Title
MomentumRNN: Integrating Momentum into Recurrent Neural Networks

Authors
Nguyen, Tan M
Baraniuk, Richard G
Bertozzi, Andrea L
et al.

Publication Date
2020-06-11

Peer reviewed

MomentumRNN: Integrating Momentum into Recurrent Neural Networks

Tan M. Nguyen
Department of ECE
Rice University, Houston, USA

Richard G. Baraniuk
Department of ECE
Rice University, Houston, USA

Andrea L. Bertozzi
Department of Mathematics
University of California, Los Angeles

Stanley J. Osher
Department of Mathematics
University of California, Los Angeles

Bao Wang*
Department of Mathematics
Scientific Computing and Imaging (SCI) Institute
University of Utah, Salt Lake City, UT, USA

June 15, 2020

Abstract

Designing deep neural networks is an art that often involves an expensive search over candidate architectures. To overcome this for recurrent neural nets (RNNs), we establish a connection between the hidden state dynamics in an RNN and gradient descent (GD). We then integrate momentum into this framework and propose a new family of RNNs, called *MomentumRNNs*. We theoretically prove and numerically demonstrate that MomentumRNNs alleviate the vanishing gradient issue in training RNNs. We study the momentum long-short term memory (MomentumLSTM) and verify its advantages in convergence speed and accuracy over its LSTM counterpart across a variety of benchmarks, with little compromise in computational or memory efficiency. We also demonstrate that MomentumRNN is applicable to many types of recurrent cells, including those in the state-of-the-art orthogonal RNNs. Finally, we show that other advanced momentum-based optimization methods, such as Adam and Nesterov accelerated gradients with a restart, can be easily incorporated into the MomentumRNN framework for designing new recurrent cells with even better performance. The code is available at `https://github.com/minhtannguyen/MomentumRNN`.

1 Introduction

Mathematically principled recurrent neural nets (RNNs) facilitate the network design process and reduce the cost of searching over many candidate architectures. A particular advancement in RNNs is the long short-term memory (LSTM) model [23] which has achieved state-of-the-art results in many applications, including speech recognition [14], acoustic modeling [47, 49], and language modeling [42]. There have been many efforts in improving LSTM: [18] introduces a forget gate into the original LSTM cell, which can forget information selectively; [17] further adds peephole connections to the LSTM cell to inspect its current internal states[16]; to reduce the computational cost, a gated recurrent unit (GRU) [10] uses a single update gate to replace the forget and input gates in LSTM. Phased LSTM [38] adds a new time gate to the LSTM cell and achieves faster convergence than the regular LSTM on learning long sequences. In addition, [48] and [46] introduce a biological cell state and working memory into LSTM, respectively. Nevertheless, most of RNNs, including LSTMs, are biologically informed or even ad-hoc instead of being guided by mathematical principles.

*Please correspond to: wangbaonj@gmail.com or mn15@rice.edu

1.1 Recap on RNNs and LSTM

Recurrent cells are the building blocks of RNNs. A recurrent cell employs a cyclic connection to update the current hidden state ($\boldsymbol{h}_t$) using the past hidden state ($\boldsymbol{h}_{t-1}$) and the current input data ($\boldsymbol{x}_t$) [13]; the dependence of $\boldsymbol{h}_t$ on $\boldsymbol{h}_{t-1}$ and $\boldsymbol{x}_t$ in a recurrent cell can be written as

$$\boldsymbol{h}_t = \sigma(\mathbf{U}\boldsymbol{h}_{t-1} + \mathbf{W}\boldsymbol{x}_t + \boldsymbol{b}),\ \boldsymbol{x}_t \in \mathbb{R}^d,\ \text{and}\ \boldsymbol{h}_{t-1}, \boldsymbol{h}_t \in \mathbb{R}^h,\quad t = 1, 2, \cdots, T, \tag{1}$$

where $\mathbf{U} \in \mathbb{R}^{h\times h}, \mathbf{W} \in \mathbb{R}^{h\times d}$, and $\boldsymbol{b} \in \mathbb{R}^h$ are trainable parameters; $\sigma(\cdot)$ is a nonlinear activation function, e.g., sigmoid or hyperbolic tangent. Error backpropagation in time is used to train RNN, but it tends to result in exploding or vanishing gradients [4]. Thus RNNs may fail to learn long term dependencies. Several approaches exist to improve RNNs' performance, including enforcing unitary weight matrices [1, 21, 24, 34, 55, 57] and leveraging LSTM cells.

LSTM cells augment the recurrent cell with "gates" [23] and can be formulated as

$$\begin{aligned}
\boldsymbol{i}_t &= \sigma(\mathbf{U}_{ih}\boldsymbol{h}_{t-1} + \mathbf{W}_{ix}\boldsymbol{x}_t + \boldsymbol{b}_i), && (\boldsymbol{i}_t : \text{input gate})\\
\widetilde{\boldsymbol{c}}_t &= \tanh\left(\mathbf{U}_{\widetilde{c}h}\boldsymbol{h}_{t-1} + \mathbf{W}_{\widetilde{c}x}\boldsymbol{x}_t + \boldsymbol{b}_{\widetilde{c}}\right), && (\widetilde{\boldsymbol{c}}_t : \text{cell input})\\
\boldsymbol{c}_t &= \boldsymbol{c}_{t-1} + \boldsymbol{i}_t \odot \widetilde{\boldsymbol{c}}_t, && (\boldsymbol{c}_t : \text{cell state})\\
\boldsymbol{o}_t &= \sigma(\mathbf{U}_{oh}\boldsymbol{h}_{t-1} + \mathbf{W}_{ox}\boldsymbol{x}_t + \boldsymbol{b}_o), && (\boldsymbol{o}_t : \text{output gate})\\
\boldsymbol{h}_t &= \boldsymbol{o}_t \odot \tanh \boldsymbol{c}_t, && (\boldsymbol{h}_t : \text{hidden state})
\end{aligned} \tag{2}$$

where $\mathbf{U}_* \in \mathbb{R}^{h\times h}$, $\mathbf{W}_* \in \mathbb{R}^{h\times d}$, and $\boldsymbol{b}_* \in \mathbb{R}^h$ are learnable parameters, and $\odot$ denotes the Hadamard product. The input gate decides what new information to be stored in the cell state, and the output gate decides what information to output based on the cell state value.

1.2 Our Contributions

In this paper, we develop a gradient descent (GD) analogy of the recurrent cell. In particular, the hidden state update in a recurrent cell is associated with a gradient descent step towards the optimal representation of the hidden state. We then propose to integrate momentum that used for accelerating gradient dynamics into the recurrent cell, which results in the momentum cell. At the core of the momentum cell is the use of momentum to accelerate the hidden state learning in RNNs. The architectures of the standard recurrent cell and our momentum cell are illustrated in Fig. 1. We provide the design principle and detailed derivation of the momentum cell in Sections 2.2 and 2.4. We call the RNN that consists of momentum cells the MomentumRNN. The major advantages of MomentumRNN are fourfold:

- MomentumRNN can alleviate the vanishing gradient problem in training RNN.
- MomentumRNN accelerates training and improves the test accuracy of the baseline RNN.
- MomentumRNN is universally applicable to many existing RNNs. It can be easily implemented by changing a few lines of the baseline RNN code.
- MomentumRNN is principled with theoretical guarantees provided by the momentum-accelerated dynamical system for optimization and sampling. The design principle can be generalized to other advanced momentum-based optimization methods, including Adam [27] and Nesterov accelerated gradients with a restart [40, 56].

1.3 Related Work

Dynamical system viewpoint of RNNs. Leveraging the theory of dynamical system to improve RNNs has been an interesting research direction: [28] proposes a gated RNN, which is principled by non-chaotical dynamical systems and achieves comparable performance to GRUs and LSTMs. [53] proposes a weight initialization strategy inspired by dynamical system theory, which helps the training of RNNs with ReLU nonlinearity. Other RNN algorithms derived from the dynamical system theories include [8, 9, 25, 41]. Our

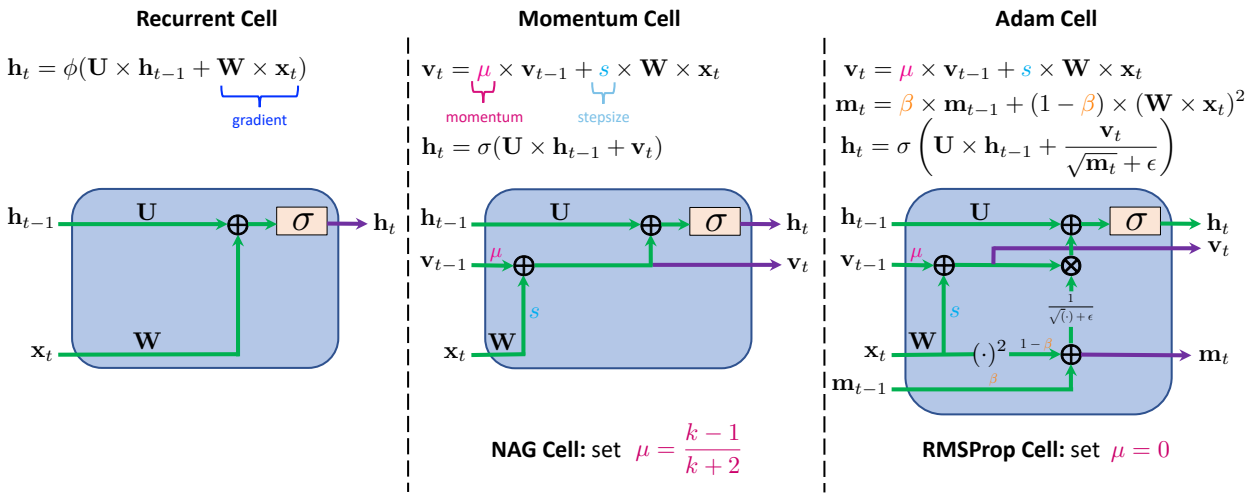

Figure 1: Illustration of the recurrent cell (left), Momentum/NAG cell (middle), and Adam/RMSProp cell (right). We draw a connection between the dynamics of hidden states in the recurrent cell and GD. We then introduce momentum to recurrent cell as an analogy of the momentum accelerated GD.

work is the first that directly integrates momentum into an RNN to accelerate the underlying dynamics and improve the model's performance.

