Batch Normalized Recurrent Neural Networks

César Laurent *
Université de Montréal

Gabriel Pereyra *
University of Southern California

Philémon Brakel
Université de Montréal

Ying Zhang
Université de Montréal

Yoshua Bengio †
Université de Montréal

Abstract

Recurrent Neural Networks (RNNs) are powerful models for sequential data that have the potential to learn long-term dependencies. However, they are computationally expensive to train and difficult to parallelize. Recent work has shown that normalizing intermediate representations of neural networks can significantly improve convergence rates in feedforward neural networks [1]. In particular, batch normalization, which uses mini-batch statistics to standardize features, was shown to significantly reduce training time. In this paper, we show that applying batch normalization to the hidden-to-hidden transitions of our RNNs doesn't help the training procedure. We also show that when applied to the input-to-hidden transitions, batch normalization can lead to a faster convergence of the training criterion but doesn't seem to improve the generalization performance on both our language modelling and speech recognition tasks. All in all, applying batch normalization to RNNs turns out to be more challenging than applying it to feedforward networks, but certain variants of it can still be beneficial.

1 Introduction

Recurrent Neural Networks (RNNs) have received renewed interest due to their recent success in various domains, including speech recognition [2], machine translation [3, 4] and language modelling [5]. The so-called Long Short-Term Memory (LSTM) [6] type RNN has been particularly successful. Often, it seems beneficial to train deep architectures in which multiple RNNs are stacked on top of each other [2]. Unfortunately, the training cost for large datasets and deep architectures of stacked RNNs can be prohibitively high, often times an order of magnitude greater than simpler models like n-grams [7]. Because of this, recent work has explored methods for parallelizing RNNs across multiple graphics cards (GPUs). In [3], an LSTM type RNN was distributed layer-wise across multiple GPUs and in [8] a bidirectional RNN was distributed across time. However, due to the sequential nature of RNNs, it is difficult to achieve linear speed ups relative to the number of GPUs.

Another way to reduce training times is through a better conditioned optimization procedure. Standardizing or whitening of input data has long been known to improve the convergence of gradient-based optimization methods [9]. Extending this idea to multi-layered networks suggests that normalizing or whitening intermediate representations can similarly improve convergence. However, applying these transforms would be extremely costly. In [1], batch normalization was used to standardize intermediate representations by approximating the population statistics using sample-based approximations obtained from small subsets of the data, often called mini-batches, that are also used to obtain gradient approximations for stochastic gradient descent, the most commonly used optimization method for neural network training. It has also been shown that convergence can be improved even more by whitening intermediate representations instead of simply standardizing

*Equal contribution
†CIFAR Senior Fellow

them [10]. These methods reduced the training time of Convolutional Neural Networks (CNNs) by an order of magnitude and additionallly provided a regularization effect, leading to state-of-the-art results in object recognition on the ImageNet dataset [11]. In this paper, we explore how to leverage normalization in RNNs and show that training time can be reduced.

2 Batch Normalization

In optimization, feature standardization or whitening is a common procedure that has been shown to reduce convergence rates [9]. Extending the idea to deep neural networks, one can think of an arbitrary layer as receiving samples from a distribution that is shaped by the layer below. This distribution changes during the course of training, making any layer but the first responsible not only for learning a good representation but also for adapting to a changing input distribution. This distribution variation is termed *Internal Covariate Shift*, and reducing it is hypothesized to help the training procedure [1].

To reduce this internal covariate shift, we could whiten each layer of the network. However, this often turns out to be too computationally demanding. Batch normalization [1] approximates the whitening by standardizing the intermediate representations using the statistics of the current mini-batch. Given a mini-batch $\mathbf{x}$, we can calculate the sample mean and sample variance of each feature k along the mini-batch axis

$$\bar{\mathbf{x}}_k = \frac{1}{m}\sum_{i=1}^{m}\mathbf{x}_{i,k}, \tag{1}$$

$$\sigma_k^2 = \frac{1}{m}\sum_{i=1}^{m}(\mathbf{x}_{i,k} - \bar{\mathbf{x}}_k)^2, \tag{2}$$

where m is the size of the mini-batch. Using these statistics, we can standardize each feature as follows

$$\hat{\mathbf{x}}_k = \frac{\mathbf{x}_k - \bar{\mathbf{x}}_k}{\sqrt{\sigma_k^2 + \epsilon}}, \tag{3}$$

where ϵ is a small positive constant to improve numerical stability.

