Layer Normalization

Jimmy Lei Ba
University of Toronto
jimmy@psi.toronto.edu

Jamie Ryan Kiros
University of Toronto
rkiros@cs.toronto.edu

Geoffrey E. Hinton
University of Toronto
and Google Inc.
hinton@cs.toronto.edu

Abstract

Training state-of-the-art, deep neural networks is computationally expensive. One way to reduce the training time is to normalize the activities of the neurons. A recently introduced technique called batch normalization uses the distribution of the summed input to a neuron over a mini-batch of training cases to compute a mean and variance which are then used to normalize the summed input to that neuron on each training case. This significantly reduces the training time in feed-forward neural networks. However, the effect of batch normalization is dependent on the mini-batch size and it is not obvious how to apply it to recurrent neural networks. In this paper, we transpose batch normalization into layer normalization by computing the mean and variance used for normalization from all of the summed inputs to the neurons in a layer on a *single* training case. Like batch normalization, we also give each neuron its own adaptive bias and gain which are applied after the normalization but before the non-linearity. Unlike batch normalization, layer normalization performs exactly the same computation at training and test times. It is also straightforward to apply to recurrent neural networks by computing the normalization statistics separately at each time step. Layer normalization is very effective at stabilizing the hidden state dynamics in recurrent networks. Empirically, we show that layer normalization can substantially reduce the training time compared with previously published techniques.

1 Introduction

Deep neural networks trained with some version of Stochastic Gradient Descent have been shown to substantially outperform previous approaches on various supervised learning tasks in computer vision [Krizhevsky et al., 2012] and speech processing [Hinton et al., 2012]. But state-of-the-art deep neural networks often require many days of training. It is possible to speed-up the learning by computing gradients for different subsets of the training cases on different machines or splitting the neural network itself over many machines [Dean et al., 2012], but this can require a lot of communication and complex software. It also tends to lead to rapidly diminishing returns as the degree of parallelization increases. An orthogonal approach is to modify the computations performed in the forward pass of the neural net to make learning easier. Recently, batch normalization [Ioffe and Szegedy, 2015] has been proposed to reduce training time by including additional normalization stages in deep neural networks. The normalization standardizes each summed input using its mean and its standard deviation across the training data. Feedforward neural networks trained using batch normalization converge faster even with simple SGD. In addition to training time improvement, the stochasticity from the batch statistics serves as a regularizer during training.

Despite its simplicity, batch normalization requires running averages of the summed input statistics. In feed-forward networks with fixed depth, it is straightforward to store the statistics separately for each hidden layer. However, the summed inputs to the recurrent neurons in a recurrent neural network (RNN) often vary with the length of the sequence so applying batch normalization to RNNs appears to require different statistics for different time-steps. Furthermore, batch normaliza-

tion cannot be applied to online learning tasks or to extremely large distributed models where the minibatches have to be small.

This paper introduces layer normalization, a simple normalization method to improve the training speed for various neural network models. Unlike batch normalization, the proposed method directly estimates the normalization statistics from the summed inputs to the neurons within a hidden layer so the normalization does not introduce any new dependencies between training cases. We show that layer normalization works well for RNNs and improves both the training time and the generalization performance of several existing RNN models.

2 Background

A feed-forward neural network is a non-linear mapping from a input pattern $\mathbf{x}$ to an output vector y. Consider the l^{th} hidden layer in a deep feed-forward, neural network, and let a^l be the vector representation of the summed inputs to the neurons in that layer. The summed inputs are computed through a linear projection with the weight matrix W^l and the bottom-up inputs h^l given as follows:

$$a_i^l = {w_i^l}^\top h^l \qquad h_i^{l+1} = f(a_i^l + b_i^l) \tag{1}$$

where $f(\cdot)$ is an element-wise non-linear function and w_i^l is the incoming weights to the i^{th} hidden units and b_i^l is the scalar bias parameter. The parameters in the neural network are learnt using gradient-based optimization algorithms with the gradients being computed by back-propagation.

One of the challenges of deep learning is that the gradients with respect to the weights in one layer are highly dependent on the outputs of the neurons in the previous layer especially if these outputs change in a highly correlated way. Batch normalization [Ioffe and Szegedy, 2015] was proposed to reduce such undesirable "covariate shift". The method normalizes the summed inputs to each hidden unit over the training cases. Specifically, for the i^{th} summed input in the l^{th} layer, the batch normalization method rescales the summed inputs according to their variances under the distribution of the data

$$\bar{a}_i^l = \frac{g_i^l}{\sigma_i^l}\left(a_i^l - \mu_i^l\right) \qquad \mu_i^l = \mathop{\mathbb{E}}_{\mathbf{x}\sim P(\mathbf{x})}\left[a_i^l\right] \qquad \sigma_i^l = \sqrt{\mathop{\mathbb{E}}_{\mathbf{x}\sim P(\mathbf{x})}\left[\left(a_i^l - \mu_i^l\right)^2\right]} \tag{2}$$

where $\bar{a}_i^l$ is normalized summed inputs to the i^{th} hidden unit in the l^{th} layer and g_i is a gain parameter scaling the normalized activation before the non-linear activation function. Note the expectation is under the whole training data distribution. It is typically impractical to compute the expectations in Eq. (2) exactly, since it would require forward passes through the whole training dataset with the current set of weights. Instead, μ and σ are estimated using the empirical samples from the current mini-batch. This puts constraints on the size of a mini-batch and it is hard to apply to recurrent neural networks.

3 Layer normalization

We now consider the layer normalization method which is designed to overcome the drawbacks of batch normalization.

Notice that changes in the output of one layer will tend to cause highly correlated changes in the summed inputs to the next layer, especially with ReLU units whose outputs can change by a lot. This suggests the "covariate shift" problem can be reduced by fixing the mean and the variance of the summed inputs within each layer. We, thus, compute the layer normalization statistics over all the hidden units in the same layer as follows:

$$\mu^l = \frac{1}{H}\sum_{i=1}^{H} a_i^l \qquad \sigma^l = \sqrt{\frac{1}{H}\sum_{i=1}^{H}\left(a_i^l - \mu^l\right)^2} \tag{3}$$

where H denotes the number of hidden units in a layer. The difference between Eq. (2) and Eq. (3) is that under layer normalization, all the hidden units in a layer share the same normalization terms μ and σ, but different training cases have different normalization terms. Unlike batch normalization, layer normaliztion does not impose any constraint on the size of a mini-batch and it can be used in the pure online regime with batch size 1.

3.1 Layer normalized recurrent neural networks

The recent sequence to sequence models [Sutskever et al., 2014] utilize compact recurrent neural networks to solve sequential prediction problems in natural language processing. It is common among the NLP tasks to have different sentence lengths for different training cases. This is easy to deal with in an RNN because the same weights are used at every time-step. But when we apply batch normalization to an RNN in the obvious way, we need to to compute and store separate statistics for each time step in a sequence. This is problematic if a test sequence is longer than any of the training sequences. Layer normalization does not have such problem because its normalization terms depend only on the summed inputs to a layer at the current time-step. It also has only one set of gain and bias parameters shared over all time-steps.

