Neural Processing Letters (2024) 56:78
https://doi.org/10.1007/s11063-024-11442-1

# Breaking Time Invariance: Assorted-Time Normalization for RNNs

Cole Pospisil[1,2] · Vasily Zadorozhnyy[1] · Qiang Ye[1]

Accepted: 20 October 2023

## Abstract

Methods such as Layer Normalization (LN) and Batch Normalization have proven to be effective in improving the training of Recurrent Neural Networks (RNNs). However, existing methods normalize using only the instantaneous information at one particular time step, and the result of the normalization is a preactivation state with a time-independent distribution. This implementation fails to account for certain temporal differences inherent in the inputs and the architecture of RNNs. Since these networks share weights across time steps, it may also be desirable to account for the connections between time steps in the normalization scheme. In this paper, we propose a normalization method called Assorted-Time Normalization (ATN), which preserves information from multiple consecutive time steps and normalizes using them. This setup allows us to introduce longer time dependencies into the traditional normalization methods without introducing any new trainable parameters. We present theoretical derivations for the gradient propagation and prove the weight scaling invariance property. Our experiments applying ATN to LN demonstrate consistent improvement on various tasks, such as Adding, Copying, and Denoise Problems and Language Modeling Problems.

**Keywords** Normalization methods · ATN · Layer Normalization · LN · LSTM

Cole Pospisil and Vasily Zadorozhnyy contributed equally to this work.

✉ Qiang Ye
qye3@uky.edu

Cole Pospisil
pospisilcole@gmail.com

Vasily Zadorozhnyy
vasily.zadorozhnyy@gmail.com

[1] Mathematics Department, University of Kentucky, 719 Patterson Office Tower, Lexington, KY 40506, USA

[2] NSWC Crane, 300 Highway 361, Crane, IN 47522, USA

Published online: 01 March 2024

## 1 Introduction

The Recurrent Neural Network (RNN) [8, 20], and variants such as Long Short Term Memory (LSTM) [6] or Gated Recurrent Unit (GRU) [2, 11, 18], are some of the core architectures used for modeling time-series data in Deep Learning today. While LSTMs and GRUs are effective in avoiding problems with vanishing gradients, all of these recurrent models are still subject to issues with exploding gradients, as well as over-fitting. One of the most successful ideas that have been introduced over the years is the normalization of RNNs using methods such as Layer Normalization (LN) [1] and Batch Normalization (BN) [3, 9]. These methods recenter and rescale the preactivation information using the statistics of that time step. This allows for the norm of the model's states and gradients to be controlled, which speeds up training and prevents exploding gradients.

While these normalization methods have been successful, their applications to RNNs do not involve adaptations to some of the primary characteristics of this class of models, namely that the variation across time imparts usable information. For example, the LN or BN models are invariant to the scaling in the input at any time step and are, therefore, independent of the norm of the input vector at each time step. Depending on the applications, this may have devastating consequences. Additionally, LN and BN produce a preactivation state with a distribution that is invariant across time. Such time invariance properties may affect the architectural structure of RNN's ability to exploit the temporal dependencies fully. Since RNNs share weights across time steps, it would be quite natural to introduce this dependency into the normalization method as well. An attempted version of this involving averaging statistics across time was mentioned in [3] but was unsuccessful and was presented without much detail. It appears that simply averaging over every time step is an overcorrection that makes the statistics susceptible to diluted averages and loses effectiveness further into the sequence. Instead, we argue that by collecting the mean and variance across a smaller subsequence, one can gain the benefits of these time dependencies without overly weakening the impact of a single time step.

In this paper, we propose a normalization method called Assorted-Time Normalization (ATN), which preserves information from multiple consecutive time steps and normalizes using them. Our ATN method can be combined with other normalization methods, such as LN and BN, that normalize input information along some dimensions but not time. It maintains a short-term memory of the previous $k$ time steps, which allows it to account for the temporal dependencies in a way in which previous methods were incapable. We use that memory to calculate the statistics with respect to which we normalize, giving us an output that has a controlled mean and variance while still being capable of changing between time steps. By using just a limited subsequence at each point in time, we can avoid the problems that come from using all or none of the sequences and find the length best suited to the dataset. Since this process adds a time component to the normalization method, it is adapting without the introduction of any new learnable parameters.

We present theoretical derivations for the gradient propagation and prove the weight scaling invariance property. Our experiments demonstrate consistent improvement using our method on various tasks, such as Adding, Copying, and Denoise Problems as well as Language Modeling Problems. Our code is available at https://github.com/vasily789/atn.

## 2 Related Work

One of the earliest attempts to use some normalization technique throughout model layers was Batch Normalization (BN) [9]. It was proposed for Fully Connected (FC) and Convolutional (CNN) Neural Networks to normalize network activations across the batch dimension. BN is known often to provide a more stable and accelerated training regimen while improving generalizations. The Instance Normalization (IN) [23] method, contrary to BN, acts like contrast normalization and has primarily been used for image-containing datasets. The paper points out that the output stylized images should not rely on the contrast of the input image content, and hence normalizing the instances helps. The Group Normalization (GN) [24] method, which is primarily used for CNNs, normalizes a 3D feature in a convolutional layer by dividing its channels into groups and then normalizing the features in the group in all three dimensions. Although these normalization methods can be applied at each time step to a recurrent neural network with sequential data, there have been no known successful implementations yet.

Consider the typical structure of an RNN, also known as an RNN cell:

$$h^{(t)} = f\left(W_h h^{(t-1)} + W_x x^{(t)} + \beta_h\right) \tag{1}$$

$$y^{(t)} = W_y h^{(t)} + \beta_y \tag{2}$$

where $f$ is a nonlinear activation function applied entrywise, $W_h$ is the hidden-to-hidden weight, $W_x$ is the input-to-hidden weight, $W_y$ is the hidden-to-output weight, $\beta_h$ is the hidden state bias, $\beta_y$ is the output bias, $x^{(t)}$ is the $t^{th}$ entry in the input sequence, $h^{(t)}$ is the $t^{(th)}$ hidden state, and $y^{(t)}$ is the $t^{(th)}$ entry in the output sequence.

The Recurrent Batch Normalization [3] method applies BN to the hidden-to-hidden and memory cell parts of the LSTM model, which aims to reduce the internal covariate shift between consecutive time steps. Salimans and Kingma [21] proposed a Weight Normalization (WN) method. Their idea lies in decoupling the magnitude from the direction of the weight vector to change the parameters of the network, which helps speed up learning. Unfortunately, WN appears not widely used in practice due to its limited stability compared to BN [5].