However, standardizing the intermediate activations reduces the representational power of the layer. To account for this, batch normalization introduces additional learnable parameters γ and β, which respectively scale and shift the data, leading to a layer of the form

$$BN(\mathbf{x}_k) = \gamma_k\hat{\mathbf{x}}_k + \beta_k. \tag{4}$$

By setting γ_k to σ_k and β_k to $\bar{x}_k$, the network can recover the original layer representation. So, for a standard feedforward layer in a neural network

$$\mathbf{y} = \phi(\mathbf{W}\mathbf{x} + \mathbf{b}), \tag{5}$$

where $\mathbf{W}$ is the weights matrix, $\mathbf{b}$ is the bias vector, $\mathbf{x}$ is the input of the layer and ϕ is an arbitrary activation function, batch normalization is applied as follows

$$\mathbf{y} = \phi(BN(\mathbf{W}\mathbf{x})). \tag{6}$$

Note that the bias vector has been removed, since its effect is cancelled by the standardization. Since the normalization is now part of the network, the back propagation procedure needs to be adapted to propagate gradients through the mean and variance computations as well.

At test time, we can't use the statistics of the mini-batch. Instead, we can estimate them by either forwarding several training mini-batches through the network and averaging their statistics, or by maintaining a running average calculated over each mini-batch seen during training.

3 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) extend Neural Networks to sequential data. Given an input sequence of vectors $(\mathbf{x}_1, \ldots, \mathbf{x}_T)$, they produce a sequence of hidden states $(\mathbf{h}_1, \ldots, \mathbf{h}_T)$, which are computed at time step t as follows

$$\mathbf{h}_t = \phi(\mathbf{W}_h \mathbf{h}_{t-1} + \mathbf{W}_x \mathbf{x}_t), \tag{7}$$

where $\mathbf{W}_h$ is the recurrent weight matrix, $\mathbf{W}_x$ is the input-to-hidden weight matrix, and ϕ is an arbitrary activation function.

If we have access to the whole input sequence, we can use information not only from the past time steps, but also from the future ones, allowing for bidirectional RNNs [12]

$$\overrightarrow{\mathbf{h}}_t = \phi(\overrightarrow{\mathbf{W}}_h \overrightarrow{\mathbf{h}}_{t-1} + \overrightarrow{\mathbf{W}}_x \mathbf{x}_t), \tag{8}$$

$$\overleftarrow{\mathbf{h}}_t = \phi(\overleftarrow{\mathbf{W}}_h \overleftarrow{\mathbf{h}}_{t+1} + \overleftarrow{\mathbf{W}}_x \mathbf{x}_t), \tag{9}$$

$$\mathbf{h}_t = [\overrightarrow{\mathbf{h}}_t : \overleftarrow{\mathbf{h}}_t], \tag{10}$$

where $[\mathbf{x} : \mathbf{y}]$ denotes the concatenation of $\mathbf{x}$ and $\mathbf{y}$. Finally, we can stack RNNs by using $\mathbf{h}$ as the input to another RNN, creating deeper architectures [13]

$$\mathbf{h}_t^l = \phi(\mathbf{W}_h \mathbf{h}_{t-1}^l + \mathbf{W}_x \mathbf{h}_t^{l-1}). \tag{11}$$

In vanilla RNNs, the activation function ϕ is usually a sigmoid function, such as the hyperbolic tangent. Training such networks is known to be particularly difficult, because of vanishing and exploding gradients [14].