Momentum in Optimization and Sampling. Momentum has been a popular technique for accelerating (stochastic) gradient-based optimization [3, 19, 27, 44, 45, 51] and sampling algorithms [12, 37] A particularly interesting momentum is the iteration-dependent one in NAG [2, 39, 40], which has a significantly better convergence rate than constant momentum for convex optimization. The stochastic gradient NAG that employs a scheduled restart can also be used to accelerate DNN training with better accuracy and faster convergence [56].

Momentum in DNNs. Momentum has also been used in designing DNN architectures. [20] develops momentum contrast as a way of building large and consistent dictionaries for unsupervised learning with contrastive loss. At the core of this approach is a momentum-based moving average of the queue encoder. Many DNN-based algorithms for sparse coding are designed by unfolding the classical optimization algorithms, e.g., FISTA [2], in which momentum can be used in the underpinning optimizer [7, 26, 32, 36, 52].

1.4 Notation

We denote scalars by lower or upper case letters; vectors and matrices by lower and upper case bold face letters, respectively. For a vector $\boldsymbol{x} = (x_1, \cdots, x_d)^T \in \mathbb{R}^d$, we use $\|\boldsymbol{x}\| = (\sum_{i=1}^d |x_i|^2)^{1/2}$ to denote its ℓ_2 norm. For a matrix $\mathbf{A}$, we use $\mathbf{A}^{\mathrm{T}}$ (T in roman type) and $\mathbf{A}^{-1}$ to denote its transpose and inverse, respectively. Also, we denote the spectral norm of $\mathbf{A}$ as $\|\mathbf{A}\|$. We denote the d-dimensional standard Gaussian as $\mathcal{N}(\mathbf{0}, \mathbf{I}_{d\times d})$, where $\mathbf{0}$ is the d-dimensional zero-vector and $\mathbf{I}_{d\times d}$ is an identity matrix. For a function $\phi(\boldsymbol{x}) : \mathbb{R}^d \to \mathbb{R}$, we denote $\phi^{-1}(\boldsymbol{x})$ as its inverse and $\nabla\phi(\boldsymbol{x})$ as its gradient.

2 Momentum RNNs

2.1 Background: Momentum Acceleration for Gradient Based Optimization and Sampling

Momentum has been successfully used to accelerate the gradient-based algorithms for optimization and sampling. In optimization, we aim to find a stationary point of a given function $f(\boldsymbol{x}), \boldsymbol{x} \in \mathbb{R}^d$. Starting from $\boldsymbol{x}_0 \in \mathbb{R}^d$, GD iterates as $\boldsymbol{x}_t = \boldsymbol{x}_{t-1} - s\nabla f(\boldsymbol{x}_t)$ with $s > 0$ being the step size. This can be significantly

accelerated by using the momentum [51], which results in

$$\boldsymbol{p}_0 = \boldsymbol{x}_0;\ \boldsymbol{p}_t = \mu \boldsymbol{p}_{t-1} + s\nabla f(\boldsymbol{x}_t);\ \boldsymbol{x}_t = \boldsymbol{x}_{t-1} - \boldsymbol{p}_t,\quad t \geq 1, \tag{3}$$

where $\mu \geq 0$ is the momentum constant. In sampling, Langevin Monte Carlo (LMC) [11] is used to sample from the distribution $\pi \propto \exp\{-f(\boldsymbol{x})\}$, where $\exp\{-f(\boldsymbol{x})\}$ is the probability distribution function. The update at each iteration is given by

$$\boldsymbol{x}_t = \boldsymbol{x}_{t-1} - s\nabla f(\boldsymbol{x}_t) + \sqrt{2s}\boldsymbol{\epsilon}_t,\ s \geq 0,\ t \geq 1,\ \boldsymbol{\epsilon}_t \sim \mathcal{N}(\mathbf{0}, \mathbf{I}_{d\times d}). \tag{4}$$

We can also use momentum to accelerate LMC, which results in the following Hamiltonian Monte Carlo (HMC) update [11]:

$$\boldsymbol{p}_0 = \boldsymbol{x}_0;\ \boldsymbol{p}_t = \boldsymbol{p}_{t-1} - \gamma s \boldsymbol{p}_{t-1} - s\eta\nabla f(\boldsymbol{x}_{t-1}) + \sqrt{2\gamma s\eta}\boldsymbol{\epsilon}_t;\ \boldsymbol{x}_t = \boldsymbol{x}_{t-1} + s\boldsymbol{p}_t,\quad t \geq 1, \tag{5}$$

where $\boldsymbol{\epsilon}_t \sim \mathcal{N}(\mathbf{0}, \mathbf{I}_{d\times d})$ while $\gamma, \eta, s > 0$ are the friction parameter, inverse mass, and step size, resp.

2.2 Gradient Descent Analogy for RNN and MomentumRNN

Now, we are going to establish a connection between RNN and GD, and further leverage momentum to improve RNNs. Let $\widetilde{\mathbf{W}} = [\mathbf{W}, \boldsymbol{b}]$ and $\widetilde{\boldsymbol{x}}_t = [\boldsymbol{x}_t, 1]^T$ in (1), then we have $\boldsymbol{h}_t = \sigma(\mathbf{U}\boldsymbol{h}_{t-1} + \widetilde{\mathbf{W}}\widetilde{\boldsymbol{x}}_t)$. For the ease of notation, without ambiguity we denote $\mathbf{W} := \widetilde{\mathbf{W}}$ and $\boldsymbol{x}_t := \widetilde{\boldsymbol{x}}_t$. Then the recurrent cell can be reformulated as

$$\boldsymbol{h}_t = \sigma(\mathbf{U}\boldsymbol{h}_{t-1} + \mathbf{W}\boldsymbol{x}_t). \tag{6}$$

Moreover, let $\phi(\cdot) := \sigma(\mathbf{U}(\cdot))$ and $\boldsymbol{u}_t := \mathbf{U}^{-1}\mathbf{W}\boldsymbol{x}_t$, we can rewrite (6) as

$$\boldsymbol{h}_t = \phi(\boldsymbol{h}_{t-1} + \boldsymbol{u}_t). \tag{7}$$

If we regard $-\boldsymbol{u}_t$ as the "gradient" at the t-th iteration, then we can consider (7) as the dynamical system which updates the hidden state by the gradient and then transforms the updated hidden state by the nonlinear activation function ϕ. We propose the following accelerated dynamical system to accelerate the dynamics of (7), which is principled by the accelerated gradient descent theory (see subsection 2.1):

$$\boldsymbol{p}_t = \mu\boldsymbol{p}_{t-1} - s\boldsymbol{u}_t;\quad \boldsymbol{h}_t = \phi(\boldsymbol{h}_{t-1} - \boldsymbol{p}_t), \tag{8}$$

where $\mu \geq 0, s > 0$ are two hyperparameters, which are the analogies of the momentum coefficient and step size in the momentum-accelerated GD, respectively. Let $\boldsymbol{v}_t := -\mathbf{U}\boldsymbol{p}_t$, we arrive at the following dynamical system:

$$\boldsymbol{v}_t = \mu\boldsymbol{v}_{t-1} + s\mathbf{W}\boldsymbol{x}_t;\quad \boldsymbol{h}_t = \sigma(\mathbf{U}\boldsymbol{h}_{t-1} + \boldsymbol{v}_t). \tag{9}$$

The architecture of the momentum cell that corresponds to the dynamical system (9) is plotted in Fig. 1 (middle). Compared with the recurrent cell, the momentum cell introduces an auxiliary momentum state in each update and scales the dynamical system with two positive constants μ and s.

