows.

$$4 \times (d_H + d_H \times (d_H + d_X)) = 4d_H(1 + d_H + d_X)$$

Adding the parameters of the fully connected layer responsible for computing the logits, we have the following.

$$4d_H(1 + d_H + d_X) + V(1 + d_H)$$

Where V is the number of classes.

1.6 Palindrome LSTM

Similar to the experiments of section 1.3, we test the LSTM unit with palindromes of varying length.

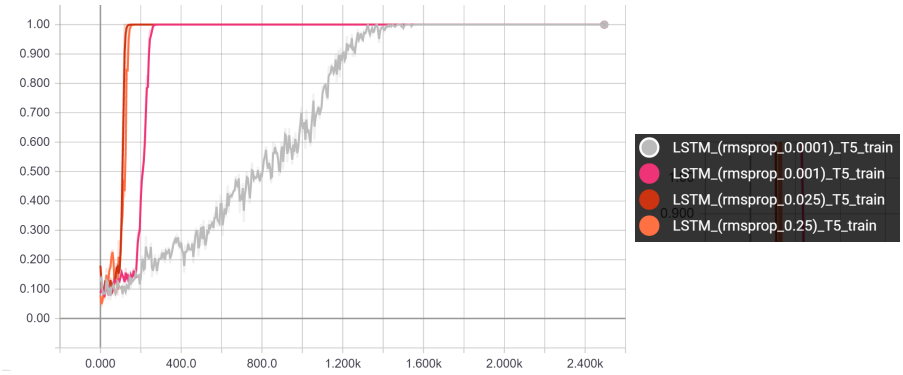

Figure 5: T=5 with LSTM, optimized with RMSProp

As we observe in 5, the LSTM is capable of learning palindromes of length 5 with 100% accuracy. We empirically observed that LSTMs tend to reach this level of accuracy after more training steps than the RNN, which is expected since the LSTM has more parameters to fit. Across all our experiments, we observed that the LSTM performs very strongly with large learning rates when optimized with RMSProp. In particular, all experiments with a learning rate of 0.5 converged to 100% accuracy, with the exception of $T = 100$ which we omit from this report.

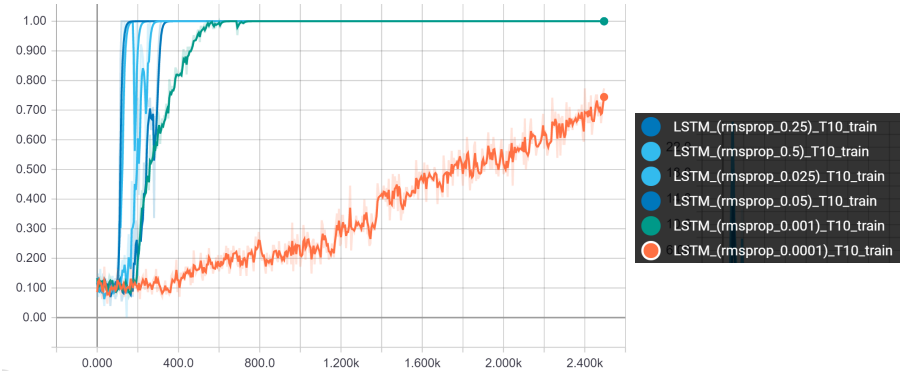

Figure 6: T=10 with LSTM, optimized with RMSProp

Figure 6 presents good performance of the LSTM on palindromes of length 10. The model converges to 100% accuracy in all cases, asuuming that with additional training steps the model with a learning rate of $1e-4$. The best performance is achieved with a learning rate of 0.5. In comparison to the RNN, the LSTM does not experience a degradation in performance up to $T = 10$.

From Figures 7 and 8 we observe that the LSTM begins to struggle to model long-term dependencies past 30 timesteps when the learning rate is too low, i.e. the model converges to an undesirable local optima. The best models in both experiments have learning rates of 0.5 and 0.25 despite the overshooting behavior experienced when the model leaves a desirable optima (visible in both figures around timestep 400).

Based on the results of this task, we have seen that LSTM units are more capable of learning long-term dependencies. We can attribute this behavior to the gating mechanisms of the LSTM units, which allow it to explicitly make decisions on whether to incorporate new information to the cell state at each time step. In the case of the palindrome task, the LSTM simply needs to ignore all inputs (i.e. avoid updating the information of the initial input in its cell state).