3.1 Long Short-Term Memory

A commonly used recurrent structure is the Long Short-Term Memory (LSTM). It addresses the vanishing gradient problem commonly found in vanilla RNNs by incorporating gating functions into its state dynamics [6]. At each time step, an LSTM maintains a hidden vector $\mathbf{h}$ and a cell vector $\mathbf{c}$ responsible for controlling state updates and outputs. More concretely, we define the computation at time step t as follows [15]:

$$\mathbf{i}_t = \text{sigmoid}(\mathbf{W}_{hi} \mathbf{h}_{t-1} + \mathbf{W}_{xi} \mathbf{x}_t) \tag{12}$$

$$\mathbf{f}_t = \text{sigmoid}(\mathbf{W}_{hf} \mathbf{h}_{t-1} + \mathbf{W}_{hf} \mathbf{x}_t) \tag{13}$$

$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tanh(\mathbf{W}_{hc} \mathbf{h}_{t-1} + \mathbf{W}_{xc} \mathbf{x}_t) \tag{14}$$

$$\mathbf{o}_t = \text{sigmoid}(\mathbf{W}_{ho} \mathbf{h}_{t-1} + \mathbf{W}_{hx} \mathbf{x}_t + \mathbf{W}_{co} \mathbf{c}_t) \tag{15}$$

$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t) \tag{16}$$

where $\text{sigmoid}(\cdot)$ is the logistic sigmoid function, $\tanh$ is the hyperbolic tangent function, $\mathbf{W}_{h\cdot}$ are the recurrent weight matrices and $\mathbf{W}_{x\cdot}$ are the input-to-hiddent weight matrices. $\mathbf{i}_t$, $\mathbf{f}_t$ and $\mathbf{o}_t$ are respectively the input, forget and output gates, and $\mathbf{c}_t$ is the cell.

4 Batch Normalization for RNNs

From equation 6, an analogous way to apply batch normalization to an RNN would be as follows:

$$\mathbf{h}_t = \phi(BN(\mathbf{W}_h \mathbf{h}_{t-1} + \mathbf{W}_x \mathbf{x}_t)). \tag{17}$$

However, in our experiments, when batch normalization was applied in this fashion, it didn't help the training procedure (see appendix A for more details). Instead we propose to apply batch normalization only to the input-to-hidden transition ($\mathbf{W}_x \mathbf{x}_t$), i.e. as follows:

$$\mathbf{h}_t = \phi(\mathbf{W}_h \mathbf{h}_{t-1} + BN(\mathbf{W}_x \mathbf{x}_t)). \tag{18}$$

This idea is similar to the way dropout [16] can be applied to RNNs [17]: batch normalization is applied only on the vertical connections (i.e. from one layer to another) and not on the horizontal connections (i.e. within the recurrent layer). We use the same principle for LSTMs: batch normalization is only applied after multiplication with the input-to-hidden weight matrices $\mathbf{W}_{x\cdot}$.

Model	Train		Dev	
	FCE	FER	FCE	FER
BiRNN	0.95	0.28	**1.11**	**0.33**
BiRNN (BN)	**0.73**	**0.22**	1.19	0.34

Table 1: Best framewise cross entropy (FCE) and frame error rate (FER) on the training and development sets for both networks.

4.1 Frame-wise and Sequence-wise Normalization

In experiments where we don't have access to the future frames, like in language modelling where the goal is to predict the next character, we are forced to compute the normalization a each time step

$$\hat{\mathbf{x}}_{k,t} = \frac{\mathbf{x}_{k,t} - \bar{\mathbf{x}}_{k,t}}{\sqrt{\sigma_{k,t}^2 + \epsilon}}. \tag{19}$$

We'll refer to this as *frame-wise normalization*.