Remark 1. *Different parameterizations of* (8) *can result in different momentum cell architectures. For instance, if we let* $\boldsymbol{v}_t = -\boldsymbol{p}_t$*, we end up with the following dynamical system:*

$$\boldsymbol{v}_t = \mu\boldsymbol{v}_{t-1} + s\widehat{\mathbf{W}}\boldsymbol{x}_t;\quad \boldsymbol{h}_t = \sigma(\mathbf{U}\boldsymbol{h}_{t-1} + \mathbf{U}\boldsymbol{v}_t), \tag{10}$$

where $\widehat{\mathbf{W}} := \mathbf{U}^{-1}\mathbf{W}$ *is the trainable weight matrix. Even though* (9) *and* (10) *are mathematically equivalent, the training procedure might cause the MomentumRNNs that are derived from different parameterizations to have different performances.*

Remark 2. *We put the nonlinear activation in the second equation of* (8) *to ensure that the value of* $\boldsymbol{h}_t$ *is in the same range as the original recurrent cell.*

Remark 3. *The derivation above also applies to the dynamical systems in the LSTM cells, and we can design the MomentumLSTM in the same way as designing the MomentumRNN.*

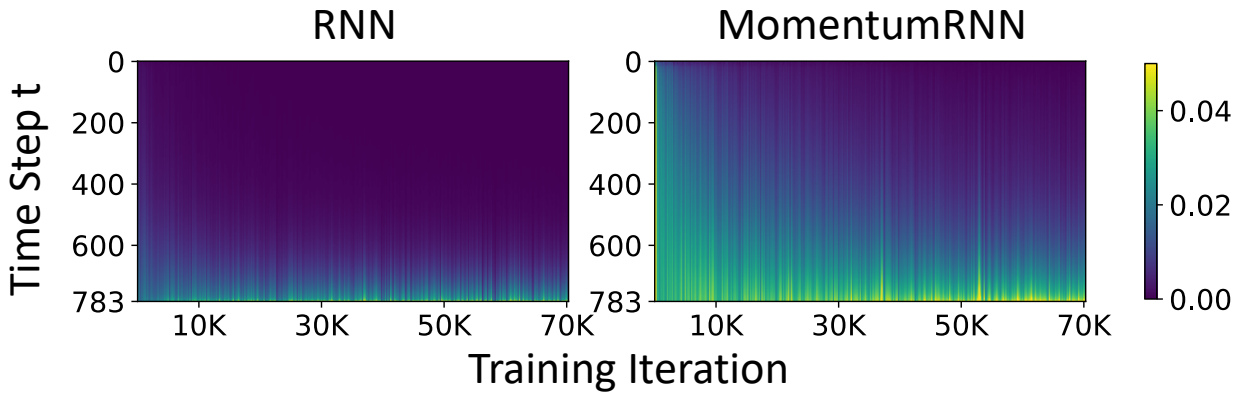

Figure 2: ℓ_2 norm of the gradients of the loss $\mathcal{L}$ w.r.t. the state vector $\boldsymbol{h}_t$ at each time step t for RNN (left) and MomentumRNN (right). MomentumRNN does not suffer from vanishing gradients.

2.3 Analysis of the Vanishing Gradient Issue: Momentum Cell vs. Recurrent Cell

Let $\boldsymbol{h}_T$ and $\boldsymbol{h}_t$ be the state vectors at the time step T and t, respectively, and we suppose $T \gg t$. Furthermore, assume that $\mathcal{L}$ is the objective to minimize, then

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{h}_t} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{h}_T} \cdot \frac{\partial \boldsymbol{h}_T}{\partial \boldsymbol{h}_t} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{h}_T} \cdot \prod_{k=t}^{T-1} \frac{\partial \boldsymbol{h}_{k+1}}{\partial \boldsymbol{h}_k} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{h}_T} \cdot \prod_{k=t}^{T-1} (\mathbf{D}_k \mathbf{U}^{\mathrm{T}}), \tag{11}$$

where $\mathbf{U}^{\mathrm{T}}$ is the transpose of $\mathbf{U}$ and $\mathbf{D}_k = \mathrm{diag}(\sigma'(\mathbf{U}\boldsymbol{h}_k + \mathbf{W}\boldsymbol{x}_{k+1}))$ is a diagonal matrix with $\sigma'(\mathbf{U}\boldsymbol{h}_k + \mathbf{W}\boldsymbol{x}_{k+1})$ being its diagonal entries. $\| \prod_{k=t}^{T-1} (\mathbf{D}_k \mathbf{U}^{\mathrm{T}}) \|_2$ tends to either vanish or explode [4]. We can use regularization or gradient clipping to mitigate the exploding gradient, leaving vanishing gradient as the major obstacle to training RNN to learn long-term dependency [43]. We can rewrite (9) as

$$\boldsymbol{h}_t = \sigma\left(\mathbf{U}(\boldsymbol{h}_{t-1} + \mu \boldsymbol{h}_{t-2}) + \mu \sigma^{-1}(\boldsymbol{h}_{t-1}) + s\mathbf{W}\boldsymbol{x}_t\right), \tag{12}$$

where $\sigma^{-1}(\cdot)$ is the inverse function of $\sigma(\cdot)$. We compute $\partial \mathcal{L}/\partial \boldsymbol{h}_t$ as follows

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{h}_t} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{h}_T} \cdot \frac{\partial \boldsymbol{h}_T}{\partial \boldsymbol{h}_t} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{h}_T} \cdot \prod_{k=t}^{T-1} \frac{\partial \boldsymbol{h}_{k+1}}{\partial \boldsymbol{h}_k} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{h}_T} \cdot \prod_{k=t}^{T-1} \widehat{\mathbf{D}}_k [\mathbf{U}^{\mathrm{T}} + \mu \boldsymbol{\Sigma}_k], \tag{13}$$

where $\widehat{\mathbf{D}}_k = \mathrm{diag}(\sigma'(\mathbf{U}(\boldsymbol{h}_k + \mu \boldsymbol{h}_{k-1}) + \mu \sigma^{-1}(\boldsymbol{h}_k) + s\mathbf{W}\boldsymbol{x}_{k+1}))$ and $\boldsymbol{\Sigma} = \mathrm{diag}((\sigma^{-1})'(\boldsymbol{h}_k))$. For mostly used σ, e.g., sigmoid and tanh, $(\sigma^{-1}(\cdot))' > 1$ and $\mu \boldsymbol{\Sigma}_k$ dominants $\mathbf{U}^{\mathrm{T}}$.[1] Therefore, with an appropriate choice of μ, the momentum cell can alleviate vanishing gradient and accelerate training.

We empirically corroborate that momentum cells can alleviate vanishing gradients by training a MomentumRNN and its corresponding RNN on the PMNIST classification task and plot $\|\partial \mathcal{L}/\partial \boldsymbol{h}_t\|_2$ for each time step t. Figure 2 confirms that unlike in RNN, the gradients in MomentumRNN do not vanish. More details on this experiment are provided in the Appendix A.

2.4 Beyond MomentumRNN: NAG and Adam Principled Recurrent Neural Nets

There are several other advanced formalisms of momentum existing in optimization, which can be leveraged for RNN architecture design. In this subsection, we present two additional variants of MomentumRNN that

[1] In the vanishing gradient scenario, $\|\mathbf{U}\|_2$ is small; also it can be controlled by regularizing the loss function.

are derived from the Nesterov accelerated gradient (NAG)-style momentum with restart [40, 56] and Adam [27].

NAG Principled RNNs. The momentum-accelerated GD can be further accelerated by replacing the constant momentum coefficient μ in (9) with the NAG-style momentum, i.e. setting μ to $(t-1)/(t+2)$ at the t-th iteration. Furthermore, we can accelerate NAG by resetting the momentum to 0 after every F iterations, i.e. $\mu = (t \mod F)/((t \mod F)+3)$, which is the NAG-style momentum with a scheduled restart of the appropriately selected frequency F [56]. For convex optimization, NAG has a convergence rate $O(1/t^2)$, which is significantly faster than GD or GD with constant momentum whose convergence rate is $O(1/t)$. Scheduled restart not only accelerates NAG to a linear convergence rate $O(\alpha^{-t})(0<\alpha<1)$ under mild extra assumptions but also stabilizes the NAG iteration [56]. We call the MomentumRNN with the NAG-style momentum and scheduled restart momentum the NAG-based RNN and the scheduled restart RNN (SRRNN), respectively.

Adam Principled RNNs. Adam [27] leverages the moving average of historical gradients and entry-wise squared gradients to accelerate the stochastic gradient dynamics. We use Adam to accelerate (7) and end up with the following iteration

$$\boldsymbol{p}_t = \mu \boldsymbol{p}_{t-1} - (1-\mu)\boldsymbol{u}_t;\ \boldsymbol{m}_t = \beta \boldsymbol{m}_{t-1} + (1-\beta)\boldsymbol{u}_t \odot \boldsymbol{u}_t;\ \boldsymbol{h}_t = \phi\left(\boldsymbol{h}_{t-1} - s\frac{\boldsymbol{p}_t}{\sqrt{\boldsymbol{r}_t}+\epsilon}\right), \tag{14}$$

where $\mu, s, \beta > 0$ are hyperparameters, ϵ is a small constant and chosen to be 10^{-8} by default, and $\odot/\sqrt{\cdot}$ denotes the entrywise product/square root[2]. Again, let $\boldsymbol{v}_t = -\mathbf{U}\boldsymbol{p}_t$, we rewrite (14) as follows

$$\boldsymbol{v}_t = \mu \boldsymbol{v}_{t-1} + (1-\mu)\mathbf{W}\boldsymbol{x}_t;\ \boldsymbol{m}_t = \beta \boldsymbol{m}_{t-1} + (1-\beta)\boldsymbol{u}_t \odot \boldsymbol{u}_t;\ \boldsymbol{h}_t = \sigma\left(\mathbf{U}\boldsymbol{h}_{t-1} + s\frac{\boldsymbol{v}_t}{\sqrt{\boldsymbol{m}_t}+\epsilon}\right).$$

