Reversible Recurrent Neural Networks

Mazhar Shaikh
IIT Madras
ed13b033@smail.iitm.ac.in

Ganesh Anand
IIT Madras
ed13b020@smail.iitm.ac.in

Mitesh Khapra
Department of Computer Science
IIT Madras
miteshk@cse.iitm.ac.in

Abstract

LSTMs have achieved state-of-the-art performance on the task of Language Modelling. To achieve better performances, We need to train models with longer sequences. However, memory requirements of models grow linearly with sequence length. We introduce Reversible RNNs inspired by LSTMs and RevNets, which have memory requirements independent of sequence length. We test our network on the Penn Tree Bank dataset. The network achieves performance better than or comparable to LSTMs when trained with stored activations. While the network is reversible algebraically, We find that it is numerically unstable. We analyze this and report our hypothesis and conclusions on this instability. We provide code to encourage further exploration of these architectures[1].

1 Introduction

Recurrent neural networks with architectures such as Long Short-Term Memory(LSTM)[1] and Gated Recurrent Units(GRU)[2] have become a commonly used and effective model for many tasks related to sequence learning. These architectures overcome the instability during training of Simple Recurrent Neural Networks(referred to as just SRNN from now on).

The key feature of LSTM and GRU units and variants of them are that they have gating mechanisms. Additionally LSTM units maintain a memory. The gating mechanism allows them to be stable during training and also to remember only important information. But this added complexity improves performance at the cost of increased memory usage.

In this work we focus on the sequence learning task of language modelling. A language model evaluates the probability of a given sentence. The state of the art on language modelling[3] on the Google 1 Billion Words dataset[4] makes use of LSTMs.They can be trained on sequences of words or characters. In the case of [3] they use the output of a character CNN as input to the LSTM.

Backpropagation has become the standard method of training modern neural networks, whatever be their architecture and area of application. This however, requires the storing of activations of all the hidden units. Hence, the space complexity of a network is proportional to the number of hidden units in the networks. For example, a network built for image classification like ResNet[5] has a memory requirement proportional to the number of layers in the network. This however means that increase in complexity of the network in terms of the depth and width of the network leads to an increase in the space complexity and memory requirements of the network. The main bottleneck in the training of larger models is thus the limited memory available in GPUs.

[1] www.github.com/gan3sh500/revrnn

For recurrent neural networks, the memory requirement is dependent on the sequence length of the the training data as well as the number of layers in the network. In typical applications of RNNs, the sequence is often on the order of tens of time steps. This puts a constraint on the number of layers in the network and width of the layers in the network. We introduce Reversible RNNs, which can reconstruct the activations of the network across time steps. This makes the memory requirements of these networks independent of the sequence length.

In Section 2 we review the concepts of language modelling and their training as well as Reversible Networks which have been developed and used in Computer Vision using architectures based on Resnet. In Section 3,...

2 Background

2.1 Language Modelling

The task of language modelling deals with evaluating the probability of a given sentence. This helps them learn not only whether or not a sentence is grammatically correct but also if the sentence is likely in a given language. When trained on enough data they can even gain information about the world such as facts about people or places.

Language models commonly used today make use of Recurrent Neural Networks especially LSTM and GRU and sometimes N-gram models together with Recurrent Neural Networks. Training of Recurrent Neural Networks involves backpropagation of errors over many time steps. During this process, the activations for the Recurrent Neural Network at every time step is required to be stored in memory which leads to linear increase in memory with sequence length. This can cause limitations on the length of sequences that can be trained on and consecutively the length of sequences that the model performs well on.

The gating mechanism of LSTM and GRU also play a crucial role in their performance. The gating allows for the models to selectively remember and forget parts of the sequence into a fixed size vector. This vector can change over the sequence also.

2.2 Reversible Networks

RevNets[6] were introduced as a variant of ResNets which had reversible layers in the sense that the activations of a layer could be reconstructed from the activations of the subsequent layer. The RevNet architecture had memory requirements which were an order of magnitude lower than than similarly sized ResNets. Each residual layer in this architecture takes in inputs (x_1, x_2) and gives the output (y_1, y_2) according to the equations below.

$$y_1 = x_1 + F(x_2)$$
$$y_2 = x_2 + G(y_1)$$

The inputs can then be reconstructed from the outputs using the following equations.

$$x_2 = y_2 - G(y_1)$$
$$x_1 = y_1 - F(x_2)$$

Subsequently, Hamiltonian Reversible Networks[7] were introduced with theoretical proofs of reversibility that guaranteed reversibility with arbitrary depth. These networks achieve stability by placing restrictions on the functions F and G.

$$y_1 = x_1 + K_1^T * \sigma(K_1 * x_2 + b_1)$$
$$y_2 = x_2 - K_2^T * \sigma(K_2 * y_1 + b_2)$$

3 Methods

3.1 Cell states as Partitions

The residual module in RevNet partitions the input x to a layer into (x_1, x_2). The output (y_1, y_2) is concatenated to give y.

$$(y_1, y_2) = RevModule((x_1, x_2))$$

$$(c_t, h_t) = LSTM((c_{t-1}, h_{t-1}), i_t)$$

Comparing with a LSTM cell, we see that the partitions required for the RevModule arise naturally with the structure of the LSTM.