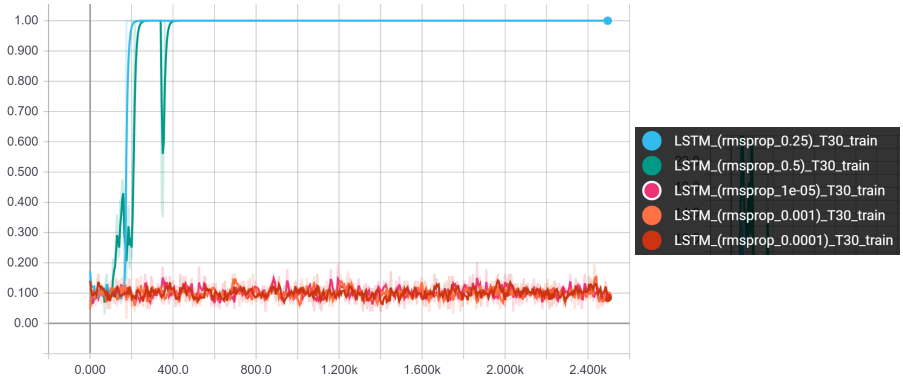

Figure 7: T=30 with RNN, optimized with RMSProp

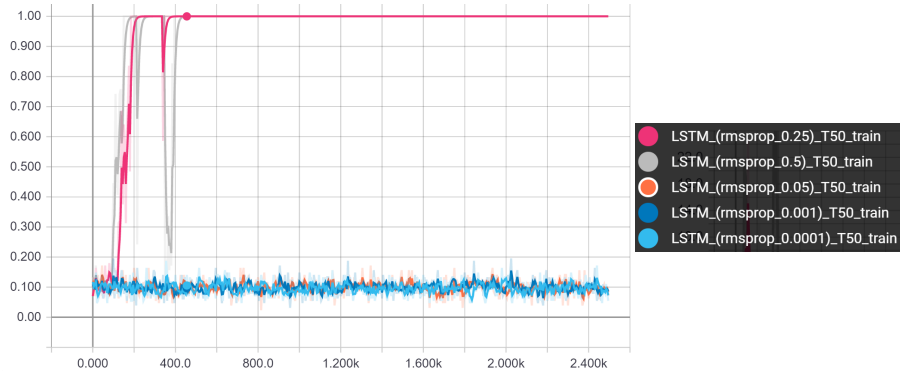

Figure 8: T=50 with RNN, optimized with RMSProp

2 Recurrent Nets as Generative Models

In this section we train an LSTM as a character-level language model on a small corpus. In addition to the default parameters, we experiments with the Adam optimizer, different learning rates, and an embedding matrix. During training, we periodically perform inference via two methods. In the first, we initialize the hidden state of the LSTMs to zero and provide it with a random character input, and perform inference by sampling from the softmax distribution for 50 steps. In the second method, we encode a short sequence of text and use the LSTM state to continue complete the sequence by decoding for 100 steps. In our experiments, greedy decoding resulted in repetitive generated text.

Random Character Decoding We train an LSTM model on a compilation of several books by Carl Sagan. We sample from the model every 50 steps and present a subset of the decoded sequences below. Since our models are trained using truncated backpropagation with 30 steps, in order to separate the first 30 decoded characters we split them in Table 1. For the decoded tokens at step 0, we omit the characters past time step 30.

Step	First 30 Characters	Last 20 Characters
0	q,6N–e]Ü5“18>JS’qfeh;+*D*ⓒCⓒJ	-
100	Ze sugthage we clol sutes ute	Ze sugthage we clol
200	‘in capsionary It the moving 2	26, byfest on the marne of animent Fermar
300	ould beis. Thereece count of t	he lawsian are reall
400	I may to paracellions and in	time radio time.. Wou
500	moneasts are not the idea down	too dead, but triliple
1500	You look wavellices in millions	of Triton than change from magnifications

Table 1: Random character decoding

At step 0, the model effectively decodes random characters. By steps 100 and 200, we notice that the model begins to learn spacing and syntax. It then learns how to form short words, and the length and "correctness" of the words improve at training continues. In the later examples, the model could even decode a few words that are related to a topic. We note that the quality of the decoded sequences degrades after 30 characters, since the model has not been trained to model longer dependencies.

Sequence Completion We train a model until convergence on the first 10 chapters of Origin of Species by Charles Darwin. We then initialize the network by encoding a pre-defined sequence and then decoding for 50 steps. The results are displayed in Table 2.

Initial Sequence	Completion
theory of evolution	require to be compared within their states that there is nothing their cell can cross
Humans grew up in forests	through the rule the pollen traces the seeds we have seen there has been variated animals

Table 2: Sequence completion decoding