Layer Normalization (LN) was proposed in Ba et al. [1] to normalize activations along the hidden dimension for both FC networks and RNNs and has since become very popular in RNNs. LN normalizes the preactivation state as follows:

$$h^{(t)} = f\left(LN\left(W_h h^{(t-1)}\right) + LN\left(W_x x^{(t)}\right) + \beta_h\right) \tag{3}$$

$$y^{(t)} = LN\left(W_y h^{(t)}\right) + \beta_y \tag{4}$$

where the LN operator is defined by

$$\mu_t = \frac{1}{n}\sum_{i=1}^{n} a_i^{(t)} \quad \sigma_t^2 = \frac{1}{n}\sum_{i=1}^{n}\left(a_i^{(t)} - \mu_t\right)^2 \tag{5}$$

$$y^{(t)} = LN(a^{(t)}; \gamma, \beta) = \gamma \odot \frac{a^{(t)} - \mu_t}{\sqrt{\sigma_t^2 + \varepsilon}} + \beta \tag{6}$$

with $\gamma$ and $\beta$ being trainable gain and bias parameters which we omitted from the $LN$ operator notation in Eqs. 3 and 4 to allow for cleaner notation, and $\varepsilon$ is a small value included to prevent division by zero.

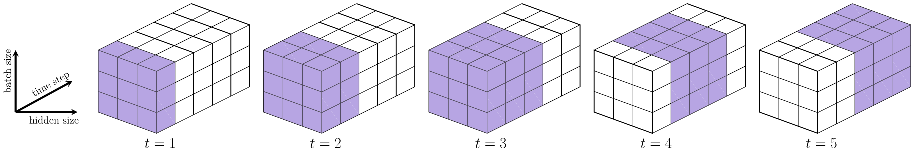


**Fig. 1** Illustration of the ATN method combined with LN using $k = 3$ time steps: consider the preactivation state tensor in $\mathbb{R}^{5\times3\times3}$. At $t = 1$, we use a standard LN; at $t = 2$, we normalize using information from time steps $1, 2$; at $t = 3$ ATN method uses information from time step $t = 1, 2, 3$; at $t = 4$, we normalize with respect to time steps $t = 2, 3, 4$; and so on after that

Such a setup helps eliminate the BN batch dependency and simplifies the application to RNNs.

More recently, Adaptive Normalization (AdaNorm) [25] made a thorough analysis of LN and concluded that the rescaling and recentering factors, $\gamma$ and $\beta$ in (6), are not as essential as the backward gradients of the mean and variance inside of the LN method. In addition, they proposed a new method, AdaNorm, which replaces weight and bias with some new transformation function. The key observation of this paper is that all the existing normalization methods do not account for the temporal variations for RNNs. Normalization at each time point may leverage the unique properties of the RNN architecture and provide improved performance similar to how Instance Norm and Group Norm did with CNNs.

## 3 Assorted Time Normalization

One undesirable property of the adaptation of LN to RNNs is that the statistics for the normalization are calculated at each time step, resulting in a post-normalization state which has mean and variance that are invariant across time. This prevents the model from effectively representing the shifting distributions across time that might be critical in modeling sequential data. For example, the normalization $LN\left(W_x x^{(t)}\right)$ in (3) is invariant to scaling in $x^{(t)}$, which restricts the model from learning the changing norm of $x^{(t)}$. This may be mitigated by including a bias in the linear term, which is often used in implementations; see Sect. 5.1 for more discussion. Most of the above discussions also apply to BN.

We propose a new normalization method to break this time invariance. Consider a sequence $\mathbf{a} = \left\{a^{(t)}\right\} \subset \mathbb{R}^n$ produced in an RNN, such as the preactivation state that we wish to normalize. At time step $t$ of the RNN, we maintain a memory of the previous $k$ entries, $\mathbf{a}_k^{(t)} = \left\{a^{(t-k+1)}, \ldots, a^{(t-1)}, a^{(t)}\right\} \subset \mathbf{a}$, in the normalization layer, using this extended set to compute the mean and variance to be used for normalization. This can be combined with other normalization methods. Combining with Layer Normalization, for example, these statistics are calculated at time-step $t$ for a sequence of $n$-dimensional inputs as follows:

$$\mu_{t,k} = \frac{1}{nk} \sum_{j=0}^{k-1} \sum_{s=1}^{n} a_s^{(t-j)} \tag{7}$$

$$\sigma_{t,k}^2 = \frac{1}{nk} \sum_{j=0}^{k-1} \sum_{s=1}^{n} \left(a_s^{(t-j)} - \mu_{t,k}\right)^2 \tag{8}$$

See Fig. 1 for a visual depiction of our method. The computational cost of LN-ATN scales linearly with $k$ as a multiple of the normal cost of Layer Normalization.

Once these statistics are calculated, we then normalize only the current term $a^{(t)}$ and optionally recenter and rescale using $\gamma$ and $\beta$, two trainable parameters shared across time while adding a small epsilon to the variance to prevent division by zero, similar to the LN method in (6).

$$y^{(t)} = ATN(\mathbf{a}_k^{(t)}; \gamma, \beta) := \gamma \odot \frac{a^{(t)} - \mu_{t,k}}{\sqrt{\sigma_{t,k}^2 + \varepsilon}} + \beta \tag{9}$$

This differs from the process in (6) in that we include multiple time steps in our statistic calculations, giving us a double sum instead of the single one for LN. This definition of the statistics is more stable in time, at least for large $k$, changing modestly at each time step with only one term in the set being replaced. This results in a normalized output that is not expected to have a uniform mean and variance across time steps. We argue that this is desirable for sequential problems. Having this potential for variation allows for the model to account for changing norms of the inputs across the sequence, providing additional information about the distribution that is lost with previous methods.

We may consider using all previous terms in the sequence to compute the statistics, but this causes variation in the weight of information between early and later time steps. Since the first few steps have fewer terms to normalize over, the effect in each step is heightened, while at the end of the sequence, each step is normalized over a large number of terms so their effect is dampened. By keeping only $k$ time steps and not the entire sequence, the statistics will vary gradually across time, and we are able to fix the memory and computational costs, which could be significant for long sequences.

Using the information from multiple time steps also effectively provides a larger set on which to calculate statistics. This allows a more accurate approximation to gain a clearer glimpse at the underlying distribution of the dataset. In other words, ATN uses statistics over a larger set that is more stable across time so that the normalized state can retain more variations in time. In contrast, the traditional normalization methods use high-frequency statistics at each time step to produce a normalized state that becomes time-invariant. In particular, the ATN network depends on the scaling of the input vector at a time step, while LN and BN do not. However, ATN preserves the desirable weight scaling invariant property, which we show as follows:

Let $H$ and $\tilde{H}$ be weight matrices for two sets of model parameters, $\theta$ and $\tilde{\theta}$ respectively, which differ by a scaling factor of $\delta$, i.e. $\tilde{H} = \delta H$. Then the outputs of ATN are the same:

$$\tilde{y}^{(t)} = \frac{\gamma}{\tilde{\sigma}_{t,k}} \odot \left(\tilde{H}a^{(t)} - \tilde{\mu}_{t,k}\right) + \beta \ = \frac{\gamma}{\sigma_{t,k}} \odot \left(Ha^{(t)} - \mu_{t,k}\right) + \beta = y^{(t)} \tag{10}$$

where $\tilde{\sigma}_{t,k} = \delta\sigma_{t,k}$ and $\tilde{\mu}_{t,k} = \delta\mu_{t,k}$. Since ATN is applied before the activation function, equivalence at this stage guarantees equivalence post-activation. This invariance property makes the ATN network independent of the norm of $H$, mitigating the exploding/vanishing gradient problems.