In a standard RNN, the summed inputs in the recurrent layer are computed from the current input $\mathbf{x}^t$ and previous vector of hidden states $\mathbf{h}^{t-1}$ which are computed as $\mathbf{a}^t = W_{hh}h^{t-1} + W_{xh}\mathbf{x}^t$. The layer normalized recurrent layer re-centers and re-scales its activations using the extra normalization terms similar to Eq. (3):

$$\mathbf{h}^t = f\left[\frac{\mathbf{g}}{\sigma^t} \odot \left(\mathbf{a}^t - \mu^t\right) + \mathbf{b}\right] \qquad \mu^t = \frac{1}{H}\sum_{i=1}^{H} a_i^t \qquad \sigma^t = \sqrt{\frac{1}{H}\sum_{i=1}^{H}\left(a_i^t - \mu^t\right)^2} \tag{4}$$

where W_{hh} is the recurrent hidden to hidden weights and W_{xh} are the bottom up input to hidden weights. $\odot$ is the element-wise multiplication between two vectors. $\mathbf{b}$ and $\mathbf{g}$ are defined as the bias and gain parameters of the same dimension as $\mathbf{h}^t$.

In a standard RNN, there is a tendency for the average magnitude of the summed inputs to the recurrent units to either grow or shrink at every time-step, leading to exploding or vanishing gradients. In a layer normalized RNN, the normalization terms make it invariant to re-scaling all of the summed inputs to a layer, which results in much more stable hidden-to-hidden dynamics.

4 Related work

Batch normalization has been previously extended to recurrent neural networks [Laurent et al., 2015, Amodei et al., 2015, Cooijmans et al., 2016]. The previous work [Cooijmans et al., 2016] suggests the best performance of recurrent batch normalization is obtained by keeping independent normalization statistics for each time-step. The authors show that initializing the gain parameter in the recurrent batch normalization layer to 0.1 makes significant difference in the final performance of the model. Our work is also related to weight normalization [Salimans and Kingma, 2016]. In weight normalization, instead of the variance, the L2 norm of the incoming weights is used to normalize the summed inputs to a neuron. Applying either weight normalization or batch normalization using expected statistics is equivalent to have a different parameterization of the original feed-forward neural network. Re-parameterization in the ReLU network was studied in the Path-normalized SGD [Neyshabur et al., 2015]. Our proposed layer normalization method, however, is not a re-parameterization of the original neural network. The layer normalized model, thus, has different invariance properties than the other methods, that we will study in the following section.

5 Analysis

In this section, we investigate the invariance properties of different normalization schemes.

5.1 Invariance under weights and data transformations

The proposed layer normalization is related to batch normalization and weight normalization. Although, their normalization scalars are computed differently, these methods can be summarized as normalizing the summed inputs a_i to a neuron through the two scalars μ and σ. They also learn an adaptive bias b and gain g for each neuron after the normalization.

$$h_i = f(\frac{g_i}{\sigma_i}\left(a_i - \mu_i\right) + b_i) \tag{5}$$

Note that for layer normalization and batch normalization, μ and σ is computed according to Eq. 2 and 3. In weight normalization, μ is 0, and $\sigma = \|w\|_2$.

	Weight matrix re-scaling	Weight matrix re-centering	Weight vector re-scaling	Dataset re-scaling	Dataset re-centering	Single training case re-scaling
Batch norm	Invariant	No	Invariant	Invariant	Invariant	No
Weight norm	Invariant	No	Invariant	No	No	No
Layer norm	Invariant	Invariant	No	Invariant	No	Invariant

Table 1: Invariance properties under the normalization methods.

Table 1 highlights the following invariance results for three normalization methods.

Weight re-scaling and re-centering: First, observe that under batch normalization and weight normalization, any re-scaling to the incoming weights w_i of a single neuron has no effect on the normalized summed inputs to a neuron. To be precise, under batch and weight normalization, if the weight vector is scaled by δ, the two scalar μ and σ will also be scaled by δ. The normalized summed inputs stays the same before and after scaling. So the batch and weight normalization are invariant to the re-scaling of the weights. Layer normalization, on the other hand, is not invariant to the individual scaling of the single weight vectors. Instead, layer normalization is invariant to scaling of the entire weight matrix and invariant to a shift to all of the incoming weights in the weight matrix. Let there be two sets of model parameters θ, θ' whose weight matrices W and W' differ by a scaling factor δ and all of the incoming weights in W' are also shifted by a constant vector $\boldsymbol{\gamma}$, that is $W' = \delta W + \mathbf{1}\boldsymbol{\gamma}^\top$. Under layer normalization, the two models effectively compute the same output:

$$\begin{aligned}\mathbf{h}' =& f(\frac{\mathbf{g}}{\sigma'}\left(W'\mathbf{x} - \mu'\right) + \mathbf{b}) = f(\frac{\mathbf{g}}{\sigma'}\left((\delta W + \mathbf{1}\boldsymbol{\gamma}^\top)\mathbf{x} - \mu'\right) + \mathbf{b}) \\ =& f(\frac{\mathbf{g}}{\sigma}\left(W\mathbf{x} - \mu\right) + \mathbf{b}) = \mathbf{h}. \end{aligned} \tag{6}$$

Notice that if normalization is only applied to the input before the weights, the model will not be invariant to re-scaling and re-centering of the weights.

Data re-scaling and re-centering: We can show that all the normalization methods are invariant to re-scaling the dataset by verifying that the summed inputs of neurons stays constant under the changes. Furthermore, layer normalization is invariant to re-scaling of individual training cases, because the normalization scalars μ and σ in Eq. (3) only depend on the current input data. Let $\mathbf{x}'$ be a new data point obtained by re-scaling $\mathbf{x}$ by δ. Then we have,

$$h_i' = f(\frac{g_i}{\sigma'}\left(w_i^\top \mathbf{x}' - \mu'\right) + b_i) = f(\frac{g_i}{\delta\sigma}\left(\delta w_i^\top \mathbf{x} - \delta\mu\right) + b_i) = h_i. \tag{7}$$

It is easy to see re-scaling individual data points does not change the model's prediction under layer normalization. Similar to the re-centering of the weight matrix in layer normalization, we can also show that batch normalization is invariant to re-centering of the dataset.

5.2 Geometry of parameter space during learning

We have investigated the invariance of the model's prediction under re-centering and re-scaling of the parameters. Learning, however, can behave very differently under different parameterizations, even though the models express the same underlying function. In this section, we analyze learning behavior through the geometry and the manifold of the parameter space. We show that the normalization scalar σ can implicitly reduce learning rate and makes learning more stable.

5.2.1 Riemannian metric

The learnable parameters in a statistical model form a smooth manifold that consists of all possible input-output relations of the model. For models whose output is a probability distribution, a natural way to measure the separation of two points on this manifold is the Kullback-Leibler divergence between their model output distributions. Under the KL divergence metric, the parameter space is a Riemannian manifold.