3.2 Architectures

We propose two architectures which are reversible. For designing these architectures we take inspiration from LSTMs as well as from Residual Networks. They were chosen empirically on the basis of performance on being trained using backpropagation without restoration.

3.2.1 Proposal 1

The first proposed architecture makes use of similar activation functions and gating to LSTMs. The computations upto c_t are the same as in LSTM. Afterwards instead of computing an output gate, the output is based on a combination of h_{t-1} and $\widetilde{h}$ which is computed from c_t. The gating is done using r and u which depend on c_t similar to how f and i depend on h_{t-1}.

$$\begin{aligned}
f &= \sigma(h_{t-1}, x_t) \\
i &= \sigma(h_{t-1}, x_t) \\
\widetilde{c} &= \tanh(h_{t-1}, x_t) \\
c_t &= f\frac{1}{2}(\odot c_{t-1} + i \odot \widetilde{c}) \\
r &= \sigma(c_t, x_t) \\
u &= \sigma(c_t, x_t) \\
\widetilde{h} &= \tanh(c_t, x_t) \\
h_t &= \frac{1}{2}(r \odot h_{t-1} + u \odot \widetilde{h})
\end{aligned}$$

The above architecture is reversible. From using h_t, c_t and x_t, we can reconstruct. To reconstruct h_{t-1} we use the following equations.

$$\begin{aligned}
r &= \sigma(c_t, x_t) \\
u &= \sigma(c_t, x_t) \\
\widetilde{h} &= \tanh(c_t, x_t) \\
h_{t-1} &= (2h_t - u \odot \widetilde{h}) \oslash r
\end{aligned}$$

We use $\oslash$ to denote element-wise division operation. Using h_{t-1} we can now compute f and i and $\widetilde{c}$ and then get c_{t-1} as shown in equations below.

$$\begin{aligned}
f &= \sigma(h_{t-1}, x_t) \\
i &= \sigma(h_{t-1}, x_t) \\
\widetilde{c} &= \tanh(h_{t-1}, x_t) \\
c_{t-1} &= (2c_t - i \odot \widetilde{c}) \oslash f
\end{aligned}$$

3.2.2 Proposal 2

The second proposed architecture makes use of ReLUs but uses a similar structure to the first architecture. Here we use multiple terms with different signs instead of gating. Not having element-wise multiplication for gating allows to avoid element-wise divisions in the reconstruction which can help avoid problems when gates have values close to 0 in them.

$$\begin{aligned} f &= \text{ReLU}(h_{t-1}, x_t) \\ i &= \text{ReLU}(h_{t-1}, x_t) \\ c_t &= \frac{1}{2}(c_{t-1} + i - f) \\ u &= \text{ReLU}(c_t, x_t) \\ r &= \text{ReLU}(c_t, x_t) \\ h_t &= \frac{1}{2}(h_{t-1} + u - r) \end{aligned}$$

Here the reconstruction of activation is similar. Using h_t, c_t and x_t, the other activations are reconstructed. First we obtain h_{t-1} as shown below.

$$\begin{aligned} u &= \text{ReLU}(c_t, x_t) \\ r &= \text{ReLU}(c_t, x_t) \\ h_{t-1} &= 2h_{t-1} - u + r \end{aligned}$$

Then using h_{t-1} along with x_t and c_t we can obtain c_{t-1} as shown below.

$$\begin{aligned} f &= \text{ReLU}(h_{t-1}, x_t) \\ i &= \text{ReLU}(h_{t-1}, x_t) \\ c_{t-1} &= 2c_{t-1} - i + f \end{aligned}$$

3.2.3 Proposal 2

The third proposed architecture is very simplistic and is very similar in structure to a residual network. It involves adding a gated vector to its' previous state.

$$\begin{aligned} c_t &= c_{t-1} + \sigma(h_{t-1}, x_t) \odot \tanh(h_{t-1}, x_t) \\ h_t &= h_{t-1} + \sigma(c_t, x_t) \odot \tanh(c_t, x_t) \end{aligned}$$

Reversing this architecture is the most straightforward. We require h_t, c_t and x_t to find h_{t-1} and c_{t-1}. The process is shown below.

$$\begin{aligned} h_{t-1} &= h_t - \sigma(c_t, x_t) \odot \tanh(c_t, x_t) \\ c_{t-1} &= c_t - \sigma(h_{t-1}, x_t) \odot \tanh(h_{t-1}, x_t) \end{aligned}$$

4 Experiments

4.1 Dataset

For our experiments the Penn Tree Bank(PTB) Dataset[8] was used. The PTB dataset consists of around 930,000 words in its training set, 73,000 in validation and 82,000 in test. The vocabulary is 10,000 words and the remaining are replaced with unknown tokens.