It is also easy to see that ATN is also invariant to the rescaling of the whole input sequence, since the rescaling factor can be pulled out of the summation across the sequence. Likewise, ATN is not invariant under the rescaling of an individual element in the sequence since such factoring is not possible.

During training, we backpropagate the gradients with respect to the model parameters. With ATN, a key step is to propagate the gradient through the normalization layer, i.e.,

$\dfrac{\partial y_i^{(t)}}{\partial a_i^{(t-m)}}$. The following proposition gives the formulas for computing these derivatives. The proof is provided in "Appendix A".

**Proposition 1** *Consider ATN for a sequence* $\mathbf{a} = \{a^{(t)}\} \subset \mathbb{R}^n$ *produced in a RNN and let* $y^{(t)} = ATN(\mathbf{a}_k^{(t)}; \gamma, \beta)$*. Then, for* $0 \le m \le k-1$*, we have:*

$$\frac{\partial y_i^{(t)}}{\partial a_i^{(t-m)}} = \gamma \odot \frac{\dfrac{\partial a_i^{(t)}}{\partial y_i^{(t-m)}} \dfrac{\partial y_i^{(t-m)}}{\partial a_i^{(t-m)}} - \dfrac{\partial \mu_{t,k}}{\partial a_i^{(t-m)}}}{\sqrt{\sigma_{t,k}^2 + \varepsilon}} - \gamma \odot \frac{a_i^{(t)} - \mu_{t,k}}{2\left(\sigma_{t,k}^2 + \varepsilon\right)^{3/2}} \frac{\partial \sigma_{t,k}^2}{\partial a_i^{(t-m)}} \tag{11}$$

*where*

$$\frac{\partial \mu_{t,k}}{\partial a_i^{(t-m)}} = \frac{1}{nk} \textstyle\sum_{j=0}^{m} \dfrac{\partial a_i^{(t-j)}}{\partial a_i^{(t-m)}} \tag{12}$$

$$\frac{\partial \sigma_{t,k}^2}{\partial a_i^{(t-m)}} = \frac{2}{nk} \sum_{j=0}^{m} \left(a_i^{(t-j)} - \mu_{t,k}\right) \frac{\partial a_i^{(t-j)}}{\partial a_i^{(t-m)}} - \sum_{j=0}^{k-1} \sum_{s=1}^{n} \left(a_s^{(t-j)} - \mu_{t,k}\right) \frac{\partial \mu_{t,k}}{\partial a_i^{(t-m)}}. \tag{13}$$

Note that the computations of $\dfrac{\partial y_i^{(t)}}{\partial \beta}$ and $\dfrac{\partial y_i^{(t)}}{\partial \gamma}$ are straightforward and are omitted.

In our experiments, we will use ATN (combined with LN) on LSTM networks. Following [1, 3], our ATN method for LSTM is as follows:

$$\begin{pmatrix} f^{(t)} \\ i^{(t)} \\ o^{(t)} \\ g^{(t)} \end{pmatrix} = ATN(W_h h^{(t-1)}) + ATN(W_x x^{(t)}) + b \tag{14}$$

$$c^{(t)} = \sigma(f^{(t)}) \odot c^{(t-1)} + \sigma(i^{(t)}) \odot \tanh(g^{(t)}) \tag{15}$$

$$h^{(t)} = \sigma(o^{(t)}) \odot \tanh(ATN(c^{(t)})) \tag{16}$$

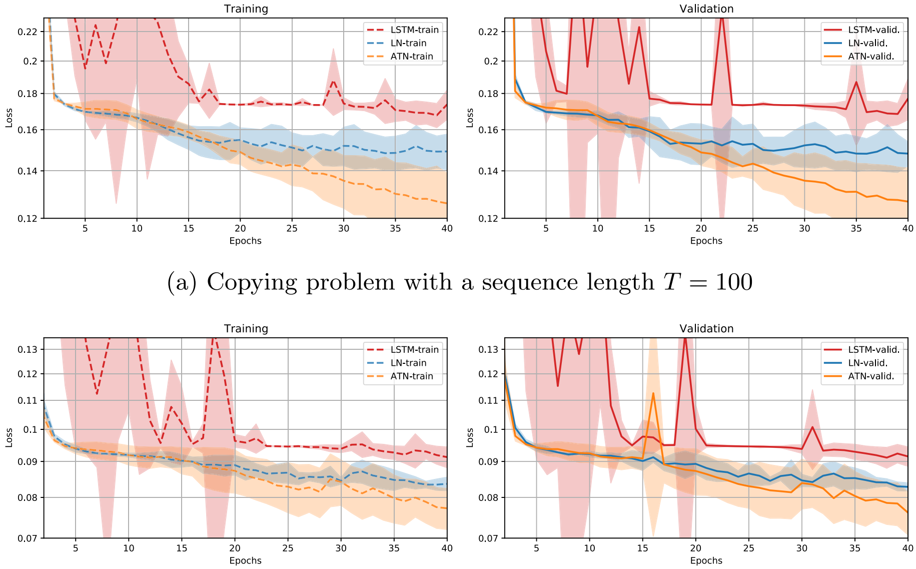
where $\odot$ is the Hadamard product and $\sigma(\cdot)$ is the sigmoid function.

This model has comparable computational costs as the LN-LSTM model since each LN-ATN block has the cost of an LN block scaled linearly in $k$. This is a small portion of the overall cost of the model and can be regulated by the choice of $k$.

## 4 Experiments

We have performed a series of experiments which include the Copying [7], Adding [7], and Denoise problems [4, 11] as well as Language Modeling on character level Penn Treebank dataset [15] and word level WikiText-2 dataset [16]. All experiments compare LSTM models with no normalization (LSTM), Layer Normalization (LN), and Assorted Temporal Normalization combined with Layer Normalization (ATN).

All experiments were run using Python 3.7.0, PyTorch 1.1.0, and CUDA 9.0 on a single NVIDIA Tesla V100 GPU.



(a) Copying problem with a sequence length $T = 100$

(b) Copying problem with a sequence length $T = 200$

**Fig. 2** Results on the copying problem for sequence lengths $T = 100$ and $T = 200$. The LSTM and LN-LSTM (LN) models are provided here for comparison purposes with our ATN-LSTM (ATN) method

## 4.1 Synthetic Tasks

### 4.1.1 Copying

The copying problem is a common synthetic task that is used to test RNNs, which was originally proposed in Hochreiter and Schmidhuber [7]. For this problem, a string of 10 digits is fed into the RNN sampled uniformly from the integers between 1 and 8. A sequence of $T$ zeros follows this, and a 9, marking the start of a string of 9 zeros, for a total length of $T + 20$. The objective of the task is to output the initial string of 10 digits beginning at the marker's location, copying the initial string from the front to the back. Cross-entropy loss is used to evaluate this model, with a baseline expected cross-entropy of $\frac{10 \log(8)}{T+20}$ which represents selecting digits 1–8 at random after the 9.