The curvature of a Riemannian manifold is entirely captured by its Riemannian metric, whose quadratic form is denoted as ds^2. That is the infinitesimal distance in the tangent space at a point in the parameter space. Intuitively, it measures the changes in the model output from the parameter space along a tangent direction. The Riemannian metric under KL was previously studied [Amari, 1998] and was shown to be well approximated under second order Taylor expansion using the Fisher

information matrix:

$$ds^2 = \mathrm{D}_{\mathrm{KL}}\big[P(y \,|\, \mathbf{x};\ \theta) \| P(y \,|\, \mathbf{x};\ \theta + \delta)\big] \approx \frac{1}{2}\delta^\top F(\theta)\delta, \tag{8}$$

$$F(\theta) = \mathop{\mathbb{E}}_{\mathbf{x}\sim P(\mathbf{x}), y\sim P(y\,|\,\mathbf{x})}\left[\frac{\partial \log P(y \,|\, \mathbf{x};\ \theta)}{\partial \theta} \frac{\partial \log P(y \,|\, \mathbf{x};\ \theta)}{\partial \theta}^\top\right], \tag{9}$$

where, δ is a small change to the parameters. The Riemannian metric above presents a geometric view of parameter spaces. The following analysis of the Riemannian metric provides some insight into how normalization methods could help in training neural networks.

5.2.2 The geometry of normalized generalized linear models

We focus our geometric analysis on the generalized linear model. The results from the following analysis can be easily applied to understand deep neural networks with block-diagonal approximation to the Fisher information matrix, where each block corresponds to the parameters for a single neuron.

A generalized linear model (GLM) can be regarded as parameterizing an output distribution from the exponential family using a weight vector w and bias scalar b. To be consistent with the previous sections, the log likelihood of the GLM can be written using the summed inputs a as the following:

$$\log P(y \,|\, \mathbf{x};\ w, b) = \frac{(a+b)y - \eta(a+b)}{\phi} + c(y, \phi), \tag{10}$$

$$\mathbb{E}[y \,|\, \mathbf{x}] = f(a+b) = f(w^\top \mathbf{x} + b), \ \ \mathrm{Var}[y \,|\, \mathbf{x}] = \phi f'(a+b), \tag{11}$$

where, $f(\cdot)$ is the transfer function that is the analog of the non-linearity in neural networks, $f'(\cdot)$ is the derivative of the transfer function, $\eta(\cdot)$ is a real valued function and $c(\cdot)$ is the log partition function. ϕ is a constant that scales the output variance. Assume a H-dimensional output vector $\mathbf{y} = [y_1, y_2, \cdots, y_H]$ is modeled using H independent GLMs and $\log P(\mathbf{y} \,|\, \mathbf{x};\ W, \mathbf{b}) = \sum_{i=1}^{H} \log P(y_i \,|\, \mathbf{x};\ w_i, b_i)$. Let W be the weight matrix whose rows are the weight vectors of the individual GLMs, $\mathbf{b}$ denote the bias vector of length H and $\mathrm{vec}(\cdot)$ denote the Kronecker vector operator. The Fisher information matrix for the multi-dimensional GLM with respect to its parameters $\theta = [w_1^\top, b_1, \cdots, w_H^\top, b_H]^\top = \mathrm{vec}([W, \mathbf{b}]^\top)$ is simply the expected Kronecker product of the data features and the output covariance matrix:

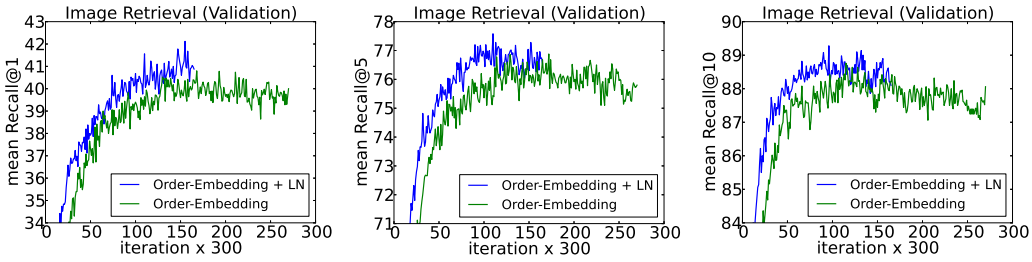
$$F(\theta) = \mathop{\mathbb{E}}_{\mathbf{x}\sim P(\mathbf{x})}\left[\frac{\mathrm{Cov}[\mathbf{y} \,|\, \mathbf{x}]}{\phi^2} \otimes \begin{bmatrix}\mathbf{x}\mathbf{x}^\top & \mathbf{x} \\ \mathbf{x}^\top & 1\end{bmatrix}\right]. \tag{12}$$

We obtain normalized GLMs by applying the normalization methods to the summed inputs a in the original model through μ and σ. Without loss of generality, we denote $\bar{F}$ as the Fisher information matrix under the normalized multi-dimensional GLM with the additional gain parameters $\theta = \mathrm{vec}([W, \mathbf{b}, \mathbf{g}]^\top)$:

$$\bar{F}(\theta) = \begin{bmatrix}\bar{F}_{11} & \cdots & \bar{F}_{1H} \\ \vdots & \ddots & \vdots \\ \bar{F}_{H1} & \cdots & \bar{F}_{HH}\end{bmatrix}, \quad \bar{F}_{ij} = \mathop{\mathbb{E}}_{\mathbf{x}\sim P(\mathbf{x})}\left[\frac{\mathrm{Cov}[y_i,\ y_j \ |\, \mathbf{x}]}{\phi^2} \begin{bmatrix}\frac{g_i g_j}{\sigma_i \sigma_j}\chi_i \chi_j^\top & \chi_i \frac{g_i}{\sigma_i} & \chi_i \frac{g_i (a_j - \mu_j)}{\sigma_i \sigma_j} \\ \chi_j^\top \frac{g_j}{\sigma_j} & 1 & \frac{a_j - \mu_j}{\sigma_j} \\ \chi_j^\top \frac{g_j (a_i - \mu_i)}{\sigma_i \sigma_j} & \frac{a_i - \mu_i}{\sigma_i} & \frac{(a_i - \mu_i)(a_j - \mu_j)}{\sigma_i \sigma_j}\end{bmatrix}\right] \tag{13}$$

$$\chi_i = \mathbf{x} - \frac{\partial \mu_i}{\partial w_i} - \frac{a_i - \mu_i}{\sigma_i}\frac{\partial \sigma_i}{\partial w_i}. \tag{14}$$

Implicit learning rate reduction through the growth of the weight vector: Notice that, comparing to standard GLM, the block $\bar{F}_{ij}$ along the weight vector w_i direction is scaled by the gain parameters and the normalization scalar σ_i. If the norm of the weight vector w_i grows twice as large, even though the model's output remains the same, the Fisher information matrix will be different. The curvature along the w_i direction will change by a factor of $\frac{1}{2}$ because the σ_i will also be twice as large. As a result, for the same parameter update in the normalized model, the norm of the weight vector effectively controls the learning rate for the weight vector. During learning, it is harder to change the orientation of the weight vector with large norm. The normalization methods, therefore,



(a) Recall@1 (b) Recall@5 (c) Recall@10

Figure 1: Recall@K curves using order-embeddings with and without layer normalization.