As before, here $\boldsymbol{u}_t := \mathbf{U}^{-1}\mathbf{W}\boldsymbol{x}_t$. Computing $\mathbf{U}^{-1}$ is expensive. Our experiments suggest that replacing $\boldsymbol{u}_t \odot \boldsymbol{u}_t$ by $\mathbf{W}\boldsymbol{x}_t \odot \mathbf{W}\boldsymbol{x}_t$ is sufficient and more efficient to compute. In our implementation, we also relax $\boldsymbol{v}_t = \mu\boldsymbol{v}_{t-1} + (1-\mu)\mathbf{W}\boldsymbol{x}_t$ to $\boldsymbol{v}_t = \mu\boldsymbol{v}_{t-1} + s\mathbf{W}\boldsymbol{x}_t$ that follows the momentum in the MomentumRNN (9) for better performance. Therefore, we propose the *AdamRNN* that is given by

$$\boldsymbol{v}_t = \mu\boldsymbol{v}_{t-1} + s\mathbf{W}\boldsymbol{x}_t;\quad \boldsymbol{m}_t = \beta\boldsymbol{m}_{t-1} + (1-\beta)(\mathbf{W}\boldsymbol{x}_t \odot \mathbf{W}\boldsymbol{x}_t);\quad \boldsymbol{h}_t = \sigma\left(\mathbf{U}\boldsymbol{h}_{t-1} + \frac{\boldsymbol{v}_t}{\sqrt{\boldsymbol{m}_t}+\epsilon}\right). \tag{15}$$

In AdamRNN, if μ is set to 0, we achieve another new RNN, which obeys the RMSProp gradient update rule [54]. We call this new model the *RMSPropRNN*.

Remark 4. *Both AdamRNN and RMSPropRNN can also be derived by letting $\boldsymbol{v}_t = -\boldsymbol{p}_t$ and $\widehat{\mathbf{W}} := \mathbf{U}^{-1}\mathbf{W}$ as in Remark 1. This parameterization yields the following formulation for AdamRNN*

$$\boldsymbol{v}_t = \mu\boldsymbol{v}_{t-1} + s\widehat{\mathbf{W}}\boldsymbol{x}_t;\quad \boldsymbol{m}_t = \beta\boldsymbol{m}_{t-1} + (1-\beta)(\widehat{\mathbf{W}}\boldsymbol{x}_t \odot \widehat{\mathbf{W}}\boldsymbol{x}_t);\quad \boldsymbol{h}_t = \sigma\left(\mathbf{U}\boldsymbol{h}_{t-1} + \frac{\mathbf{U}\boldsymbol{v}_t}{\sqrt{\boldsymbol{m}_t}+\epsilon}\right).$$

Here, we simply need to learn $\widehat{\mathbf{W}}$ and $\mathbf{U}$ without any relaxation. In contrast, we relaxed $\mathbf{U}^{-1}$ to an identity matrix in (15)*. Our experiments suggest that both parameterizations yield similar results.*

3 Experimental Results

In this section, we evaluate the effectiveness of our momentum approach in designing RNNs in terms of convergence speed and accuracy. We compare the performance of the MomentumLSTM with the baseline LSTM [23] in the following tasks: 1) the object classification task on pixel-permuted MNIST [29], 2) the speech prediction task on the TIMIT dataset [1, 21, 22, 34, 57], and 3) the language modeling task on the Penn TreeBank (PTB) dataset [35]. These three tasks are among standard benchmarks to measure the performance of RNNs and their ability to handle long-term dependencies. Also, these tasks cover different data modalities – image, speech, and text data – as well as a variety of model sizes, ranging from thousands to millions of

[2]In contrast to Adam, we do not normalize $\boldsymbol{p}_t$ and $\boldsymbol{m}_t$ since they can be absorbed in the weight matrices.

Table 1: Best test accuracy at the MNIST and PMNIST tasks (%). We use the baseline results reported in [21], [57], [55]. All of our proposed models outperform the baseline LSTM. Among the models using $N = 256$ hidden units, RMSPropLSTM yields the best results in both tasks.

Model	n	# params	MNIST	PMNIST
LSTM	128	$\approx 68K$	98.70[21],97.30 [55]	92.00 [21],92.62 [55]
LSTM	256	$\approx 270K$	98.90 [21], 98.50 [57]	92.29 [21], 92.10 [57]
MomentumLSTM	128	$\approx 68K$	$\mathbf{99.04 \pm 0.04}$	$\mathbf{93.40 \pm 0.25}$
MomentumLSTM	256	$\approx 270K$	$\mathbf{99.08 \pm 0.05}$	$\mathbf{94.72 \pm 0.16}$
AdamLSTM	256	$\approx 270K$	99.09 ± 0.03	95.05 ± 0.37
RMSPropLSTM	256	$\approx 270K$	$\mathbf{99.15 \pm 0.06}$	$\mathbf{95.38 \pm 0.19}$
SRLSTM	256	$\approx 270K$	99.01 ± 0.07	93.82 ± 1.85

parameters with one (MNIST and TIMIT tasks) or multiple (PTB task) recurrent cells in concatenation. Our experimental results confirm that MomentumLSTM converges faster and yields better test accuracy than the baseline LSTM across tasks and settings. We also discuss the AdamLSTM, RMSPropLSTM, and scheduled restart LSTM (SRLSTM) and show their advantage over MomentumLSTM in specific tasks. Computation time and memory cost of our models versus the baseline LSTM are provided in Appendix D. All of our results are averaged over 5 runs with different seeds. We include details on the models, datasets, training procedure, and hyperparameters used in our experiments in Appendix A. For MNIST and TIMIT experiments, we use the baseline codebase provided by [5]. For PTB experiments, we use the baseline codebase provided by [50].

3.1 Pixel-by-Pixel MNIST

In this task, we classify image samples of hand-written digits from the MNIST dataset [30] into one of the ten classes. Following the implementation of [29], we flatten the image of original size 28×28 pixels and feed it into the model as a sequence of length 784. In the unpermuted task (MNIST), the sequence of pixels is processed row-by-row. In the permuted task (PMNIST), a fixed permutation is selected at the beginning of the experiments and then applied to both training and test sequences. We summarize the results in Table 1. Our experiments show that *MomentumLSTM achieves better test accuracy than the baseline LSTM in both MNIST and PMNIST digit classification tasks* using different numbers of hidden units (i.e. $N = 128, 256$). Especially, the improvement is significant on the PMNIST task, which is designed to test the performance of RNNs in the context of long-term memory. Furthermore, we notice that *MomentumLSTM converges faster than LSTM* in all settings. Figure 3 (left two panels) corroborates this observation when using $N = 256$ hidden units.

3.2 TIMIT Speech Dataset

We study how MomentumLSTM performs on audio data with speech prediction experiments on the TIMIT speech dataset [15], which is a collection of real-world speech recordings. As first proposed by [57], the recordings are downsampled to 8kHz and then transformed into log-magnitudes via a short-time Fourier transform (STFT). The task accounts for predicting the next log-magnitude given the previous ones. We use the standard train/validation/test separation in [6, 31, 57], thereby having 3640 utterances for the training set with a validation set of size 192 and a test set of size 400.

The results for this TIMIT speech prediction are shown in Table 2. Results are reported on the test set using the model parameters that yield the best validation loss. Again, we see the advantage of MomentumLSTM over the baseline LSTM. In particular, MomentumLSTM yields much better prediction accuracy and faster convergence speed compared to LSTM. Figure 3 (right two panels) shows the convergence of MomentumLSTM vs. LSTM when using $N = 158$ hidden units.

Remark: The TIMIT dataset is not open for public, so we do not have access to the preprocessed data from previous papers. We followed the data preprocessing in [6, 31, 57] to generate the preprocessed data for our experiments and did our best to reproduce the baseline results. In Table 2 and 5, we include both our reproduced results and the ones reported from previous works.

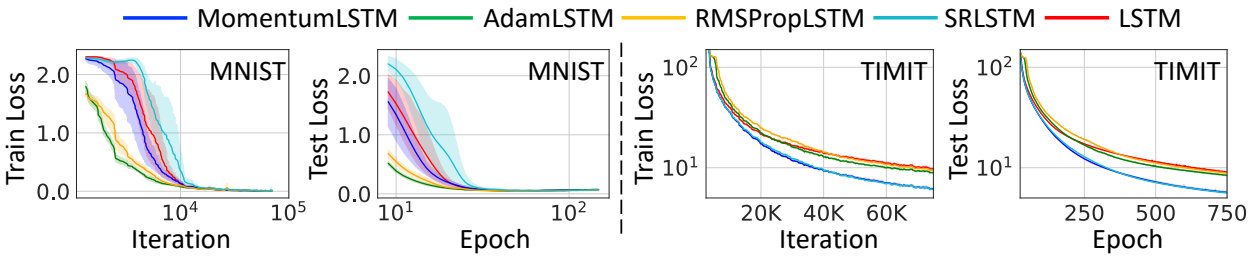

Figure 3: Train and test loss of MomentumLSTM (blue), AdamLSTM (green), RMSPropLSTM (orange), SRLSTM (cyan), and LSTM (red) for MNIST (left two panels) and TIMIT (right two panels) tasks. MomentumLSTM converges faster than LSTM in both tasks. For MNIST, AdamLSTM and RMSPropLSTM converge fastest. For TIMIT, MomentumLSTM and SRLSTM converge fastest.