In applications like speech recognition, we usually have access to the entire sequences. However, those sequences may have variable length. Usually, when using mini-batches, the smaller sequences are padded with zeroes to match the size of the longest sequence of the mini-batch. In such setups we can't use frame-wise normalization, because the number of unpadded frames decreases along the time axis, leading to increasingly poorer statistics estimates. To solve this problem, we apply a sequence-wise normalization, where we compute the mean and variance of each feature along both the time and batch axis using

$$\bar{\mathbf{x}}_k = \frac{1}{n} \sum_{i=1}^{m} \sum_{t=1}^{T} \mathbf{x}_{i,t,k}, \tag{20}$$

$$\sigma_k^2 = \frac{1}{n} \sum_{i=1}^{m} \sum_{t=1}^{T} (\mathbf{x}_{i,t,k} - \bar{\mathbf{x}}_k)^2, \tag{21}$$

where T is the length of each sequence and n is the total number of unpadded frames in the mini-batch. We'll refer to this type of normalization as *sequence-wise normalization*.

5 Experiments

We ran experiments on a speech recognition task and a language modelling task. The models were implemented using Theano [18] and Blocks [19].

5.1 Speech Alignment Prediction

For the speech task, we used the Wall Street Journal (WSJ) [20] speech corpus. We used the si284 split as training set and evaluated our models on the dev93 development set. The raw audio was transformed into 40 dimensional log mel filter-banks (plus energy), with deltas and delta-deltas. As in [21], the forced alignments were generated from the Kaldi recipe tri4b, leading to 3546 clustered triphone states. Because of memory issues, we removed from the training set the sequences that were longer than 1300 frames (4.6% of the set), leading to a training set of 35746 sequences.

The baseline model (BL) is a stack of 5 bidirectional LSTM layers with 250 hidden units each, followed by a size 3546 softmax output layer. All the weights were initialized using the Glorot [22] scheme and all the biases were set to zero. For the batch normalized model (BN) we applied sequence-wise normalization to each LSTM of the baseline model. Both networks were trained using standard SGD with momentum, with a fixed learning rate of 1e-4 and a fixed momentum factor of 0.9. The mini-batch size is 24.

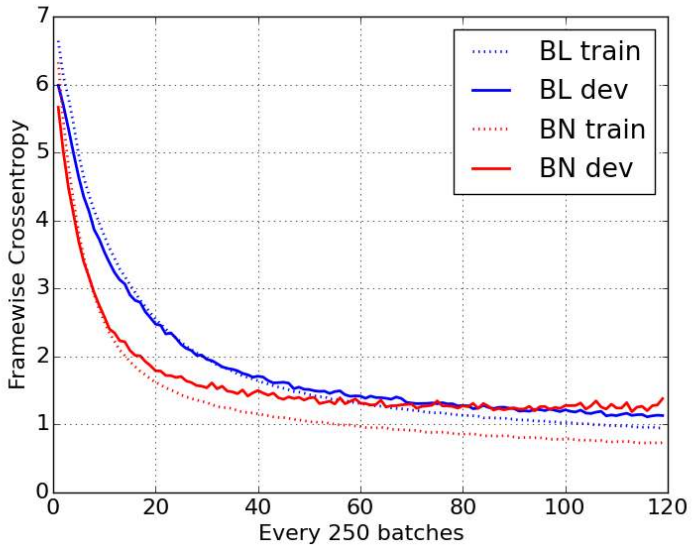

Figure 1: Frame-wise cross entropy on WSJ for the baseline (blue) and batch normalized (red) networks. The dotted lines are the training curves and the solid lines are the validation curves.

5.2 Language Modeling

We used the Penn Treebank (PTB) [23] corpus for our language modeling experiments. We use the standard split (929k training words, 73k validation words, and 82k test words) and vocabulary of 10k words. We train a small, medium and large LSTM as described in [17].

All models consist of two stacked LSTM layers and are trained with stochastic gradient descent (SGD) with a learning rate of 1 and a mini-batch size of 32.

The small LSTM has two layers of 200 memory cells, with parameters being initialized from a uniform distribution with range [-0.1, 0.1]. We back propagate across 20 time steps and the gradients are scaled according to the maximum norm of the gradients whenever the norm is greater than 10. We train for 15 epochs and halve the learning rate every epoch after the 6th.