MSCOCO								
	Caption Retrieval				Image Retrieval			
Model	**R@1**	**R@5**	**R@10**	**Mean *r***	**R@1**	**R@5**	**R@10**	**Mean *r***
Sym [Vendrov et al., 2016]	45.4		88.7	5.8	36.3		85.8	9.0
OE [Vendrov et al., 2016]	46.7		88.9	5.7	37.9		85.9	8.1
OE (ours)	46.6	79.3	89.1	5.2	37.8	73.6	85.7	7.9
OE + LN	**48.5**	**80.6**	**89.8**	**5.1**	**38.9**	**74.3**	**86.3**	**7.6**

Table 2: Average results across 5 test splits for caption and image retrieval. **R@K** is Recall@K (high is good). **Mean *r*** is the mean rank (low is good). Sym corresponds to the symmetric baseline while OE indicates order-embeddings.

have an implicit "early stopping" effect on the weight vectors and help to stabilize learning towards convergence.

Learning the magnitude of incoming weights: In normalized models, the magnitude of the incoming weights is explicitly parameterized by the gain parameters. We compare how the model output changes between updating the gain parameters in the normalized GLM and updating the magnitude of the equivalent weights under original parameterization during learning. The direction along the gain parameters in $\bar{F}$ captures the geometry for the magnitude of the incoming weights. We show that Riemannian metric along the magnitude of the incoming weights for the standard GLM is scaled by the norm of its input, whereas learning the gain parameters for the batch normalized and layer normalized models depends only on the magnitude of the prediction error. Learning the magnitude of incoming weights in the normalized model is therefore, more robust to the scaling of the input and its parameters than in the standard model. See Appendix for detailed derivations.

6 Experimental results

We perform experiments with layer normalization on 6 tasks, with a focus on recurrent neural networks: image-sentence ranking, question-answering, contextual language modelling, generative modelling, handwriting sequence generation and MNIST classification. Unless otherwise noted, the default initialization of layer normalization is to set the adaptive gains to 1 and the biases to 0 in the experiments.

6.1 Order embeddings of images and language

In this experiment, we apply layer normalization to the recently proposed order-embeddings model of Vendrov et al. [2016] for learning a joint embedding space of images and sentences. We follow the same experimental protocol as Vendrov et al. [2016] and modify their publicly available code to incorporate layer normalization [1] which utilizes Theano [Team et al., 2016]. Images and sentences from the Microsoft COCO dataset [Lin et al., 2014] are embedded into a common vector space, where a GRU [Cho et al., 2014] is used to encode sentences and the outputs of a pre-trained VGG ConvNet [Simonyan and Zisserman, 2015] (10-crop) are used to encode images. The order-embedding model represents images and sentences as a 2-level partial ordering and replaces the cosine similarity scoring function used in Kiros et al. [2014] with an asymmetric one.

[1] `https://github.com/ivendrov/order-embedding`

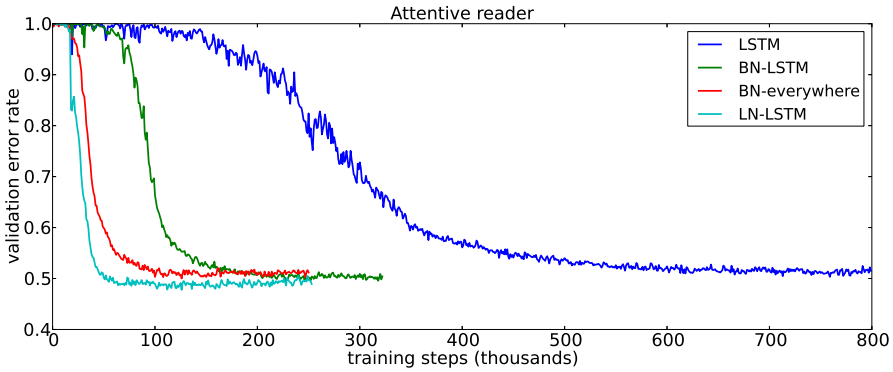


Figure 2: Validation curves for the attentive reader model. BN results are taken from [Cooijmans et al., 2016].

We trained two models: the baseline order-embedding model as well as the same model with layer normalization applied to the GRU. After every 300 iterations, we compute Recall@K (R@K) values on a held out validation set and save the model whenever R@K improves. The best performing models are then evaluated on 5 separate test sets, each containing 1000 images and 5000 captions, for which the mean results are reported. Both models use Adam [Kingma and Ba, 2014] with the same initial hyperparameters and both models are trained using the same architectural choices as used in Vendrov et al. [2016]. We refer the reader to the appendix for a description of how layer normalization is applied to GRU.

Figure 1 illustrates the validation curves of the models, with and without layer normalization. We plot R@1, R@5 and R@10 for the image retrieval task. We observe that layer normalization offers a per-iteration speedup across all metrics and converges to its best validation model in 60% of the time it takes the baseline model to do so. In Table 2, the test set results are reported from which we observe that layer normalization also results in improved generalization over the original model. The results we report are state-of-the-art for RNN embedding models, with only the structure-preserving model of Wang et al. [2016] reporting better results on this task. However, they evaluate under different conditions (1 test set instead of the mean over 5) and are thus not directly comparable.

6.2 Teaching machines to read and comprehend

In order to compare layer normalization to the recently proposed recurrent batch normalization [Cooijmans et al., 2016], we train an unidirectional attentive reader model on the CNN corpus both introduced by Hermann et al. [2015]. This is a question-answering task where a query description about a passage must be answered by filling in a blank. The data is anonymized such that entities are given randomized tokens to prevent degenerate solutions, which are consistently permuted during training and evaluation. We follow the same experimental protocol as Cooijmans et al. [2016] and modify their public code to incorporate layer normalization [2] which uses Theano [Team et al., 2016]. We obtained the pre-processed dataset used by Cooijmans et al. [2016] which differs from the original experiments of Hermann et al. [2015] in that each passage is limited to 4 sentences. In Cooijmans et al. [2016], two variants of recurrent batch normalization are used: one where BN is only applied to the LSTM while the other applies BN everywhere throughout the model. In our experiment, we only apply layer normalization within the LSTM.