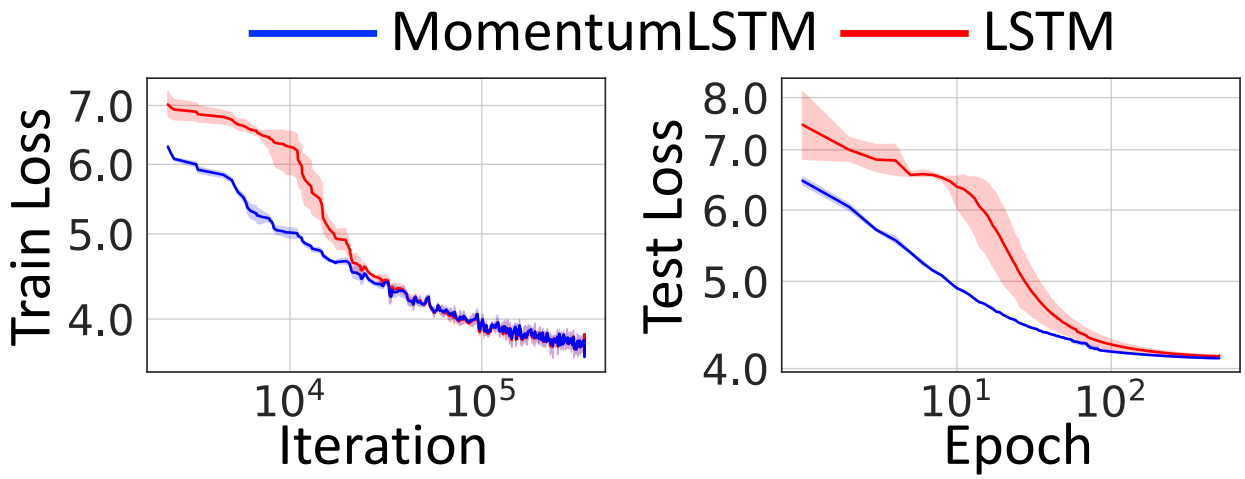

Figure 4: Train (left) and test loss (right) of MomentumLSTM (blue) and LSTM (red) for the Penn Treebank language modeling tasks at word level.

Table 2: Test and validation MSEs at the end of the epoch with the lowest validation MSE for the TIMIT task. All of our proposed models outperform the baseline LSTM. Among models using $N = 158$ hidden units, SRLSTM performs the best.

Model	n	# params	Val. MSE	Test MSE
LSTM	84	$\approx 83K$	14.87 ± 0.15 (15.42 [21, 31])	14.94 ± 0.15 (14.30 [21, 31])
LSTM	120	$\approx 135K$	11.77 ± 0.14 (13.93 [21, 31])	11.83 ± 0.12 (12.95 [21, 31])
LSTM	158	$\approx 200K$	9.33 ± 0.14 (13.66 [21, 31])	9.37 ± 0.14 (12.62 [21, 31])
MomentumLSTM	84	$\approx 83K$	$\mathbf{10.90 \pm 0.19}$	$\mathbf{10.98 \pm 0.18}$
MomentumLSTM	120	$\approx 135K$	$\mathbf{8.00 \pm 0.30}$	$\mathbf{8.04 \pm 0.30}$
MomentumLSTM	158	$\approx 200K$	$\mathbf{5.86 \pm 0.14}$	$\mathbf{5.87 \pm 0.15}$
AdamLSTM	158	$\approx 200K$	8.66 ± 0.15	8.69 ± 0.14
RMSPropLSTM	158	$\approx 200K$	9.13 ± 0.33	9.17 ± 0.33
SRLSTM	158	$\approx 200K$	$\mathbf{5.81 \pm 0.10}$	$\mathbf{5.83 \pm 0.10}$

Table 3: Model test perplexity at the end of the epoch with the lowest validation perplexity for the Penn Treebank language modeling task (word level).

Model	# params	Val. PPL	Test PPL
lstm	$\approx 24M$	61.96 ± 0.83	59.71 ± 0.99 (58.80 [33])
MomentumLSTM	$\approx 24M$	$\mathbf{60.71 \pm 0.24}$	$\mathbf{58.62 \pm 0.22}$

Table 4: Best test accuracy on the PMNIST tasks (%) for MomentumDTRIV and DTRIV. We provide both our reproduced baseline results and those reported in [6]. MomentumDTRIV yields better results than the baseline DTRIV in all settings.

n	# params	PMNIST (DTRIV)	PMNIST (MomentumDTRIV)
170	$\approx 16K$	95.21 ± 0.10 (95.20 [6])	$\mathbf{95.37 \pm 0.09}$
360	$\approx 69K$	96.45 ± 0.10 (96.50 [6])	$\mathbf{96.73 \pm 0.08}$
512	$\approx 137K$	96.62 ± 0.12 (96.80 [6])	$\mathbf{96.89 \pm 0.08}$

3.3 Word-Level Penn TreeBank

To study the advantage of MomentumLSTM over LSTM on text data, we perform language modeling over a preprocessed version of the PTB dataset [35], which has been a standard benchmark for evaluating language models. Unlike the baselines used in the (P)MNIST and TIMIT experiments which contain one LSTM cell, in this PTB experiment, we use a three-layer LSTM model, which contains three concatenated LSTM cells, as the baseline. The size of this model in terms of the number of parameters is also much larger than those in the (P)MNIST and TIMIT experiments. Table 3 shows the test and validation perplexity (PPL) using the model parameters that yield the best validation loss. Again, MomentumLSTM achieves better perplexities and converges faster than the baseline LSTM (see Figure 4).

3.4 NAG and Adam Principled Recurrent Neural Nets

We evaluate AdamLSTM, RMSPropLSTM and SRLSTM on (P)MNIST classification and TIMIT speech recognition tasks. We summarize the test accuracy of the trained models in Tables 1 and 2 and provide the plots of train and test losses in Figure 3. We observe that though AdamLSTM and RMSPropLSTM work better than the MomentumLSTM at (P)MNIST task, they yield worse results at the TIMIT task. Interestingly, SRLSTM shows an opposite behavior - better than MomentunLSTM at TIMIT task but worse at (P)MNIST task. This is somewhat expected, given the connection between our model and its analogy to optimization algorithm. An optimizer needs to be chosen for each particular task, and so is for our MomentumRNN. All of our models outperform the baseline LSTM.

4 Additional Results and Analysis

Beyond LSTM. Our interpretation of hidden state dynamics in RNNs as GD steps and the use of momentum to accelerate the convergence speed and improve the generalization of the model apply to many types of RNNs but not only LSTM. We show the applicability of our momentum-based design approach beyond LSTM by performing PMNIST and TIMIT experiments using the orthogonal RNN equipped with dynamic trivialization (DTRIV) [6]. DTRIV is currently among state-of-the-art models for PMNIST digit classification and TIMIT speech prediction tasks. Tables 4 and 5 consist of results for our method, namely MomentumDTRIV, in comparison with the baseline results. Again, MomentumDTRIV outperforms the baseline DTRIV by a margin in both PMNIST and TIMIT tasks while converging faster and overfitting less (see Figure 5).

Table 5: Test and validation MSE of MomentumDTRIV vs. DTRIV at the epoch with the lowest validation MSE for the TIMIT task. MomentumDTRIV yields much better results than DTRIV.

Model	n	# params	Val. MSE	Test MSE
DTRIV	224	$\approx 83K$	4.74 ± 0.06 (4.75 [6])	4.70 ± 0.07 (4.71 [6])
DTRIV	322	$\approx 135K$	1.92 ± 0.17 (3.39 [6])	1.87 ± 0.17 (3.76 [6])
MomentumDTRIV	224	$\approx 83K$	$\mathbf{3.10 \pm 0.09}$	$\mathbf{3.06 \pm 0.09}$
MomentumDTRIV	322	$\approx 135K$	$\mathbf{1.21 \pm 0.05}$	$\mathbf{1.17 \pm 0.05}$

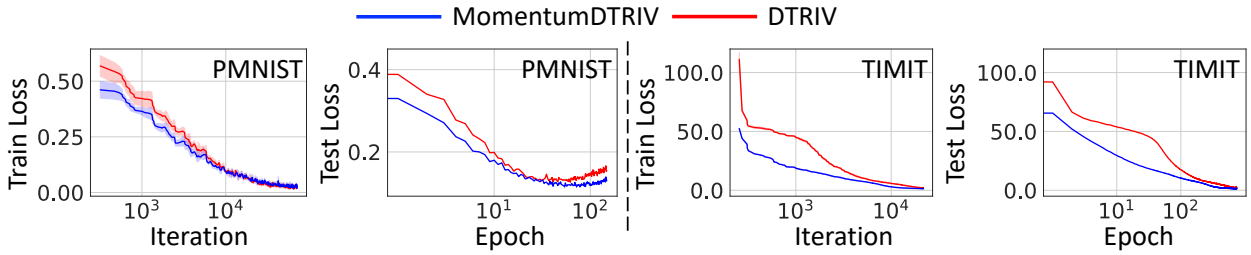

Figure 5: Train and test loss of MomentumDTRIV (blue) and DTRIV (red) for PMNIST (left two panels) and TIMIT (right two panels) tasks. MomentumDTRIV converges faster than DTRIV in both tasks. For PMNIST task, DTRIV suffers from overtting while MomentumDTRIV overfits less.