The medium LSTM has a hidden size of 650 for both layers, with parameters being initialized from a uniform distribution with range [-0.05, 0.05]. We apply dropout with probability of 50% between all layers. We back propagate across 35 time steps and gradients are scaled according to the maximum norm of the gradients whenever the norm is greater than 5. We train for 40 epochs and divide the learning rate by 1.2 every epoch after the 6th.

The Large LSTM has two layers of 1500 memory cells, with parameters being initialized from a uniform distribution with range [-0.04, 0.04]. We apply dropout between all layers. We back propagate across 35 time steps and gradients are scaled according to the maximum norm of the gradients whenever the norm is greater than 5. We train for 55 epochs and divide the learning rate by 1.15 every epoch after the 15th.

6 Results and Discussion

Figure 1 shows the training and development framewise cross entropy curves for both networks of the speech experiments. As we can see, the batch normalized networks trains faster (at some points about twice as fast as the baseline), but overfits more. The best results, reported in table 1, are comparable to the ones obtained in [21].

Figure 2 shows the training and validation perplexity for the large LSTM network of the language experiment. We can also observe that the training is faster when we apply batch normalization to

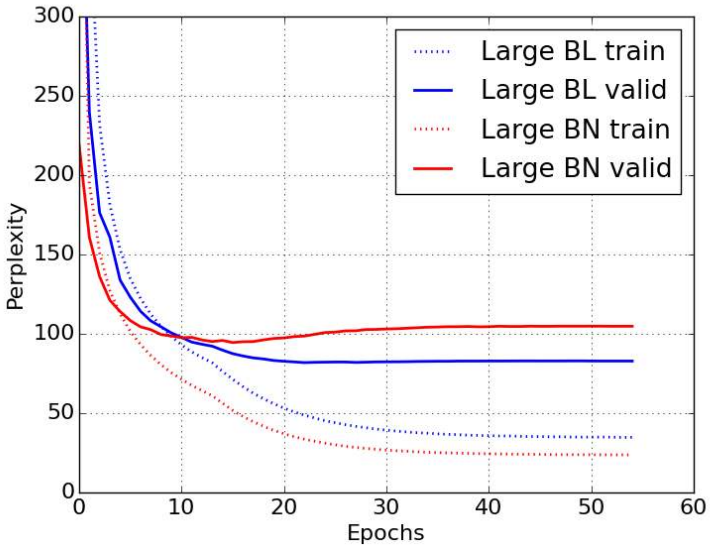

Figure 2: Large LSTM on Penn Treebank for the baseline (blue) and the batch normalized (red) networks. The dotted lines are the training curves and the solid lines are the validation curves.

Model	**Train**	**Valid**
Small LSTM	78.5	119.2
Small LSTM (BN)	62.5	120.9
Medium LSTM	49.1	89.0
Medium LSTM (BN)	41.0	90.6
Large LSTM	49.3	**81.8**
Large LSTM (BN)	**35.0**	97.4

Table 2: Best perplexity on training and development sets for LSTMs on Penn Treebank.

the network. However, it also overfits more than the baseline version. The best results are reported in table 2.

For both experiments we observed a faster training and a greater overfitting when using our version of batch normalization. This last effect is less prevalent in the speech experiment, perhaps because the training set is way bigger, or perhaps because the frame-wise normalization is less effective than the sequence-wise one. It can also be caused by the experimental setup: in the language modeling task we predict one character at a time, whereas we predict the whole sequence in the speech experiment.

Batch normalization also allows for higher learning rates in feedforward networks, however since we only applied batch normalization to parts of the network, higher learning rates didn't work well because they affected un-normalized parts as well.

Our experiments suggest that applying batch normalization to the input-to-hidden connections in RNNs can improve the conditioning of the optimization problem. Future directions include whitening input-to-hidden connections [10] and normalizing the hidden state instead of just a portion of the network.

Acknowledgments

Part of this work was funded by Samsung. We also want to thank Nervana Systems for providing GPUs.