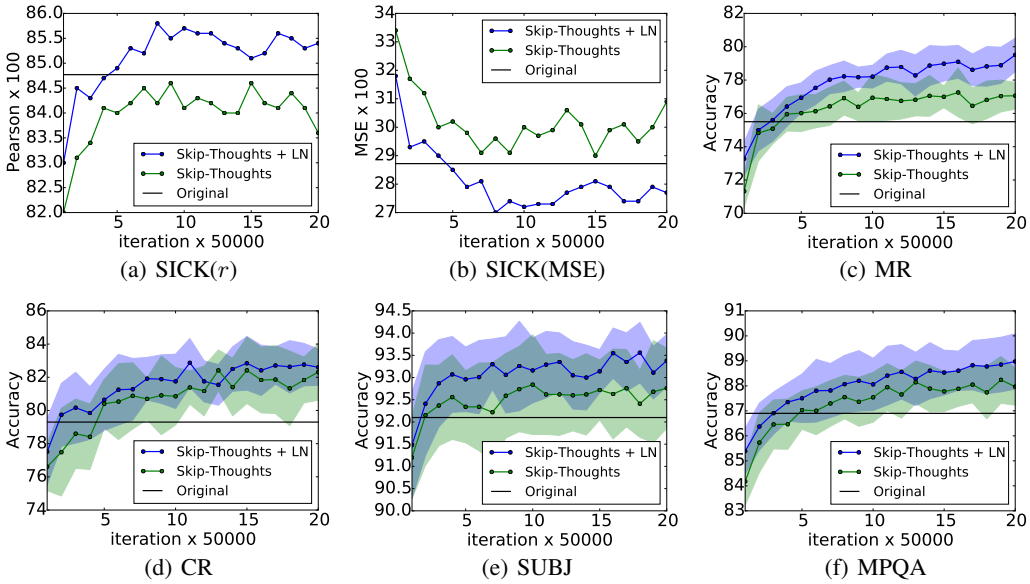
The results of this experiment are shown in Figure 2. We observe that layer normalization not only trains faster but converges to a better validation result over both the baseline and BN variants. In Cooijmans et al. [2016], it is argued that the scale parameter in BN must be carefully chosen and is set to 0.1 in their experiments. We experimented with layer normalization for both 1.0 and 0.1 scale initialization and found that the former model performed significantly better. This demonstrates that layer normalization is not sensitive to the initial scale in the same way that recurrent BN is. [3]

6.3 Skip-thought vectors

Skip-thoughts [Kiros et al., 2015] is a generalization of the skip-gram model [Mikolov et al., 2013] for learning unsupervised distributed sentence representations. Given contiguous text, a sentence is

[2] `https://github.com/cooijmanstim/Attentive_reader/tree/bn`

[3] We only produce results on the validation set, as in the case of Cooijmans et al. [2016]



(a) SICK(r) (b) SICK(MSE) (c) MR

(d) CR (e) SUBJ (f) MPQA

Figure 3: Performance of skip-thought vectors with and without layer normalization on downstream tasks as a function of training iterations. The original lines are the reported results in [Kiros et al., 2015]. Plots with error use 10-fold cross validation. Best seen in color.

Method	**SICK**(r)	**SICK**(ρ)	**SICK**(MSE)	**MR**	**CR**	**SUBJ**	**MPQA**
Original [Kiros et al., 2015]	0.848	0.778	0.287	75.5	79.3	92.1	86.9
Ours	0.842	0.767	0.298	77.3	81.8	92.6	87.9
Ours + LN	0.854	0.785	0.277	**79.5**	82.6	93.4	89.0
Ours + LN †	**0.858**	**0.788**	**0.270**	79.4	**83.1**	**93.7**	**89.3**

Table 3: Skip-thoughts results. The first two evaluation columns indicate Pearson and Spearman correlation, the third is mean squared error and the remaining indicate classification accuracy. Higher is better for all evaluations except MSE. Our models were trained for 1M iterations with the exception of (†) which was trained for 1 month (approximately 1.7M iterations)

encoded with a encoder RNN and decoder RNNs are used to predict the surrounding sentences. Kiros et al. [2015] showed that this model could produce generic sentence representations that perform well on several tasks without being fine-tuned. However, training this model is time-consuming, requiring several days of training in order to produce meaningful results.

In this experiment we determine to what effect layer normalization can speed up training. Using the publicly available code of Kiros et al. [2015] [4], we train two models on the BookCorpus dataset [Zhu et al., 2015]: one with and one without layer normalization. These experiments are performed with Theano [Team et al., 2016]. We adhere to the experimental setup used in Kiros et al. [2015], training a 2400-dimensional sentence encoder with the same hyperparameters. Given the size of the states used, it is conceivable layer normalization would produce slower per-iteration updates than without. However, we found that provided CNMeM [5] is used, there was no significant difference between the two models. We checkpoint both models after every 50,000 iterations and evaluate their performance on five tasks: semantic-relatedness (SICK) [Marelli et al., 2014], movie review sentiment (MR) [Pang and Lee, 2005], customer product reviews (CR) [Hu and Liu, 2004], subjectivity/objectivity classification (SUBJ) [Pang and Lee, 2004] and opinion polarity (MPQA) [Wiebe et al., 2005]. We plot the performance of both models for each checkpoint on all tasks to determine whether the performance rate can be improved with LN.

The experimental results are illustrated in Figure 3. We observe that applying layer normalization results both in speedup over the baseline as well as better final results after 1M iterations are performed as shown in Table 3. We also let the model with layer normalization train for a total of a month, resulting in further performance gains across all but one task. We note that the performance

[4] `https://github.com/ryankiros/skip-thoughts`

[5] `https://github.com/NVIDIA/cnmem`

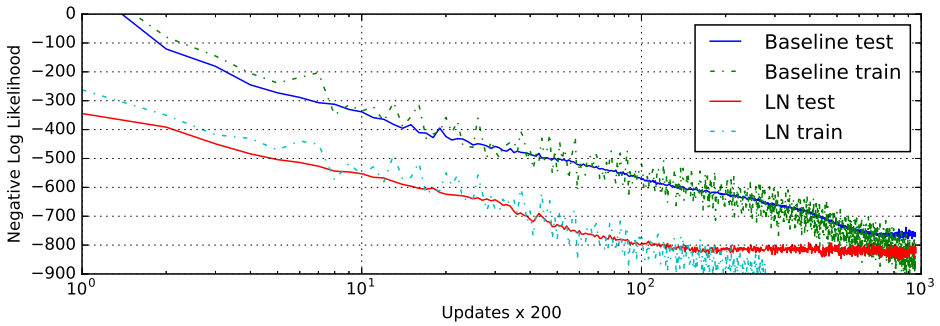


Figure 5: Handwriting sequence generation model negative log likelihood with and without layer normalization. The models are trained with mini-batch size of 8 and sequence length of 500,

differences between the original reported results and ours are likely due to the fact that the publicly available code does not condition at each timestep of the decoder, where the original model does.

6.4 Modeling binarized MNIST using DRAW

We also experimented with the generative modeling on the MNIST dataset. Deep Recurrent Attention Writer (DRAW) [Gregor et al., 2015] has previously achieved the state-of-the-art performance on modeling the distribution of MNIST digits. The model uses a differential attention mechanism and a recurrent neural network to sequentially generate pieces of an image. We evaluate the effect of layer normalization on a DRAW model using 64 glimpses and 256 LSTM hidden units. The model is trained with the default setting of Adam [Kingma and Ba, 2014] optimizer and the minibatch size of 128. Previous publications on binarized MNIST have used various training protocols to generate their datasets. In this experiment, we used the fixed binarization from Larochelle and Murray [2011]. The dataset has been split into 50,000 training, 10,000 validation and 10,000 test images.