*Implementation details* The models were trained with a batch size of 128, a single LSTM layer with a hidden size of 68, an RMSProp [22] optimizer with a learning rate of $10^{-4}$, and $T$ values of 100 and 200. The ATN model is implemented with $k = 45$ for both $T$ values. All models were trained for 40 epochs using cross-entropy loss.

*Results* For each of the sequence lengths tested, the plain LSTM is incapable of achieving losses below the baseline. While the LN-LSTM is able to do so to some extent on the $T = 100$ version, see Fig. 2a, it also gets stuck at the baseline loss on the $T = 200$ task, Fig. 2b. For both of these tasks, our ATN-LSTM model demonstrates eventual losses below those reached by the LN-LSTM model, Fig. 2a, b. We also note that the initial rate of convergence is at least as steep if not steeper than that of the LN-LSTM model, demonstrating that the ATN-LSTM positively contributes to training in both the short and long term. For more quantitative results, see Table 1.

**Table 1** Copying results: attained minimum values

| Sequence length | Loss × $10^{-1}$ ↓ <br> $T = 100$ | | Loss × $10^{-2}$ ↓ <br> $T = 200$ | |
|---|---|---|---|---|
| | Train | Validation | Train | Validation |
| LSTM | 1.677 ± 0.006 | 1.684 ± 0.006 | 9.129 ± 0.311 | 9.114 ± 0.311 |
| LN | 1.483 ± 0.080 | 1.480 ± 0.074 | 8.343 ± 0.153 | 8.282 ± 0.110 |
| ATN | 1.260 ± 0.091 | 1.267 ± 0.134 | 7.719 ± 0.517 | 7.608 ± 0.533 |

↓—denotes the smaller, the better the result

**Table 2** Adding results: attained minimum values

| Sequence length | Loss ×$10^{-3}$ ↓ <br> $T = 100$ | | Loss ×$10^{-3}$ ↓ <br> $T = 200$ | |
|---|---|---|---|---|
| | Train | Validation | Train | Validation |
| LSTM | 1.036 ± 0.232 | 0.988 ± 0.154 | 1.907 ± 0.322 | 2.216 ± 1.225 |
| LN | 0.996 ± 0.348 | 0.696 ± 0.256 | 1.511 ± 0.720 | 1.813 ± 0.967 |
| ATN | 0.481 ± 0.086 | 0.459 ± 0.074 | 1.208 ± 0.133 | 1.172 ± 0.226 |

↓—denotes the smaller, the better result

### 4.1.2 Adding

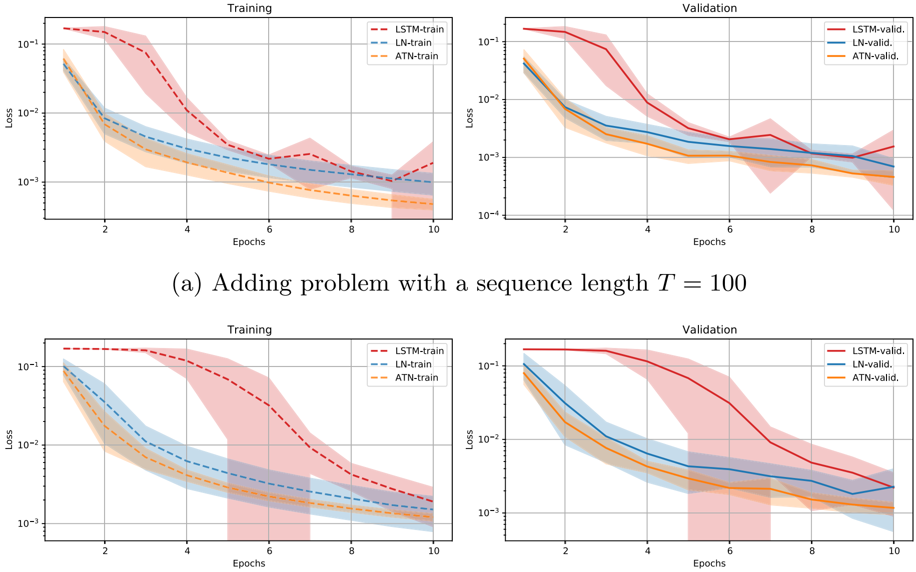
The adding problem is another synthetic task for RNNs proposed in Hochreiter and Schmidhuber [7]. Our implementation of this problem is a variation of the original problem. The RNN takes a 2-dimensional input of length T. The first dimension consists of a sequence of zeros except for two ones placed randomly in the first and second half of the sequence. The second dimension is a sequence of numbers selected uniformly from [0, 1). The goal of the task is to take the numbers from the second dimension in positions corresponding to the ones and to output their sum.

*Implementation details* The models were trained with a batch size of 50, a single LSTM layer with a hidden size of 60, and an RMSprop [22] optimizer with a learning rate of $10^{-3}$. We use $T$ values of 100 and 200. This task is trained and evaluated with a mean-squared error (MSE) loss function. The ATN model is implemented with $k$ values of 25 for $T = 100$ and 5 for $T = 200$. The models were trained for 10 epochs.

*Results* Our model shows consistent improvement over the LSTM and LN-LSTM models. For each example, the ATN shows a rapid initial convergence before settling into a slower rate which is roughly parallel to that of the LN-LSTM. In Fig. 3a, this initial conversion almost manages to take the model to the same loss as is achieved by the LN-LSTM after the entirety of the training. In Fig. 3b, the LN-LSTM is able to separate itself further from the LSTM than in Fig. 3a but is still at a higher loss than the ATN for all but the very beginning of training. For more quantitative results, see Table 2.

### 4.1.3 Denoise Task

The Denoise Task [4, 11] is another synthetic problem that requires filtering out the noise out of a noisy sequence. This problem requires the forgetting ability of the network as well as learning long-term dependencies coming from the data [11]. The input sequence of length $T$