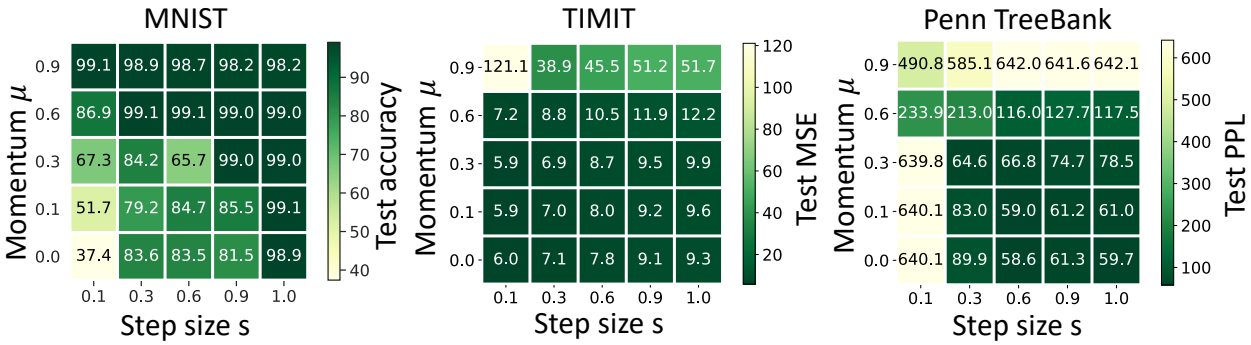

Figure 6: Ablation study of the effects of momentum and step size on MomentumLSTM's performance. We use $N = 256/158$ hidden units for MNIST/TIMIT task. Green denotes better results.

Effects of Momentum and Step Size. To better understand the effects of momentum and step size on the final performance of the trained MomentumLSTM models, we do an ablation study and include the results in Figure 6. The result in each cell is averaged over 5 runs.

5 Conclusion

In this paper, we propose a universal framework for integrating momentum into RNNs. The resulting MomentumRNN achieves significant acceleration in training and remarkably better performance on the benchmark sequential data prediction tasks over the RNN counterpart. From a theoretical viewpoint, it would be interesting to derive a theory to decipher why training MomentumRNN converges faster and generalizes better. From the neural architecture design perspective, it would be interesting to integrate momentum into the design of the standard convolutional and graph convolutional neural nets. Moreover, the current

MomentumRNN requires calibration of the momentum and step size-related hyperparameters; developing an adaptive momentum for MomentumRNN is of interest.

References

[1] Martin Arjovsky, Amar Shah, and Yoshua Bengio. Unitary evolution recurrent neural networks. In *International Conference on Machine Learning*, pages 1120–1128, 2016.

[2] Amir Beck and Marc Teboulle. A fast iterative shrinkage-thresholding algorithm for linear inverse problems. *SIAM Journal on Imaging Sciences*, 2(1):183–202, 2009.

[3] Yoshua Bengio, Nicolas Boulanger-Lewandowski, and Razvan Pascanu. Advances in optimizing recurrent networks. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 8624–8628. IEEE, 2013.

[4] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166, 1994.

[5] Mario Lezcano Casado. Optimization with orthogonal constraints and on general manifolds. `https://github.com/Lezcano/expRNN`, 2019.

[6] Mario Lezcano Casado. Trivializations for gradient-based optimization on manifolds. In *Advances in Neural Information Processing Systems*, pages 9154–9164, 2019.

[7] Rakesh Chalasani, Jose C Principe, and Naveen Ramakrishnan. A fast proximal method for convolutional sparse coding. In *The 2013 International Joint Conference on Neural Networks (IJCNN)*, pages 1–5. IEEE, 2013.

[8] Bo Chang, Minmin Chen, Eldad Haber, and Ed H Chi. Antisymmetricrnn: A dynamical system view on recurrent neural networks. *arXiv preprint arXiv:1902.09689*, 2019.

[9] Zhengdao Chen, Jianyu Zhang, Martin Arjovsky, and Léon Bottou. Symplectic recurrent neural networks. *arXiv preprint arXiv:1909.13334*, 2019.

[10] 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. *arXiv preprint arXiv:1406.1078*, 2014.

[11] William Coffey and Yu P Kalmykov. *The Langevin equation: with applications to stochastic problems in physics, chemistry and electrical engineering*, volume 27. World Scientific, 2012.

[12] Simon Duane, Anthony D Kennedy, Brian J Pendleton, and Duncan Roweth. Hybrid monte carlo. *Physics Letters B*, 195(2):216–222, 1987.

[13] Jeffrey L Elman. Finding structure in time. *Cognitive Science*, 14(2):179–211, 1990.

[14] Santiago Fernández, Alex Graves, and Jürgen Schmidhuber. Sequence labelling in structured domains with hierarchical recurrent neural networks. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence, IJCAI 2007*, 2007.

[15] John S Garofolo. Timit acoustic phonetic continuous speech corpus. *Linguistic Data Consortium, 1993*, 1993.

[16] Felix A Gers and E Schmidhuber. LSTM recurrent networks learn simple context-free and context-sensitive languages. *IEEE Transactions on Neural Networks*, 12(6):1333–1340, 2001.

[17] Felix A Gers and Jürgen Schmidhuber. Recurrent nets that time and count. In *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks. IJCNN 2000. Neural Computing: New Challenges and Perspectives for the New Millennium*, volume 3, pages 189–194. IEEE, 2000.

[18] Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to forget: Continual prediction with lstm. 1999.

[19] Gabriel Goh. Why momentum really works. *Distill*, 2(4):e6, 2017.

[20] Kaiming He, Haoqi Fan, Yuxin Wu, Saining Xie, and Ross Girshick. Momentum contrast for unsupervised visual representation learning. *arXiv preprint arXiv:1911.05722*, 2019.

[21] Kyle Helfrich, Devin Willmott, and Qiang Ye. Orthogonal recurrent neural networks with scaled Cayley transform. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 1969–1978, StockholmsmÃďssan, Stockholm Sweden, 10–15 Jul 2018. PMLR.

[22] Mikael Henaff, Arthur Szlam, and Yann LeCun. Recurrent orthogonal networks and long-memory tasks. In Maria Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 2034–2042, New York, New York, USA, 20–22 Jun 2016. PMLR.

[23] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[24] Li Jing, Yichen Shen, Tena Dubcek, John Peurifoy, Scott Skirlo, Yann LeCun, Max Tegmark, and Marin Soljačić. Tunable efficient unitary neural networks (eunn) and their application to rnns. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1733–1741. JMLR. org, 2017.

[25] Anil Kag, Ziming Zhang, and Venkatesh Saligrama. RNNs evolving in equilibrium: A solution to the vanishing and exploding gradients. *arXiv preprint arXiv:1908.08574*, 2019.

[26] US Kamilov and H Mansour. Learning mmse optimal thresholds for fista. 2016.

[27] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[28] Thomas Laurent and James von Brecht. A recurrent neural network without chaos. *arXiv preprint arXiv:1612.06212*, 2016.

[29] Quoc V Le, Navdeep Jaitly, and Geoffrey E Hinton. A simple way to initialize recurrent networks of rectified linear units. *arXiv preprint arXiv:1504.00941*, 2015.

[30] Yann LeCun, Corinna Cortes, and CJ Burges. MNIST handwritten digit database. *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist*, 2, 2010.

[31] Mario Lezcano-Casado and David Martínez-Rubio. Cheap orthogonal constraints in neural networks: A simple parametrization of the orthogonal and unitary group. In *International Conference on Machine Learning (ICML)*, pages 3794–3803, 2019.

[32] Michael T McCann, Kyong Hwan Jin, and Michael Unser. Convolutional neural networks for inverse problems in imaging: A review. *IEEE Signal Processing Magazine*, 34(6):85–95, 2017.

[33] Stephen Merity, Nitish Shirish Keskar, and Richard Socher. Regularizing and optimizing LSTM language models. In *International Conference on Learning Representations*, 2018.

[34] Zakaria Mhammedi, Andrew Hellicar, Ashfaqur Rahman, and James Bailey. Efficient orthogonal parametrisation of recurrent neural networks using householder reflections. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 2401–2409. JMLR. org, 2017.

[35] Tomáš Mikolov, Martin Karafiát, Lukáš Burget, Jan Černockỳ, and Sanjeev Khudanpur. Recurrent neural network based language model. In *Eleventh Annual Conference of the International Speech Communication Association*, 2010.

[36] Thomas Moreau and Joan Bruna. Understanding the learned iterative soft thresholding algorithm with matrix factorization. *arXiv preprint arXiv:1706.01338*, 2017.

[37] Radford M Neal et al. MCMC using Hamiltonian dynamics.