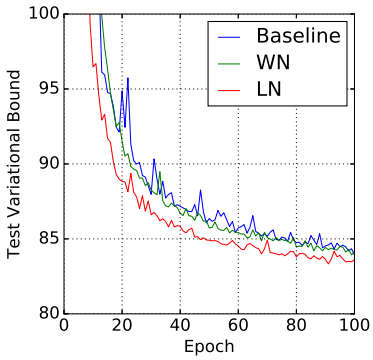


Figure 4: DRAW model test negative log likelihood with and without layer normalization.

Figure 4 shows the test variational bound for the first 100 epoch. It highlights the speedup benefit of applying layer normalization that the layer normalized DRAW converges almost twice as fast than the baseline model. After 200 epoches, the baseline model converges to a variational log likelihood of 82.36 nats on the test data and the layer normalization model obtains 82.09 nats.

6.5 Handwriting sequence generation

The previous experiments mostly examine RNNs on NLP tasks whose lengths are in the range of 10 to 40. To show the effectiveness of layer normalization on longer sequences, we performed handwriting generation tasks using the IAM Online Handwriting Database [Liwicki and Bunke, 2005]. IAM-OnDB consists of handwritten lines collected from 221 different writers. When given the input character string, the goal is to predict a sequence of x and y pen co-ordinates of the corresponding handwriting line on the whiteboard. There are, in total, 12179 handwriting line sequences. The input string is typically more than 25 characters and the average handwriting line has a length around 700.

We used the same model architecture as in Section (5.2) of Graves [2013]. The model architecture consists of three hidden layers of 400 LSTM cells, which produce 20 bivariate Gaussian mixture components at the output layer, and a size 3 input layer. The character sequence was encoded with one-hot vectors, and hence the window vectors were size 57. A mixture of 10 Gaussian functions was used for the window parameters, requiring a size 30 parameter vector. The total number of weights was increased to approximately 3.7M. The model is trained using mini-batches of size 8 and the Adam [Kingma and Ba, 2014] optimizer.

The combination of small mini-batch size and very long sequences makes it important to have very stable hidden dynamics. Figure 5 shows that layer normalization converges to a comparable log likelihood as the baseline model but is much faster.

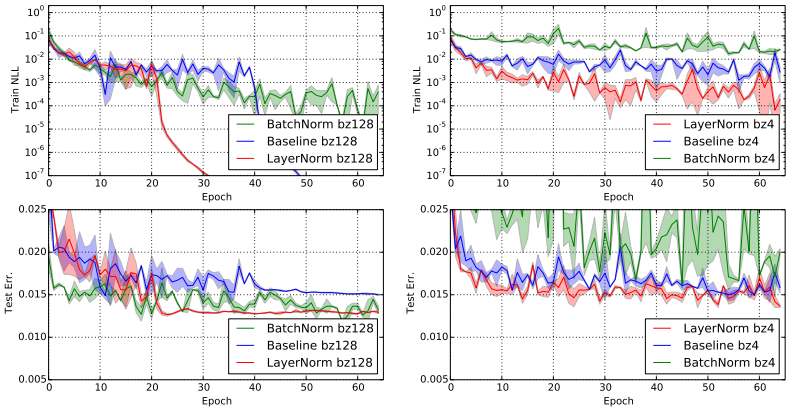


Figure 6: Permutation invariant MNIST 784-1000-1000-10 model negative log likelihood and test error with layer normalization and batch normalization. (Left) The models are trained with batch-size of 128. (Right) The models are trained with batch-size of 4.

6.6 Permutation invariant MNIST

In addition to RNNs, we investigated layer normalization in feed-forward networks. We show how layer normalization compares with batch normalization on the well-studied permutation invariant MNIST classification problem. From the previous analysis, layer normalization is invariant to input re-scaling which is desirable for the internal hidden layers. But this is unnecessary for the logit outputs where the prediction confidence is determined by the scale of the logits. We only apply layer normalization to the fully-connected hidden layers that excludes the last softmax layer.

All the models were trained using 55000 training data points and the Adam [Kingma and Ba, 2014] optimizer. For the smaller batch-size, the variance term for batch normalization is computed using the unbiased estimator. The experimental results from Figure 6 highlight that layer normalization is robust to the batch-sizes and exhibits a faster training convergence comparing to batch normalization that is applied to all layers.

6.7 Convolutional Networks

We have also experimented with convolutional neural networks. In our preliminary experiments, we observed that layer normalization offers a speedup over the baseline model without normalization, but batch normalization outperforms the other methods. With fully connected layers, all the hidden units in a layer tend to make similar contributions to the final prediction and re-centering and re-scaling the summed inputs to a layer works well. However, the assumption of similar contributions is no longer true for convolutional neural networks. The large number of the hidden units whose receptive fields lie near the boundary of the image are rarely turned on and thus have very different statistics from the rest of the hidden units within the same layer. We think further research is needed to make layer normalization work well in ConvNets.

7 Conclusion

In this paper, we introduced layer normalization to speed-up the training of neural networks. We provided a theoretical analysis that compared the invariance properties of layer normalization with batch normalization and weight normalization. We showed that layer normalization is invariant to per training-case feature shifting and scaling.

Empirically, we showed that recurrent neural networks benefit the most from the proposed method especially for long sequences and small mini-batches.

Acknowledgments

This research was funded by grants from NSERC, CFI, and Google.

References

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *NIPS*, 2012.

Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N Sainath, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE*, 2012.

Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. Large scale distributed deep networks. In *NIPS*, 2012.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *ICML*, 2015.

Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

César Laurent, Gabriel Pereyra, Philémon Brakel, Ying Zhang, and Yoshua Bengio. Batch normalized recurrent neural networks. *arXiv preprint arXiv:1510.01378*, 2015.

Dario Amodei, Rishita Anubhai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Jingdong Chen, Mike Chrzanowski, Adam Coates, Greg Diamos, et al. Deep speech 2: End-to-end speech recognition in english and mandarin. *arXiv preprint arXiv:1512.02595*, 2015.

Tim Cooijmans, Nicolas Ballas, César Laurent, and Aaron Courville. Recurrent batch normalization. *arXiv preprint arXiv:1603.09025*, 2016.

Tim Salimans and Diederik P Kingma. Weight normalization: A simple reparameterization to accelerate training of deep neural networks. *arXiv preprint arXiv:1602.07868*, 2016.

Behnam Neyshabur, Ruslan R Salakhutdinov, and Nati Srebro. Path-sgd: Path-normalized optimization in deep neural networks. In *Advances in Neural Information Processing Systems*, pages 2413–2421, 2015.

Shun-Ichi Amari. Natural gradient works efficiently in learning. *Neural computation*, 1998.

Ivan Vendrov, Ryan Kiros, Sanja Fidler, and Raquel Urtasun. Order-embeddings of images and language. *ICLR*, 2016.