(a) Adding problem with a sequence length $T = 100$

(b) Adding problem with a sequence length $T = 200$

**Fig. 3** Results on the adding problem for sequence lengths $T = 100$ and $T = 200$

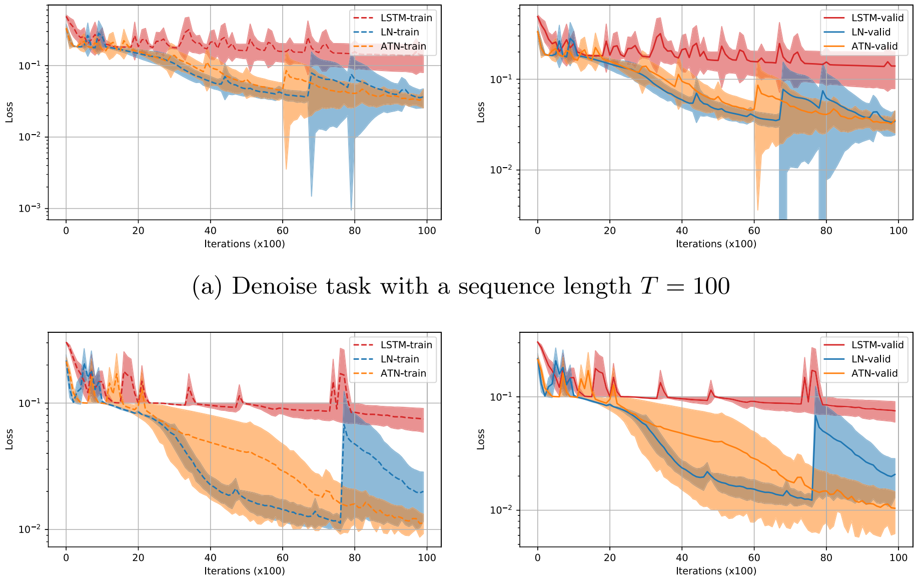
contains 10 randomly located data points, and the other $T - 10$ points are considered noise data. These 10 points are selected from a dictionary $\{a_i\}_{i=0}^{n+1}$, where the first $n$ elements are data points, and the other two are the "noise" and the "marker" respectively. The output data consists of the list of the data points from the input, and it should be outputted as soon as it receives the "marker". The goal is to filter out the noise and output the random 10 data points chosen from the input.

*Implementation details* The models were trained using a batch size of 128, a single LSTM layer with a hidden size of 100, and Adam [12] optimizer with a learning rate of $10^{-2}$, and a cross-entropy loss function. We use $T$ values of 100 and 200. The ATN model is implemented with $k$ values of 20 and 60 for $T = 100$ and $T = 200$, respectively. The models were trained for 10,000 iterations.

*Results* For both sequence lengths, our models outperform the LSTM and matched the LN-LSTM throughout training curves as can be seen in Fig. 4a, b. While we had wider fluctuations during training, our model converged to a lower loss with a tighter band than both other models. For more quantitative results, see Table 3.

### 4.2 Language Models

Language modeling is one of many natural language processing tasks. It is the development of probabilistic models that are capable of predicting the next word or character in a sequence using information that has preceded it. For both of the Language Modeling problems, we based our experiments on the AWD-LSTM model [17].



(a) Denoise task with a sequence length $T = 100$

(b) Denoise task with a sequence length $T = 200$

**Fig. 4** Results on the Denoise task for sequence lengths $T = 100$ and $T = 200$

**Table 3** Denoise results: attained minimum cross entropy loss values

| Sequence length | Loss $\times 10^{-2}$ ↓<br>$T = 100$ | | Loss $\times 10^{-3}$ ↓<br>$T = 200$ | |
|---|---|---|---|---|
| | Train | Validation | Train | Validation |
| LSTM | $13.875 \pm 5.855$ | $13.835 \pm 5.966$ | $74.739 \pm 1.594$ | $75.243 \pm 1.543$ |
| LN | $3.555 \pm 1.013$ | $3.339 \pm 0.097$ | $11.235 \pm 1.580$ | $12.258 \pm 1.629$ |
| ATN | $3.272 \pm 0.595$ | $3.226 \pm 0.056$ | $11.151 \pm 2.448$ | $10.399 \pm 4.194$ |

↓—denotes the smaller, the better result

### 4.2.1 Metrics

There are two widely used metrics in natural language processing and language modeling: Bits Per Character (`bpc`) and Perplexity (`PPL`).

Bits Per Character (`bpc`) measures the average number of bits required to represent a single character in a given text or data. Lower `bpc` values indicate more efficient compression or encoding.

Perplexity (`PPL`) is a metric particularly used in the evaluation of probabilistic language models, introduced in [10]. `PPL` measures how well a language model predicts a sample of text. Lower `PPL` values indicate that the language model is better at predicting the text and has a better understanding of the language.

**Table 4** Character level Penn treebank results: attained minimum values

| | bpc ↓ | |
|---|---|---|
| | Validation | Test |
| LSTM | 1.824 ± 0.101 | 1.705 ± 0.019 |
| LN | 1.535 ± 0.003 | 1.551 ± 0.016 |
| ATN | 1.504 ± 0.002 | 1.547 ± 0.018 |

↓—denotes the smaller, the better result

### 4.2.2 Character Level Penn Treebank

The models were tested on their suitability for language modeling tasks using the character level Penn Treebank dataset [15] also known as character-PTB or simply cPTB dataset. This dataset is a collection of English-language Wall Street Journal articles. The dataset consists of a vocabulary of 10,000 words with other words replaced as `<unk>`, resulting in approximately 6 million characters that are divided into 5.1 million, 400 thousand, and 450 thousand character sets for training, validation, and testing, respectively with a character alphabet size of 50. The goal of the character-level Language Modeling task is to predict the next character given the preceding sequence of characters.

*Implementation details* For this task, we partitioned the training sequence into 220 character length subsequences. The models were trained using a batch size of 32, a single LSTM layer with a hidden size of 1000, an Adam [12] optimizer with a learning rate of $10^{-2}$, gradient clipping by norm at 3, and learning rate decay by a factor of 10 at epoch 80 and 90. The ATN model is implemented with a $k$ value of 10. The models were trained for 150 epochs.

*Results* Our model shows improvement over the LSTM and the LN-LSTM models, the comparison results are presented in Table 4.

### 4.2.3 WikiText-2

The WikiText-2 dataset was introduced in Merity et al. [16]. It is approximately two times the size of the Penn Treebank dataset and contains preprocessed Wikipedia articles while maintaining the original structure, punctuation, and symbols. The WikiText-2 dataset consists of approximately 2.2 million words: 2 million for the training set and 200 thousand for the validation and test sets, with a vocabulary size of 33,278. This task is a word-level Language Modeling problem with the goal to predict the next word given the preceding sequence of words.

*Implementation details* We used a batch size of 32; three LSTM layers with embedding and hidden sizes of 400 and 1150, respectively; BPTT values of 70; gradient clipping on the norm of 0.25; and learning rate of 30 with Stochastic Gradient Descent (SGD) optimizer without any momentum or learning rate decay, and switch to ASGD [19] optimizer using nonmono criteria from [17] with value 5 (our experiments showed that switching happens approximately between epochs 20 and 30 for all models: LSTM, LN, and ATN). The ATN model is implemented with a $k$ value of 25. All models were trained for 500 epochs.

*Results* In this experiment, the ATN method shows improvement over LSTM and LN method in both training and validation PPL, see Table 5.