PTB was chosen due to its small size in order to test out feasibility of architectures quickly and to compare its performance with LSTMs. As such no pre-trained word embeddings were used either. For experimental evaluation we use the perplexities of the trained models.

4.2 Experiments without reconstruction

The proposed architectures along with the LSTM was trained on the Penn Tree Bank dataset. All of the models used the following network and training parameters. This experiment was done to check the effectiveness of the proposed architectures in order to then find which models to test using restoration.

The third proposed architecture proved to be unstable while the first and second architectures achieved performance similar to LSTMs as shown below.

Parameter	Value
hidden size	200
learning rate	1
lr decay	0.5 every 2 epochs
epochs	13
batch size	20
num steps	20
weight init scale	0.1
gradient clip	5

Table 1: Training Parameters

Model	Perplexity
Basic LSTM	116.033 ± 0.626
Proposed 1	108.965 ± 0.435
Proposed 2	116.008 ± 0.551

4.3 Experiments with reconstruction

The two proposed architectures which performed comparably to LSTMs were implemented such that training happened using reconstructed activations instead of storing them in memory as done before. The reconstruction proved to be unstable with the error in the reconstruction blowing up over time as the network started to train. We discuss possible reasons why this may occur and how this relates to how gating and memory in Recurrent Neural Networks in Section 6.

4.3.1 Implementation

We implemented reversible recurrent networks using Pytorch[9]. For each batch, we perform a forward pass through the entire sequence. We retain in memory only the cell states (c_t, h_t) at the current time step t in the sequence. Thus, at the end of this forward pass, we have only the final cell states (c_T, h_T) in memory. We then perform a backward pass through the sequence. At each time step t during the backward pass, we reconstruct (c'_{t-1}, h'_{t-1}) from (c_t, h_t). We then perform a forward step with (c'_{t-1}, h'_{t-1}) to get (c'_t, h'_t). We then use the torch.autograd.gradient() to calculate the gradients of the weights of the network and (c'_{t-1}, h'_{t-1}). The gradients of (c'_{t-1}, h'_{t-1}) are used in the next step in the backward pass. And the gradients of the weights of the network are accumulated. This implementation entails a computation cost of 5T compared to 3T with naive backpropagation.

5 Related

5.1 Dynamic RNNs

Dynamic RNNs perform a dynamic unfolding of the network. The activations at each step are stored in the memory of the CPU, freeing up memory in the GPU. During backpropagation, the activations are recalled from CPU memory for use. This method is useful for small RNNs but runs into problems with large models. There is also the issue of latency between the CPU and GPU memory, which slows down speed of computation.

5.2 Checkpointing

Checkpointing[10] tackles the problem of space complexity by saving the activations at $\sqrt{T}$ steps. During the backpropagation, the activations required are calculated again with a forward pass. Thus memory requirement for this method is $O(\sqrt{T})$.

6 Discussion & Future Work

We have found the following reasons to be contributing to the instability of the proposed architectures and hypothesize possible solutions worth investigating. Finally we talk about the code accompanying