[38] Daniel Neil, Michael Pfeiffer, and Shih-Chii Liu. Phased LSTM: Accelerating recurrent network training for long or event-based sequences. In *Advances in Neural Information Processing Systems*, pages 3882–3890, 2016.

[39] Arkaddii S Nemirovskii and Yu E Nesterov. Optimal methods of smooth convex minimization. *USSR Computational Mathematics and Mathematical Physics*, 25(2):21–30, 1985.

[40] Yurii E Nesterov. A method for solving the convex programming problem with convergence rate o (1/kˆ 2). In *Dokl. Akad. Nauk Sssr*, volume 269, pages 543–547, 1983.

[41] Murphy Yuezhen Niu, Lior Horesh, and Isaac Chuang. Recurrent neural networks in the eye of differential equations. *arXiv preprint arXiv:1904.12933*, 2019.

[42] Hamid Palangi, Li Deng, Yelong Shen, Jianfeng Gao, Xiaodong He, Jianshu Chen, Xinying Song, and Rabab Ward. Deep sentence embedding using long short-term memory networks: Analysis and application to information retrieval. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 24(4):694–707, 2016.

[43] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *International Conference on Machine Learning*, pages 1310–1318, 2013.

[44] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, pages 8024–8035, 2019.

[45] Boris T Polyak. Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 4(5):1–17, 1964.

[46] Andrew Pulver and Siwei Lyu. LSTM with working memory. In *2017 International Joint Conference on Neural Networks (IJCNN)*, pages 845–851. IEEE, 2017.

[47] Zhongdi Qu, Parisa Haghani, Eugene Weinstein, and Pedro Moreno. Syllable-based acoustic modeling with CTC-SMBR-LSTM. In *2017 IEEE Automatic Speech Recognition and Understanding Workshop (ASRU)*, pages 173–177. IEEE, 2017.

[48] Lamia Rahman, Nabeel Mohammed, and Abul Kalam Al Azad. A new LSTM model by introducing biological cell state. In *2016 3rd International Conference on Electrical Engineering and Information Communication Technology (ICEEICT)*, pages 1–6. IEEE, 2016.

[49] Haşim Sak, Andrew Senior, and Françoise Beaufays. Long short-term memory based recurrent neural network architectures for large vocabulary speech recognition. *arXiv preprint arXiv:1402.1128*, 2014.

[50] Salesforce. Lstm and qrnn language model toolkit for pytorch. `https://github.com/salesforce/awd-lstm-lm`, 2017.

[51] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In *International Conference on Machine Learning*, pages 1139–1147, 2013.

[52] Arthur D Szlam, Karol Gregor, and Yann L Cun. Structured sparse coding via lateral inhibition. In *Advances in Neural Information Processing Systems*, pages 1116–1124, 2011.

[53] Sachin S Talathi and Aniket Vartak. Improving performance of recurrent neural network with relu nonlinearity. *arXiv preprint arXiv:1511.03771*, 2015.

[54] T. Tieleman and G. Hinton. Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude. COURSERA: Neural Networks for Machine Learning, 2012.

[55] Eugene Vorontsov, Chiheb Trabelsi, Samuel Kadoury, and Chris Pal. On orthogonality and learning recurrent networks with long term dependencies. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 3570–3578. JMLR. org, 2017.

[56] Bao Wang, Tan M Nguyen, Andrea L Bertozzi, Richard G Baraniuk, and Stanley J Osher. Scheduled restart momentum for accelerated stochastic gradient descent. *arXiv preprint arXiv:2002.10583*, 2020.

[57] Scott Wisdom, Thomas Powers, John Hershey, Jonathan Le Roux, and Les Atlas. Full-capacity unitary recurrent neural networks. In *Advances in Neural Information Processing Systems*, pages 4880–4888, 2016.

A Experimental Details

In this section, we describe the datasets used in our experiments and provide details on the model implementation and training. MomentumLSTM, AdamLSTM, RMSPropLSTM, and SRLSTM, as well as MomentumDTRIV, AdamDTRIV, RMSPropDTRIV, and SRDTRIV share the same settings as their LSTM/DTRIV counterparts with the additional momentum μ, step size s, scheduled restart F, and the coefficient β used for computing running averages of the squared gradients. Thus, we only provide implementation and training details for the baseline LSTM and DTRIV for each task. Values for additional hyperparameters in our momentum-based models are found by grid search and reported in Table 7, 8, 9, and 10.

A.1 Pixel-by-Pixel MNIST

MNIST dataset [30] consists of 60K training images and 10K test images from 10 classes of hand-written digits. Both training and test data are binary images of size 28×28. As mentioned in Section 3.1, we flatten and process the image as a sequence of the length of 784 pixel-by-pixel. In the unpermuted task (MNIST), the images are processed row-by-row, while in the permuted task (PMNIST), a fixed permutation is applied to both training and test images.

LSTM. The baseline LSTM models consist of one LSTM cell with 128 and 256 hidden units. Orthogonal initialization is used for input-to-hidden weights, while hidden-to-hidden weights are initialized to identity matrices. The forget gate bias is initialized to 1 while all other bias scalars are initialized to 0. We follow LSTM training in [6, 31] to train LSTM models for the MNIST and PMNIST tasks. Gradient norms are clipped to 1 during training, and the smoothing constant α for the RMSProp optimizer is set to 0.9. We provide other details on hyperparameters for the LSTM training on (P)MNIST in Table 6 (top).

DTRIV. We use the best DTRIV models for each (P)MNIST task reported in [6] with Cayley initialization [21]. The gradient norms are clipped to 1 during training. Other hyperparameter details are provided in Table 6 (bottom).

A.2 TIMIT Speech Dataset

TIMIT speech dataset is a collection of real-world speech recordings [15] consisting of 3640 utterances for the training set, 192 utterances for the validation set, and 400 utterances for the test set. We follow the data preprocessing in [6, 21, 31, 57]. In particular, audio files in TIMIT are downsampled to 8kHz. A short-time Fourier transform (STFT) is then applied with a Hann window of 256 samples and a window hop of 128 samples (16 milliseconds) to yield sequences of 129 complex-valued Fourier amplitudes. The log-magnitude of these sequences is fed into the models as the input data. The task is to predict the next log-magnitude given the previous ones.

LSTM. The baseline LSTM models consist of one LSTM cell with 84, 120, and 158 hidden units. Similar to (P)MNIST experiments, orthogonal initialization is used for input-to-hidden weights, while hidden-to-hidden weights are initialized to identity matrices. However, the forget gate bias is initialized to -4 while all other bias scalars are initialized to 0. We follow LSTM training in [6, 31] to train LSTM models for the TIMIT tasks. We use the standard Adam optimizer in PyTorch [44] to train the models without using gradient clipping. We provide other details on hyperparameters for the LSTM training on TIMIT in Table 6 (top).

DTRIV. We use the best DTRIV models for each TIMIT task reported in [6] with Henaff initialization [22]. Other hyperparameter details are provided in Table 6 (bottom).

A.3 Word-Level Penn TreeBank

The Penn TreeBank (PTB) dataset is among the most popular datasets for experimenting with language modeling. The dataset has 10,000 unique words and is preprocessed to not include capital letters, numbers, or punctuation [35].

LSTM. The baseline are three-layer LSTM models with 1150 hidden units at each layer and an embedding of size 400. We follow the LSTM implementation and training in [33]. We summarize some important details in Table 6 (top).

A.4 Momentum Cells can Avoid Vanishing Gradient Issue

To confirm that MomentumRNN can alleviate vanishing gradients, we train a MomentumDTRIV and its corresponding baseline DTRIV for the PMNIST classification task. We plot $\|\partial\mathcal{L}/\partial\boldsymbol{h}_t\|_2$ for each time step t at each training iteration, as shown in Figure 2. Both MomentumDTRIV and DTRIV models used in this experiment contains one cell of 256 hidden units. The model implementation and training details are similar to those in Section A.1 above. Note that DTRIV is also an RNN with additional orthogonality constraint.

Table 6: Hyperparameters for the Baseline LSTM and DTRIV Training.