The Theano Development Team, Rami Al-Rfou, Guillaume Alain, Amjad Almahairi, Christof Angermueller, Dzmitry Bahdanau, Nicolas Ballas, Frédéric Bastien, Justin Bayer, Anatoly Belikov, et al. Theano: A python framework for fast computation of mathematical expressions. *arXiv preprint arXiv:1605.02688*, 2016.

Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco: Common objects in context. *ECCV*, 2014.

Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *EMNLP*, 2014.

Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *ICLR*, 2015.

Ryan Kiros, Ruslan Salakhutdinov, and Richard S Zemel. Unifying visual-semantic embeddings with multimodal neural language models. *arXiv preprint arXiv:1411.2539*, 2014.

D. Kingma and J. L. Ba. Adam: a method for stochastic optimization. *ICLR*, 2014. arXiv:1412.6980.

Liwei Wang, Yin Li, and Svetlana Lazebnik. Learning deep structure-preserving image-text embeddings. *CVPR*, 2016.

Karl Moritz Hermann, Tomas Kocisky, Edward Grefenstette, Lasse Espeholt, Will Kay, Mustafa Suleyman, and Phil Blunsom. Teaching machines to read and comprehend. In *NIPS*, 2015.

Ryan Kiros, Yukun Zhu, Ruslan R Salakhutdinov, Richard Zemel, Raquel Urtasun, Antonio Torralba, and Sanja Fidler. Skip-thought vectors. In *NIPS*, 2015.

Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

Yukun Zhu, Ryan Kiros, Rich Zemel, Ruslan Salakhutdinov, Raquel Urtasun, Antonio Torralba, and Sanja Fidler. Aligning books and movies: Towards story-like visual explanations by watching movies and reading books. In *ICCV*, 2015.

Marco Marelli, Luisa Bentivogli, Marco Baroni, Raffaella Bernardi, Stefano Menini, and Roberto Zamparelli. Semeval-2014 task 1: Evaluation of compositional distributional semantic models on full sentences through semantic relatedness and textual entailment. *SemEval-2014*, 2014.

Bo Pang and Lillian Lee. Seeing stars: Exploiting class relationships for sentiment categorization with respect to rating scales. In *ACL*, pages 115–124, 2005.

Minqing Hu and Bing Liu. Mining and summarizing customer reviews. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, 2004.

Bo Pang and Lillian Lee. A sentimental education: Sentiment analysis using subjectivity summarization based on minimum cuts. In *ACL*, 2004.

Janyce Wiebe, Theresa Wilson, and Claire Cardie. Annotating expressions of opinions and emotions in language. *Language resources and evaluation*, 2005.

K. Gregor, I. Danihelka, A. Graves, and D. Wierstra. DRAW: a recurrent neural network for image generation. arXiv:1502.04623, 2015.

Hugo Larochelle and Iain Murray. The neural autoregressive distribution estimator. In *AISTATS*, volume 6, page 622, 2011.

Marcus Liwicki and Horst Bunke. Iam-ondb-an on-line english sentence database acquired from handwritten text on a whiteboard. In *ICDAR*, 2005.

Alex Graves. Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850*, 2013.

Supplementary Material

Application of layer normalization to each experiment

This section describes how layer normalization is applied to each of the papers' experiments. For notation convenience, we define layer normalization as a function mapping $LN : \mathbb{R}^D \rightarrow \mathbb{R}^D$ with two set of adaptive parameters, gains $\boldsymbol{\alpha}$ and biases $\boldsymbol{\beta}$:

$$LN(\mathbf{z}; \boldsymbol{\alpha}, \boldsymbol{\beta}) = \frac{(\mathbf{z} - \mu)}{\sigma} \odot \boldsymbol{\alpha} + \boldsymbol{\beta}, \tag{15}$$

$$\mu = \frac{1}{D}\sum_{i=1}^{D} z_i, \quad \sigma = \sqrt{\frac{1}{D}\sum_{i=1}^{D}(z_i - \mu)^2}, \tag{16}$$

where, z_i is the i^{th} element of the vector $\mathbf{z}$.

Teaching machines to read and comprehend and handwriting sequence generation

The basic LSTM equations used for these experiment are given by:

$$\begin{pmatrix} \mathbf{f}_t \\ \mathbf{i}_t \\ \mathbf{o}_t \\ \mathbf{g}_t \end{pmatrix} = \mathbf{W}_h \mathbf{h}_{t-1} + \mathbf{W}_x \mathbf{x}_t + b \tag{17}$$

$$\mathbf{c}_t = \sigma(\mathbf{f}_t) \odot \mathbf{c}_{t-1} + \sigma(\mathbf{i}_t) \odot \tanh(\mathbf{g}_t) \tag{18}$$

$$\mathbf{h}_t = \sigma(\mathbf{o}_t) \odot \tanh(\mathbf{c}_t) \tag{19}$$

The version that incorporates layer normalization is modified as follows:

$$\begin{pmatrix} \mathbf{f}_t \\ \mathbf{i}_t \\ \mathbf{o}_t \\ \mathbf{g}_t \end{pmatrix} = LN(\mathbf{W}_h \mathbf{h}_{t-1}; \boldsymbol{\alpha}_1, \boldsymbol{\beta}_1) + LN(\mathbf{W}_x \mathbf{x}_t; \boldsymbol{\alpha}_2, \boldsymbol{\beta}_2) + b \tag{20}$$

$$\mathbf{c}_t = \sigma(\mathbf{f}_t) \odot \mathbf{c}_{t-1} + \sigma(\mathbf{i}_t) \odot \tanh(\mathbf{g}_t) \tag{21}$$

$$\mathbf{h}_t = \sigma(\mathbf{o}_t) \odot \tanh(LN(\mathbf{c}_t; \boldsymbol{\alpha}_3, \boldsymbol{\beta}_3)) \tag{22}$$

where $\boldsymbol{\alpha}_i, \boldsymbol{\beta}_i$ are the additive and multiplicative parameters, respectively. Each $\boldsymbol{\alpha}_i$ is initialized to a vector of zeros and each $\boldsymbol{\beta}_i$ is initialized to a vector of ones.

Order embeddings and skip-thoughts

These experiments utilize a variant of gated recurrent unit which is defined as follows:

$$\begin{pmatrix} \mathbf{z}_t \\ \mathbf{r}_t \end{pmatrix} = \mathbf{W}_h \mathbf{h}_{t-1} + \mathbf{W}_x \mathbf{x}_t \tag{23}$$

$$\hat{\mathbf{h}}_t = \tanh(\mathbf{W}\mathbf{x}_t + \sigma(\mathbf{r}_t) \odot (\mathbf{U}\mathbf{h}_{t-1})) \tag{24}$$

$$\mathbf{h}_t = (1 - \sigma(\mathbf{z}_t))\mathbf{h}_{t-1} + \sigma(\mathbf{z}_t)\hat{\mathbf{h}}_t \tag{25}$$

Layer normalization is applied as follows:

$$\begin{pmatrix} \mathbf{z}_t \\ \mathbf{r}_t \end{pmatrix} = LN(\mathbf{W}_h \mathbf{h}_{t-1}; \boldsymbol{\alpha}_1, \boldsymbol{\beta}_1) + LN(\mathbf{W}_x \mathbf{x}_t; \boldsymbol{\alpha}_2, \boldsymbol{\beta}_2) \tag{26}$$

$$\hat{\mathbf{h}}_t = \tanh(LN(\mathbf{W}\mathbf{x}_t; \boldsymbol{\alpha}_3, \boldsymbol{\beta}_3) + \sigma(\mathbf{r}_t) \odot LN(\mathbf{U}\mathbf{h}_{t-1}; \boldsymbol{\alpha}_4, \boldsymbol{\beta}_4)) \tag{27}$$

$$\mathbf{h}_t = (1 - \sigma(\mathbf{z}_t))\mathbf{h}_{t-1} + \sigma(\mathbf{z}_t)\hat{\mathbf{h}}_t \tag{28}$$

just as before, $\boldsymbol{\alpha}_i$ is initialized to a vector of zeros and each $\boldsymbol{\beta}_i$ is initialized to a vector of ones.