References

[1] Sergey Ioffe and Christian Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *arXiv preprint arXiv:1502.03167*, 2015.

[2] Alan Graves, Abdel-rahman Mohamed, and Geoffrey Hinton, "Speech recognition with deep recurrent neural networks," in *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*. IEEE, 2013, pp. 6645–6649.

[3] Ilya Sutskever, Oriol Vinyals, and Quoc Le, "Sequence to sequence learning with neural networks," in *Advances in Neural Information Processing Systems*, 2014, pp. 3104–3112.

[4] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio, "Neural machine translation by jointly learning to align and translate," *arXiv preprint arXiv:1409.0473*, 2014.

[5] Tomáš Mikolov, "Statistical language models based on neural networks," *Presentation at Google, Mountain View, 2nd April*, 2012.

[6] Sepp Hochreiter and Jürgen Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[7] Will Williams, Niranjani Prasad, David Mrva, Tom Ash, and Tony Robinson, "Scaling recurrent neural network language models," *arXiv preprint arXiv:1502.00512*, 2015.

[8] Awni Hannun, Carl Case, Jared Casper, Bryan Catanzaro, Greg Diamos, Erich Elsen, Ryan Prenger, Sanjeev Satheesh, Shubho Sengupta, Adam Coates, et al., "Deepspeech: Scaling up end-to-end speech recognition," *arXiv preprint arXiv:1412.5567*, 2014.

[9] Yann A LeCun, Léon Bottou, Genevieve B Orr, and Klaus-Robert Müller, "Efficient backprop," in *Neural networks: Tricks of the trade*, pp. 9–48. Springer, 2012.

[10] Guillaume Desjardins, Karen Simonyan, Razvan Pascanu, and Koray Kavukcuoglu, "Natural neural networks," *arXiv preprint arXiv:1507.00210*, 2015.

[11] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision (IJCV)*, pp. 1–42, April 2015.

[12] Mike Schuster and Kuldip K Paliwal, "Bidirectional recurrent neural networks," *Signal Processing, IEEE Transactions on*, vol. 45, no. 11, pp. 2673–2681, 1997.

[13] Razvan Pascanu, Caglar Gulcehre, Kyunghyun Cho, and Yoshua Bengio, "How to construct deep recurrent neural networks," *arXiv preprint arXiv:1312.6026*, 2013.

[14] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio, "On the difficulty of training recurrent neural networks," *arXiv preprint arXiv:1211.5063*, 2012.

[15] Felix A Gers, Nicol N Schraudolph, and Jürgen Schmidhuber, "Learning precise timing with lstm recurrent networks," *The Journal of Machine Learning Research*, vol. 3, pp. 115–143, 2003.

[16] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[17] Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals, "Recurrent neural network regularization," *arXiv preprint arXiv:1409.2329*, 2014.

[18] Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, James Bergstra, Ian J. Goodfellow, Arnaud Bergeron, Nicolas Bouchard, and Yoshua Bengio, "Theano: new features and speed improvements," Deep Learning and Unsupervised Feature Learning NIPS 2012 Workshop, 2012.

[19] B. van Merriënboer, D. Bahdanau, V. Dumoulin, D. Serdyuk, D. Warde-Farley, J. Chorowski, and Y. Bengio, "Blocks and Fuel: Frameworks for deep learning," *ArXiv e-prints*, June 2015.

Model	Train	Valid
Best Baseline	1.05	1.10
Best Batch Norm	1.07	1.11

Table 3: Best frame-wise crossentropy for the best baseline network and for the best batch normalized one.

[20] Douglas B Paul and Janet M Baker, "The design for the wall street journal-based csr corpus," in *Proceedings of the workshop on Speech and Natural Language*. Association for Computational Linguistics, 1992, pp. 357–362.

[21] Alan Graves, Navdeep Jaitly, and Abdel-rahman Mohamed, "Hybrid speech recognition with deep bidirectional lstm," in *Automatic Speech Recognition and Understanding (ASRU), 2013 IEEE Workshop on*. IEEE, 2013, pp. 273–278.