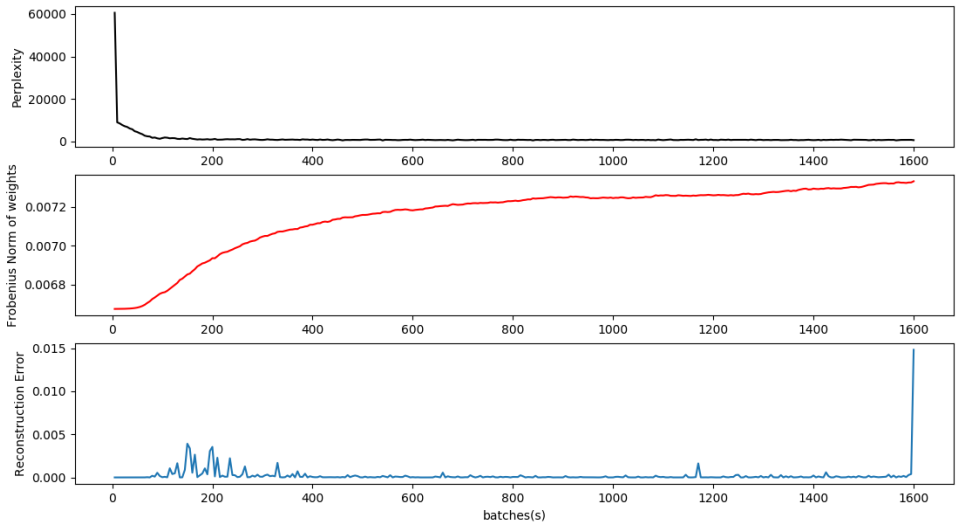

Figure 1: Training of reversible RNN

this paper how it should be helpful moving forward on the task of creating Reversible Recurrent Neural Networks.

6.1 Numerical errors

The training process as show in the figure 6.1 exhibits some interesting properties. As the magnitude of the weights goes up, so does the reconstruction error. In our experiments it was observed that for the network to start training and reduce the loss we needed to initialize the weights very small. But then as the network trains, the weights grow in magnitude and eventually the loss blows up. Naturally we tried methods to regularize the network and to prevent the blowing up of the loss.

6.1.1 L2 Regularization

The most obvious method was to use L2 regularization to keep the magnitude of the weights down but it was found that L2 regularization could only stabilize training when it was so large that the network would massively underfit the data and achieve perplexities of around 700.

6.1.2 Reconstruction Loss Regularization

Another approach we tried was to try and have the network learn to minimize the reconstruction loss by itself by adding it as one of the losses in training. It was found that increasing the scaling for this loss in the training objective helped to stabilize the training but the network again would underfit the data and would achieve perplexities from 650-700.

6.2 Language modelling

We hypothesize that the reason for instability in Reversible Recurrent Neural Networks as opposed to Residual Networks is the gating. The purpose of gating is for the network to be able to selectively forget parts of the state and selectively add to it. This forgetting of information which translates numerically as setting to a value close to 0, explains why the network, as it learns starts reconstructing activations poorly.

6.3 Code

As the problem we tackle is the optimization of memory for Recurrent Neural Networks, we were required to understand the machine learning libraries at the fundamental level. One of the biggest obstacles we faced was that lack of code to perform the reconstruction and specialized backpropagation. And overcoming this was something we spent most of our time on. We hope that with the code we have made available, others may be able to experiment with much more ease and eventually arrive at architectures which are successful at the reconstruction of the activations.

6.4 Future Work

The design of architectures which are mathematically stable using an approach similar to [7] should be done. Additionally we believe an exploration into proving whether or not Reversible Recurrent Neural Networks can be stable needs to be done.

7 References

[1] Sepp Hochreiter and Jurgen Schmidhuber. Long Short-Term Memory. Neural Computation, 9(8):1735–1780, November 1997. ISSN 0899-7667. doi: 10.1162/neco. 1997.9.8.1735. URL `http://www.bioinf.jku.at/publications/older/2604.pdf`

[2] Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning Phrase Representations using RNN EncoderDecoder for Statistical Machine Translation. arXiv preprint arXiv:1406.1078, 2014. URL `http://arxiv.org/abs/1406.1078`.

[3] Jozefowicz, Rafal, Vinyals, Oriol, Schuster, Mike, Shazeer, Noam, and Wu, Yonghui. Exploring the limits of language modeling. arXiv preprint arXiv:1602.02410, 2016.

[4] Chelba, Ciprian, Mikolov, Tomas, Schuster, Mike, Ge, Qi, Brants, Thorsten, Koehn, Phillipp, and Robinson, Tony. One billion word benchmark for measuring progress in statistical language modeling. arXiv preprint arXiv:1312.3005, 2013.

[5] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pages 770–778, 2016.

[6] Aidan N. Gomez, Mengye Ren, Raquel Urtasun and Roger B. Grosse. The Reversible Residual Network: Backpropagation Without Storing Activations. arXiv preprint arXiv:1707.04585

[7] Bo Chang, Lili Meng, Eldad Haber, Lars Ruthotto, David Begert and Elliot Holtham. Reversible Architectures for Arbitrarily Deep Residual Neural Networks. arXiv preprint arXiv:1709.03698

[8] Marcus, Mitchell P, Marcinkiewicz, Mary Ann, and Santorini, Beatrice. Building a large annotated corpus of english: The penn treebank. Computational linguistics, 19(2):313–330, 1993.

[9] `www.pytorch.org`

[10] J. Martens and I. Sutskever. Training deep and recurrent networks with Hessian-free optimization. In Neural networks: Tricks of the trade, pages 479–535. Springer, 2012.