Modeling binarized MNIST using DRAW

The layer norm is only applied to the output of the LSTM hidden states in this experiment:

The version that incorporates layer normalization is modified as follows:

$$\begin{pmatrix} \mathbf{f}_t \\ \mathbf{i}_t \\ \mathbf{o}_t \\ \mathbf{g}_t \end{pmatrix} = \mathbf{W}_h \mathbf{h}_{t-1} + \mathbf{W}_x \mathbf{x}_t + b \tag{29}$$

$$\mathbf{c}_t = \sigma(\mathbf{f}_t) \odot \mathbf{c}_{t-1} + \sigma(\mathbf{i}_t) \odot \tanh(\mathbf{g}_t) \tag{30}$$

$$\mathbf{h}_t = \sigma(\mathbf{o}_t) \odot \tanh(LN(\mathbf{c}_t; \boldsymbol{\alpha}, \boldsymbol{\beta})) \tag{31}$$

where $\boldsymbol{\alpha}, \boldsymbol{\beta}$ are the additive and multiplicative parameters, respectively. $\boldsymbol{\alpha}$ is initialized to a vector of zeros and $\boldsymbol{\beta}$ is initialized to a vector of ones.

Learning the magnitude of incoming weights

We now compare how gradient descent updates changing magnitude of the equivalent weights between the normalized GLM and original parameterization. The magnitude of the weights are explicitly parameterized using the gain parameter in the normalized model. Assume there is a gradient update that changes norm of the weight vectors by δ_g. We can project the gradient updates to the weight vector for the normal GLM. The KL metric, ie how much the gradient update changes the model prediction, for the normalized model depends only on the magnitude of the prediction error. Specifically,

under batch normalization:

$$ds^2 = \frac{1}{2} \operatorname{vec}([0, 0, \delta_g]^\top)^\top \bar{F} (\operatorname{vec}([W, \mathbf{b}, \mathbf{g}]^\top) \operatorname{vec}([0, 0, \delta_g]^\top) = \frac{1}{2} \delta_g^\top \mathop{\mathbb{E}}_{\mathbf{x} \sim P(\mathbf{x})} \left[\frac{\operatorname{Cov}[\mathbf{y} \mid \mathbf{x}]}{\phi^2} \right] \delta_g. \tag{32}$$

Under layer normalization:

$$\begin{aligned} ds^2 =& \frac{1}{2} \operatorname{vec}([0, 0, \delta_g]^\top)^\top \bar{F} (\operatorname{vec}([W, \mathbf{b}, \mathbf{g}]^\top) \operatorname{vec}([0, 0, \delta_g]^\top) \\ =& \frac{1}{2} \delta_g^\top \frac{1}{\phi^2} \mathop{\mathbb{E}}_{\mathbf{x} \sim P(\mathbf{x})} \begin{bmatrix} \operatorname{Cov}(y_1, y_1 \mid \mathbf{x}) \frac{(a_1 - \mu)^2}{\sigma^2} & \cdots & \operatorname{Cov}(y_1, y_H \mid \mathbf{x}) \frac{(a_1 - \mu)(a_H - \mu)}{\sigma^2} \\ \vdots & \ddots & \vdots \\ \operatorname{Cov}(y_H, y_1 \mid \mathbf{x}) \frac{(a_H - \mu)(a_1 - \mu)}{\sigma^2} & \cdots & \operatorname{Cov}(y_H, y_H \mid \mathbf{x}) \frac{(a_H - \mu)^2}{\sigma^2} \end{bmatrix} \delta_g \end{aligned} \tag{33}$$

Under weight normalization:

$$\begin{aligned} ds^2 =& \frac{1}{2} \operatorname{vec}([0, 0, \delta_g]^\top)^\top \bar{F} (\operatorname{vec}([W, \mathbf{b}, \mathbf{g}]^\top) \operatorname{vec}([0, 0, \delta_g]^\top) \\ =& \frac{1}{2} \delta_g^\top \frac{1}{\phi^2} \mathop{\mathbb{E}}_{\mathbf{x} \sim P(\mathbf{x})} \begin{bmatrix} \operatorname{Cov}(y_1, y_1 \mid \mathbf{x}) \frac{a_1^2}{\|w_1\|_2^2} & \cdots & \operatorname{Cov}(y_1, y_H \mid \mathbf{x}) \frac{a_1 a_H}{\|w_1\|_2 \|w_H\|_2} \\ \vdots & \ddots & \vdots \\ \operatorname{Cov}(y_H, y_1 \mid \mathbf{x}) \frac{a_H a_1}{\|w_H\|_2 \|w_1\|_2} & \cdots & \operatorname{Cov}(y_H, y_H \mid \mathbf{x}) \frac{a_H^2}{\|w_H\|_2^2} \end{bmatrix} \delta_g. \end{aligned} \tag{34}$$

Whereas, the KL metric in the standard GLM is related to its activities $a_i = w_i^\top \mathbf{x}$, that is depended on both its current weights and input data. We project the gradient updates to the gain parameter $\delta_g i$ of the i^{th} neuron to its weight vector as $\delta_{gi} \frac{w_i}{\|w_i\|_2}$ in the standard GLM model:

$$\begin{aligned} & \frac{1}{2} \operatorname{vec}([\delta_{gi} \frac{w_i}{\|w_i\|_2}, 0, \delta_{gj} \frac{w_j}{\|w_i\|_2}, 0]^\top)^\top F([w_i^\top, b_i, w_j^\top, b_j]^\top) \operatorname{vec}([\delta_{gi} \frac{w_i}{\|w_i\|_2}, 0, \delta_{gj} \frac{w_j}{\|w_j\|_2}, 0]^\top) \\ =& \frac{\delta_{gi} \delta_{gj}}{2\phi^2} \mathop{\mathbb{E}}_{\mathbf{x} \sim P(\mathbf{x})} \left[\operatorname{Cov}(y_i, y_j \mid \mathbf{x}) \frac{a_i a_j}{\|w_i\|_2 \|w_j\|_2} \right] \end{aligned} \tag{35}$$

The batch normalized and layer normalized models are therefore more robust to the scaling of the input and its parameters than the standard model.