[22] Xavier Glorot and Yoshua Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *International conference on artificial intelligence and statistics*, 2010, pp. 249–256.

[23] Mitchell P Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini, "Building a large annotated corpus of english: The penn treebank," *Computational linguistics*, vol. 19, no. 2, pp. 313–330, 1993.

A Experimentations with Normalization Inside the Recurrence

In our first experiments we investigated if batch normalization can be applied in the same way as in a feedforward network (equation 17). We tried it on a language modelling task on the PennTreebank dataset, where the goal was to predict the next characters of a fixed length sequence of 100 symbols.

The network is composed of a lookup table of dimension 250 followed by 3 layers of simple recurrent networks with 250 hidden units each. A dimension 50 softmax layer is added on the top. In the batch normalized networks, we apply batch normalization to the hidden-to-hidden transition, as in equation 17, meaning that we compute one mean and one variance for each of the 250 features at each time step. For inference, we also keep track of the statistics for each time step. However, we used the same γ and β for each time step.

The lookup table is randomly initialized using an isotropic Gaussian with zero mean and unit variance. All the other matrices of the network are initialized using the Glorot scheme [22] and all the bias are set to zero. We used SGD with momentum. We performed a random search over the learning rate (distributed in the range [0.0001, 1]), the momentum (with possible values of 0.5, 0.8, 0.9, 0.95, 0.995), and the batch size (32, 64 or 128). We let the experiment run for 20 epochs. A total of 52 experiments were performed.

In every experiment that we ran, the performances of batch normalized networks were always slightly worse than (or at best equivalent to) the baseline networks, except for the ones where the learning rate is too high and the baseline diverges while the batch normalized one is still able to train. Figure 3 shows an example of a working experiment. We observed that in practically all the experiments that converged, the normalization was actually harming the performance. Table 3 shows the results of the best baseline and batch normalized networks. We can observe that both best networks have similar performances. The settings for the best baseline are: learning rate 0.42, momentum 0.95, batch size 32. The settings for the best batch normalized network are: learning rate 3.71e-4, momentum 0.995, batch size 128.

Those results suggest that this way of applying batch normalization in the recurrent networks is not optimal. It seems that batch normalization hurts the training procedure. It may be due to the fact that we estimate new statistics at each time step, or because of the repeated application of γ and β during the recurrent procedure, which could lead to exploding or vanishing gradients. We will investigate more in depth what happens in the batch normalized networks, especially during the back-propagation.

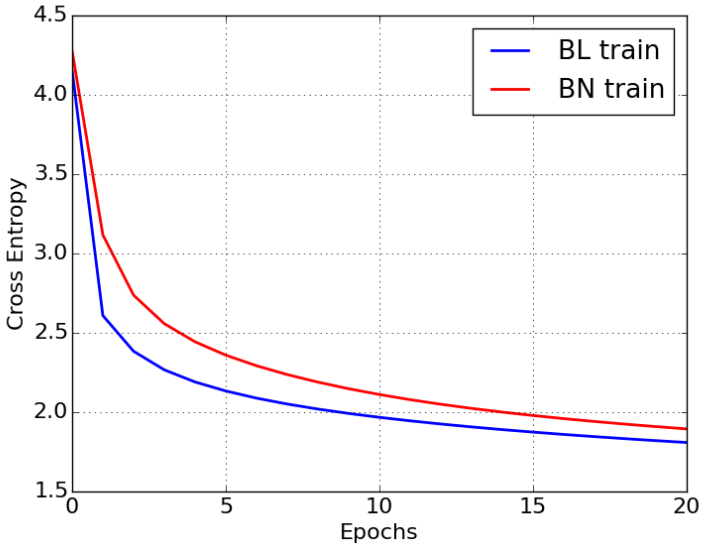

Figure 3: Typical training curves obtained during the grid search. The baseline network is in blue and batch normalized one in red. For this experiment, the hyper-parameters are: learning rate 7.8e-4, momentum 0.5, batch size 64.