**Table 5** WikiText-2 results: attained minimum values

| | PPL ↓ | |
|---|---|---|
| | Validation | Test |
| LSTM | $80.79 \pm 0.861$ | $65.76 \pm 1.05$ |
| LN | $80.08 \pm 1.116$ | $58.34 \pm 0.54$ |
| ATN | $78.32 \pm 0.695$ | $56.29 \pm 0.44$ |

↓—denotes the smaller, the better the result

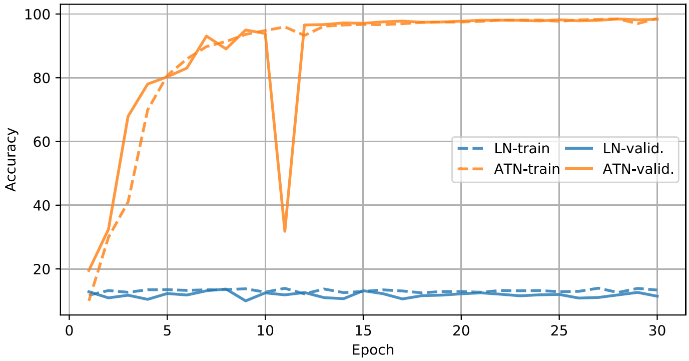


**Fig. 5** Pixel-by-pixel MNIST: ablation study showing the performance impact of time-invariant normalization

# 5 Ablation Studies

## 5.1 Input Statistic Invariance Across Time

In most implementations of LN-LSTM, including the one used in the experiments above, the inputs to the normalization method are the results of a linear layer, including both weight and bias. This differs slightly from the model proposed in Ba et al. [1] in that their version placed the LSTM bias outside of the normalization. Using that original architecture, we can clearly demonstrate the underlying problem with Layer Normalization that we aim to solve, the loss of input information, by setting the statistics to constant values across time.

To show this, we use the MNIST dataset [14] after applying Gaussian noise with variance 0.1 for the pixel-by-pixel task [13]. This task takes the pixel values of a handwritten digit and inputs them as an unpermuted sequence of length 784 in order to predict the digit class. Due to the high probability of pixels having near zero values, we needed to use $\varepsilon$ values of 1 in both normalization schemes. With this task, we can see in Fig. 5 that the use of Layer Normalization renders the model completely incapable of training. Because LN takes the information from each pixel and normalizes it to the exact same distribution, it erases everything the model could use to learn, making it no better than guessing. The ATN method with $k = 10$ solves this problem by using multiple time steps in calculating the mean and variance, meaning that the normalized outputs will not all have identical statistics. This change allows ATN to perform quite well, even when Layer Normalization cannot.

### 5.2 Post Normalization Statistics

In Fig. 6a–c, we present the statistics of the post-normalization components from a single iteration of training for the Adding Problem [7] described in Sect. 4.1.2 with $T = 75$. We present the statistics from four different models, an LN-LSTM, and three ATN($k$) models with $k$ values of 5, 25, and 55. All of the models did not include the use of trainable bias and gain parameters inside the normalization methods.

In Fig. 6a, we show the mean and variance after normalization of the product of the hidden-to-hidden weight and the hidden state, $W_h h^{(t-1)}$. While Layer Normalization produces constant mean and variance, the ATN method allows for the statistics to vary at each time step, resulting in curves that do not differ too much from those for LN in terms of scale but do demonstrate the natural fluctuations in the hidden states. From this, we can see that we are achieving the combination of a controlled output that is still capable of reflecting the temporal changes of the network.

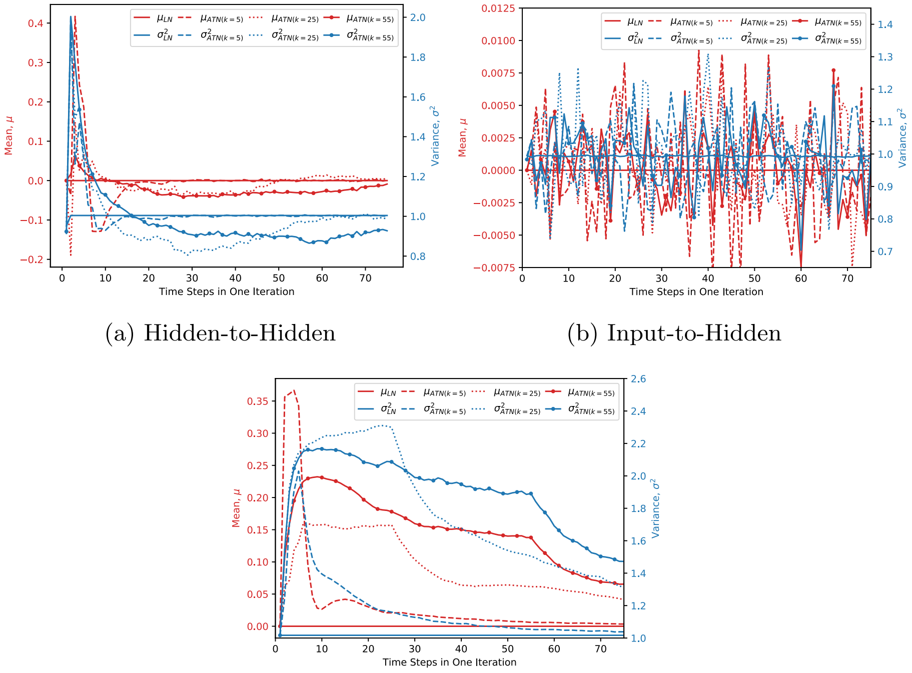
In Fig. 6b, we show the statistics from the product of the input-to-hidden weight and the input, $W_x x^{(t)}$. The ATN model provides highly variable means and variances, showcasing the amount of information about the dataset which is lost when LN resets the statistics to these constant values.

In Fig. 6c, we show the post-normalization statistics of the memory cell, $c^{(t)}$. These statistics clearly demonstrate the effect of a shorter $k$ value as opposed to a longer one in the mean. In the early iterations for the $k = 5$ model, the mean has a larger spike which flattens to a bit above zero by the end of the iteration. For the larger $k$ values, this initially increased mean gets maintained throughout a larger portion of the iteration, causing the lower values further along to have less influence on the statistics.

### 5.3 Optimal *k* Value for ATN Method

To highlight the importance of normalizing with respect to $k$ time steps instead of just one or all of them, we present a study on various $k$ values. In Fig. 7, we present results on the Copying Problem [7] described in Sect. 4.1.1 with $T = 100$. For this experiment, we have trained LSTM, LN, and three ATN($k$) models with values of $k$ being 25, 45, and 65 under the same conditions.

All ATN models perform better than both LSTM and LN. The ATN ($k = 45$) model performs better than ATN ($k = 25$), which should not be a surprise since the larger $k$ value would mean we are normalizing with respect to a larger set and getting better statistics for the mean and variance; however, ATN ($k = 65$) performs poorer than ATN ($k = 45$) and even poorer than ATN ($k = 25$) which suggests that too large $k$ may actually degrade the result. This may be due to numerical difficulties in propagating derivatives through $k$ steps in ATN for a large $k$, and due to the overwhelming of the short-term normalization statistic fluctuations by the long sequence. Since each timestep included in ATN increases the operations required for the method, adding to the computational cost in both the forward pass and the gradient computation, limiting $k$ comes with practical benefits. Furthermore, as seen in Proposition 1, the larger the $k$ value, the more often a particular timestep is used in gradient computation, allowing for greater propagation of error. This is a common concern with RNN tasks and is comparable to the reasoning behind the use of truncated backpropagation through time. While there is some difference between the model performance, based on $k$ value, the difference within short ranges is not too pronounced so while ATN introduces a tunable hyperparameter in $k$, it is not an overly sensitive one.



(a) Hidden-to-Hidden

(b) Input-to-Hidden

(c) Memory Cell

**Fig. 6** Post normalization statistics for adding problem with $T = 75$

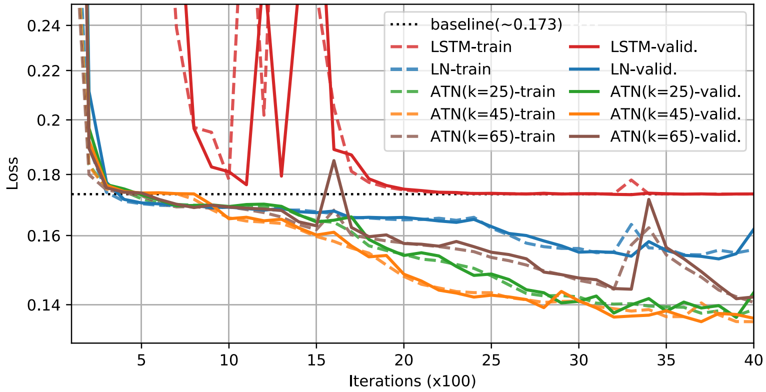


**Fig. 7** Ablation study for optimal $k$ value for ATN method using Copying Task (see Sect. 4.1.1) with a sequence length of $T = 100$

## 6 Conclusion

In this paper, we have introduced a method for adapting statistics-based normalization methods to recurrent neural networks to break the time invariance of the traditional normalization methods. We have presented theoretical results on the impact this method has on the model's gradients, as well as showing the preservation of invariance to the rescaling of the weight matrix. Our experiments demonstrate that our ATN-LSTM improves over LN for LSTM in both training and testing results. In light of the popularity of LN in practical applications, our method offers an important alternative for further improving RNN performance.

**Acknowledgements** We thank the University of Kentucky Center for Computational Sciences and Information Technology Services Research Computing for their support and use of the Lipscomb Compute Cluster and associated research computing resources.

**Author Contributions** Not applicable.

**Funding** Research supported in part by US National Science Foundation under the Grants DMS-2208314 and IIS-2327113.

**Availability of data and materials** Data sharing not applicable to this article as no datasets were generated or analysed during the current study.

## Declarations

**Conflict of interest** No potential conflict of interest was reported by the authors.

**Ethics approval** Not applicable.

**Consent to participate** Not applicable.

**Consent for publication** Not applicable.

## Appendix A: Proof of Proposition 1

We present below the derivation for the propagation of the gradient through the ATN method.

**Proposition 1** *Consider ATN for a sequence* $\mathbf{a} = \{a^{(t)}\} \subset \mathbb{R}^n$ *produced in a RNN and let* $y^{(t)} = ATN(\mathbf{a}_k^{(t)}; \gamma, \beta)$. *Then, for* $0 \leq m \leq k-1$, *we have:*

$$\frac{\partial y_i^{(t)}}{\partial a_i^{(t-m)}} = \gamma \odot \frac{\frac{\partial a_i^{(t)}}{\partial y_i^{(t-m)}} \frac{\partial y_i^{(t-m)}}{\partial a_i^{(t-m)}} - \frac{\partial \mu_{t,k}}{\partial a_i^{(t-m)}}}{\sqrt{\sigma_{t,k}^2 + \varepsilon}} - \gamma \odot \frac{a_i^{(t)} - \mu_{t,k}}{2\left(\sigma_{t,k}^2 + \varepsilon\right)^{3/2}} \frac{\partial \sigma_{t,k}^2}{\partial a_i^{(t-m)}} \tag{A1}$$

*where*

$$\frac{\partial \mu_{t,k}}{\partial a_i^{(t-m)}} = \frac{1}{nk} \sum_{j=0}^{m} \frac{\partial a_i^{(t-j)}}{\partial a_i^{(t-m)}} \tag{A2}$$

*and*

$$\frac{\partial \sigma_{t,k}^2}{\partial a_i^{(t-m)}} = \frac{2}{nk}\left(\sum_{j=0}^{m}\left(a_i^{(t-j)} - \mu_{t,k}\right)\frac{\partial a_i^{(t-j)}}{\partial a_i^{(t-m)}} - \sum_{j=0}^{k-1}\sum_{s=1}^{n}\left(a_s^{(t-j)} - \mu_{t,k}\right)\frac{\partial \mu_{t,k}}{\partial a_i^{(t-m)}}\right) \tag{A3}$$

***Proof*** Suppose $0 \leq m \leq k-1$ then

$$\mu_{t,k} = \frac{1}{nk}\sum_{j=0}^{k-1}\sum_{s=1}^{n} a_s^{(t-j)} \tag{A4}$$

and

$$\frac{\partial \mu_{t,k}}{\partial a_i^{(t-m)}} = \frac{1}{nk}\sum_{j=0}^{k-1}\sum_{s=1}^{n}\frac{\partial a_s^{(t-j)}}{\partial a_i^{(t-m)}} \tag{A5}$$

$$= \frac{1}{nk}\left(\frac{\partial a_i^{(t)}}{\partial a_i^{(t-m)}} + \frac{\partial a_i^{(t-1)}}{\partial a_i^{(t-m)}} + \cdots + \frac{\partial a_i^{(t-m+1)}}{\partial a_i^{(t-m)}} + 1\right) \tag{A6}$$

$$= \frac{1}{nk}\sum_{j=0}^{m}\frac{\partial a_i^{(t-j)}}{\partial a_i^{(t-m)}}; \tag{A7}$$

$$\sigma_{t,k}^2 = \frac{1}{nk}\sum_{j=0}^{k-1}\sum_{s=1}^{n}\left(a_s^{(t-j)} - \mu_{t,k}\right)^2 \tag{A8}$$

and

$$\frac{\partial \sigma_{t,k}^2}{\partial a_i^{(t-m)}} = \frac{1}{nk}\sum_{j=0}^{k-1}\sum_{s=1}^{n} 2\left(a_s^{(t-j)} - \mu_{t,k}\right)\left(\frac{\partial a_s^{(t-j)}}{\partial a_i^{(t-m)}} - \frac{\partial \mu_{t,k}}{\partial a_i^{(t-m)}}\right) \tag{A9}$$

$$= \frac{2}{nk}\left(\sum_{j=0}^{m}\left(a_i^{(t-j)} - \mu_{t,k}\right)\frac{\partial a_i^{(t-j)}}{\partial a_i^{(t-m)}} - \sum_{j=0}^{k-1}\sum_{s=1}^{n}\left(a_s^{(t-j)} - \mu_{t,k}\right)\frac{\partial \mu_{t,k}}{\partial a_i^{(t-m)}}\right); \tag{A10}$$

$$y^{(t)} = \gamma \odot \frac{a^{(t)} - \mu_{t,k}}{\sqrt{\sigma_{t,k}^2 + \varepsilon}} + \beta \tag{A11}$$

and

$$\frac{\partial y_i^{(t)}}{\partial a_i^{(t-m)}} = \gamma \odot \frac{\frac{\partial a_i^{(t)}}{\partial a_i^{(t-m)}} - \frac{\partial \mu_{t,k}}{\partial a_i^{(t-m)}}}{\sqrt{\sigma_{t,k}^2 + \varepsilon}} - \gamma \odot \frac{1}{2}\frac{a_i^{(t)} - \mu_{t,k}}{\left(\sigma_{t,k}^2 + \varepsilon\right)^{3/2}}\frac{\partial \sigma_{t,k}^2}{\partial a_i^{(t-m)}} \tag{A12}$$

where

$$\frac{\partial a_i^{(t)}}{\partial a_i^{(t-m)}} = \frac{\partial a_i^{(t)}}{\partial y_i^{(t-m)}}\frac{\partial y_i^{(t-m)}}{\partial a_i^{(t-m)}}. \tag{A13}$$

□

**Table 6** Invariance properties under different normalization methods

| | BN | WN | LN | ATN-BN | ATN-LN |
|---|---|---|---|---|---|
| Weight matrix re-scaling | Yes | Yes | Yes | Yes | Yes |
| Weight matrix re-centering | No | No | Yes | No | No |
| Weight vector re-scaling | Yes | Yes | No | Yes | No |
| Dataset re-scaling | Yes | No | Yes | Yes | Yes |
| Dataset re-centering | Yes | No | No | Yes | No |
| Single training case re-scaling | No | No | Yes | No | Yes |
| Input at a single time re-scaling | No | No | Yes | No | No |

*BN* batch normalization [9], *WN* weight normalization [21], *LN* layer normalization [1], *ATN-BN* assorted-time normalization built on BN method, *ATN-LN* assorted-time normalization built on LN method

## Appendix B: Invariance Properties

In Table 6 we provide a summary of invariance properties for several normalization methods. This is an expansion of Table 1 in Ba et al. [1]. Weight matrix re-scaling and re-centering are the adjustments of the weight matrix by multiplying a constant scaling factor or adding a constant re-scaling factor. Weight vector re-scaling is similar to weight matrix re-scaling, but only adjusts a single vector instead of the entire matrix. Dataset re-centering and re-scaling consist of changing every input example by multiplying or adding a constant. Single training case re-scaling is when the dataset adjustments are applied to just one example. Of particular interest is the invariance with respect to the scaling of an input at a single time point, which was referenced in Sect. 3. This is one of the invariance properties which LN has that its ATN adaptation does not, and we argue that this is one of the reasons that our method improves on LN.

## References

1. Ba JL, Kiros JR, Hinton GE (2016) Layer normalization. arXiv:1607.06450
2. Cho K, van Merrienboer B, Gulcehre C et al (2014) Learning phrase representations using RNN encoder–decoder for statistical machine translation. https://doi.org/10.48550/ARXIV.1406.1078
3. Cooijmans T, Ballas N, Laurent C et al (2017) Recurrent batch normalization. In: 5th international conference on learning representations, ICLR 2017, Toulon, France, April 24–26, 2017, conference track proceedings. OpenReview.net. https://openreview.net/forum?id=r1VdcHcxx
4. Foerster JN, Gilmer J, Chorowski J et al (2016) Input switched affine networks: an RNN architecture designed for interpretability. https://doi.org/10.48550/ARXIV.1611.09434
5. Gitman I, Ginsburg B (2017) Comparison of batch normalization and weight normalization algorithms for the large-scale image classification. arXiv:1709.08145
6. Hochreiter S, Schmidhuber J (1997) Long short-term memory. Neural Comput 9(8):1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735
7. Hochreiter S, Schmidhuber J (1997) Long short-term memory. Neural Comput 9(8):1735–1780
8. Hopfield JJ (1982) Neural networks and physical systems with emergent collective computational abilities. Proc Natl Acad Sci 79(8):2554–2558. https://doi.org/10.1073/pnas.79.8.2554
9. Ioffe S, Szegedy C (2015) Batch normalization: accelerating deep network training by reducing internal covariate shift. In: Bach F, Blei D (eds) Proceedings of the 32nd international conference on machine learning, proceedings of machine learning research, vol 37. PMLR, Lille, France, pp 448–456. https://proceedings.mlr.press/v37/ioffe15.html
10. Jelinek F, Mercer RL, Bahl LR et al (1977) Perplexity-a measure of the difficulty of speech recognition tasks. J Acoust Soc Am 62. https://api.semanticscholar.org/CorpusID:121680873

11. Jing L, Gulcehre C, Peurifoy J et al (2019) Gated orthogonal recurrent units: on learning to forget. Neural Comput 31(4):765–783. https://doi.org/10.1162/necoa01174
12. Kingma D, Ba J (2014) Adam: a method for stochastic optimization. arXiv:1412.6980
13. Le QV, Jaitly N, Hinton GE (2015) A simple way to initialize recurrent networks of rectified linear units. arXiv:1504.00941
14. Lecun Y, Bottou L, Bengio Y et al (1998) Gradient-based learning applied to document recognition. Proc IEEE 86(11):2278–2324. https://doi.org/10.1109/5.726791
15. Marcus MP, Marcinkiewicz MA, Santorini B (1993) Building a large annotated corpus of English: the Penn treebank. Comput Linguist 19(2):313–330
16. Merity S, Xiong C, Bradbury J et al (2016) Pointer sentinel mixture models. arXiv:1609.07843
17. Merity S, Keskar NS, Socher R (2018) Regularizing and optimizing LSTM language models. In: International conference on learning representations. https://openreview.net/forum?id=SyyGPP0TZ
18. Mucllari E, Zadorozhnyy V, Pospisil C et al (2022) Orthogonal gated recurrent unit with Neumann–Cayley transformation. https://doi.org/10.48550/ARXIV.2208.06496
19. Polyak BT, Juditsky AB (1992) Acceleration of stochastic approximation by averaging. SIAM J Control Optim 30(4):838–855. https://doi.org/10.1137/0330046
20. Rumelhart DE, Hinton GE, Williams RJ (1986) Learning internal representations by error propagation. MIT Press, Cambridge, pp 318–362
21. Salimans T, Kingma DP (2016) Weight normalization: a simple reparameterization to accelerate training of deep neural networks. In: Lee D, Sugiyama M, Luxburg U et al (eds) Advances in neural information processing systems, vol 29. Curran Associates Inc, New York
22. Tieleman T, Hinton G (2012) Lecture 6.5-rmsprop: divide the gradient by a running average of its recent magnitude. COURSERA Neural Netw Mach Learn 4(2):26
23. Ulyanov D, Vedaldi A, Lempitsky V (2017) Instance normalization: the missing ingredient for fast stylization. arXiv:1607.08022
24. Wu Y, He K (2018) Group normalization. In: Proceedings of the European conference on computer vision (ECCV)
25. Xu J, Sun X, Zhang Z et al (2019) Understanding and improving layer normalization. In: Wallach H, Larochelle H, Beygelzimer A et al (eds) Advances in neural information processing systems, vol 32. Curran Associates, Inc., pp 4381–4391

**Publisher's Note** Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.