LSTM

Dataset	Optimizer	Learning Rate	Batch Size	#Epochs
MNIST	RMSProp	0.001	128	150
PMNIST	RMSProp	0.001	128	150
TIMIT	Adam	0.0001	32	700
PTB	SGD	30 (initial learning rate)	20	500

DTRIV

Dataset	Size	DTRIV Opt. Step (K)	Optimizer	Learning Rate	Orthogonal Optimizer	Orthogonal Learning Rate	Batch Size	#Epochs
MNIST	170	1		0.001		0.0001	128	150
MNIST	360	∞	RMSProp	0.0005	RMSProp	0.0001	128	150
MNIST	512	100		0.0005		0.0001	128	150
PMNIST	170	1		0.0007		0.0002	128	150
PMNIST	360	∞	RMSProp	0.0007	RMSProp	0.00005	128	150
PMNIST	512	∞		0.0003		0.00007	128	150
TIMIT	224	∞	Adam	0.001	RMSProp	0.0002	128	700
TIMIT	322	∞		0.001		0.0002	128	700

Table 7: Hyperparameters for MomentumLSTM and MomentumDTRIV Training

MomentumLSTM

Dataset	Momentum μ	Step Size s	Optimizer	Learning Rate	Batch Size	#Epochs
MNIST	0.6	0.6	RMSProp	0.001	128	150
PMNIST	0.6	1.0	RMSProp	0.001	128	150
TIMIT	0.3	0.1	Adam	0.0001	32	700
PTB	0.0	0.6	SGD	30 (initial learning rate)	20	500

MomentumDTRIV

Dataset	Size	DTRIV Opt. Step (K)	Momentum μ	Step Size s	Optimizer	Learning Rate	Orthogonal Optimizer	Orthogonal Learning Rate	Batch Size	#Epochs
PMNIST	170	1	0.6	0.9		0.0007		0.0002	128	150
PMNIST	360	∞	0.3	0.3	RMSProp	0.0007	RMSProp	0.00005	128	150
PMNIST	512	∞	0.3	0.3		0.0003		0.00007	128	150
TIMIT	224	∞	0.3	0.1	Adam	0.001	RMSProp	0.0002	128	700
TIMIT	322	∞	0.3	0.1		0.001		0.0002	128	700

Table 8: Hyperparameters for AdamLSTM and AdamDTRIV Training

AdamLSTM

Dataset	Optimizer	Momentum μ	Step Size s	β	Learning Rate	Batch Size	#Epochs
MNIST	RMSProp	0.6	0.6	0.1	0.001	128	150
PMNIST	RMSProp	0.6	1.0	0.01	0.001	128	150
TIMIT	Adam	0.3	0.1	0.999	0.0001	32	700

AdamDTRIV

Dataset	Size	DTRIV Opt. Step (K)	Momentum μ	Step Size s	β	Optimizer	Learning Rate	Orthogonal Optimizer	Orthogonal Learning Rate	Batch Size	#Epochs
PMNIST	512	∞	0.3	0.3	0.8	RMSProp	0.0003	RMSProp	0.00007	128	150

B Backpropagation Through Time – A Review

In this section, we give a short review of the backpropagation through time, which is a major algorithm for training RNNs. We consider the standard recurrent cell (1), and for any given training sample $(\boldsymbol{x}, \boldsymbol{y})$ with $\boldsymbol{x} = (\boldsymbol{x}_1, \cdots, \boldsymbol{x}_T)$ being an input sequence of length T and $\boldsymbol{y} = (y_1, \cdots, y_T)$ being the sequence of labels [3]. Let $\mathcal{L}_t$ be the loss at the time step t and the total loss on the whole sequence is

$$\mathcal{L} = \sum_{t=1}^{T} \mathcal{L}_t. \tag{16}$$

For any $1 \leq t \leq T$, we can compute the gradient of the loss $\mathcal{L}_t$ with respect to the parameter $\mathbf{U}$ as

$$\frac{\partial \mathcal{L}_t}{\partial \mathbf{U}} = \sum_{k=1}^{t} \frac{\partial \boldsymbol{h}_k}{\partial \mathbf{U}} \cdot \frac{\partial \mathcal{L}_t}{\partial \boldsymbol{h}_t} \cdot \frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{h}_k} = \sum_{k=1}^{t} \frac{\partial \boldsymbol{h}_k}{\partial \mathbf{U}} \cdot \frac{\partial \mathcal{L}_t}{\partial \boldsymbol{h}_t} \cdot \prod_{k=1}^{t-1} \frac{\partial \boldsymbol{h}_{k+1}}{\partial \boldsymbol{h}_k}, \tag{17}$$

[3] Without loss of generality, we consider the sequence to sequence modeling.

Table 9: Hyperparameters for RMSPropLSTM and RMSPropDTRIV Training

RMSPropLSTM

Dataset	Optimizer	Momentum μ	Step Size s	β	Learning Rate	Batch Size	#Epochs
MNIST	RMSProp	0.0	0.6	0.9 (size $N = 256$), 0.99 (size $N = 128$)	0.001	128	150
PMNIST	RMSProp	0.0	1.0	0.01	0.001	128	150
TIMIT	Adam	0.0	0.1	0.999	0.0001	32	700

RMSPropDTRIV

Dataset	Size	DTRIV Opt. Step (K)	Momentum μ	Step Size s	β	Optimizer	Learning Rate	Orthogonal Optimizer	Orthogonal Learning Rate	Batch Size	#Epochs
PMNIST	512	∞	0.0	0.3	0.9	RMSProp	0.0003	RMSProp	0.00007	128	150

Table 10: Hyperparameters for SRLSTM and SRDTRIV Training

SRLSTM

Dataset	Optimizer	Scheduled Restart (F)	Step Size s	Learning Rate	Batch Size	#Epochs
MNIST	RMSProp	2	1.0	0.001	128	150
PMNIST	RMSProp	40 (size $N = 256$), 6 (size $N = 128$)	0.9 (size $N = 256$), 0.01 (size $N = 128$)	0.001	128	150
TIMIT	Adam	2	0.1	0.0001	32	700
PTB	SGD	2	0.6	30 (initial learning rate)	20	500

SRDTRIV

Dataset	Size	DTRIV Opt. Step (K)	Scheduled Restart (F)	Step Size s	Optimizer	Learning Rate	Orthogonal Optimizer	Orthogonal Learning Rate	Batch Size	#Epochs
PMNIST	512	∞	2	0.3	RMSProp	0.0003	RMSProp	0.00007	128	150

where $\frac{\partial \boldsymbol{h}_{k+1}}{\partial \boldsymbol{h}_k} = \mathbf{D}_k \mathbf{U}^{\mathrm{T}}$ with $\mathbf{D}_k = \mathrm{diag}(\sigma'(\mathbf{U}\boldsymbol{h}_k + \mathbf{W}\boldsymbol{x}_{k+1} + \boldsymbol{b}))$. Similarly, we can compute $\partial \mathcal{L}_t / \partial \mathbf{W}$ and $\partial \mathcal{L}_t / \partial \boldsymbol{b}$.

C More Experimental Results

We conduct more comprehensive experiments for the Adam principled and NAG principled RNNs. In particular, we perform (P)MNIST and TIMIT experiments using the AdamLSTM, RMSPropLSTM, and SRLSTM of 128 and 120 hidden units, respectively. For (P)MNIST task, RMSPropLSTM achieves the best test accuracy and converges the fastest. For the TIMIT task, MomentumLSTM and SRLSTM outperform the other models while converging faster. We summarize our results in Table 11 and 12, as well as in Figure 7. Note that in the main text, we conduct the same experiments using the same models but with different numbers of hidden units (i.e. 256 hidden units for the (P)MNIST task and 158 hidden units for the TIMIT task).

In addition, we apply our Adam and NAG principled designing methods on a DTRIV, an orthogonal RNN [6], for the PMNIST classification task. We observe that AdamDTRIV, RMSPropDTRIV, and SRDTRIV outperform the baseline DTRIV while converging faster. SRDTRIV also outperforms MomentumDTRIV. We summarize our results in Table 13 and Figure 8. Hyperparameter values for this experiment can be found in Table 8, 9, and 10 (bottom).

Finally, we apply SRLSTM for the Penn TreeBank language modeling at the word level. Our experiment suggests that SRLSTM outperforms the baseline LSTM while yields slightly worse results than MomentumL-

STM in terms of test accuracy. SRLSTM also converges the fastest. We summarize our results in Table 14 and Figure 9. Hyperparameter values for this experiment can be found in Table 10 (top).

Table 11: Best test accuracy at the MNIST and PMNIST tasks (%). We use the baseline results reported in [21], [57], [55]. All of our proposed models outperform the baseline LSTM. Among the models using $N = 128$ hidden units, RMSPropLSTM yields the best results in both tasks.

Model	n	# params	MNIST	PMNIST
LSTM	128	$\approx 68K$	98.70[21],97.30 [55]	92.00 [21],92.62 [55]
MomentumLSTM	128	$\approx 68K$	$\mathbf{99.04 \pm 0.04}$	$\mathbf{93.40 \pm 0.25}$
AdamLSTM	128	$\approx 68K$	98.98 ± 0.08	93.75 ± 0.25
RMSPropLSTM	128	$\approx 68K$	$\mathbf{99.09 \pm 0.05}$	$\mathbf{94.32 \pm 0.43}$
SRLSTM	128	$\approx 68K$	98.89 ± 0.08	93.65 ± 0.56

Table 12: Test and validation MSEs at the end of the epoch with the lowest validation MSE for the TIMIT task. All of our proposed models outperform the baseline LSTM. Among models using $N = 120$ hidden units, MomentumLSTM performs the best.

Model	n	# params	Val. MSE	Test MSE
LSTM	120	$\approx 135K$	11.77 ± 0.14 (13.93 [21, 31])	11.83 ± 0.12 (12.95 [21, 31])
MomentumLSTM	120	$\approx 135K$	$\mathbf{8.00 \pm 0.30}$	$\mathbf{8.04 \pm 0.30}$
AdamLSTM	120	$\approx 135K$	10.91 ± 0.08	10.96 ± 0.08
RMSPropLSTM	120	$\approx 135K$	11.83 ± 0.20	11.90 ± 0.19
SRLSTM	120	$\approx 135K$	8.15 ± 0.26	8.21 ± 0.26

Table 13: Best test accuracy on the PMNIST tasks (%) for MomentumDTRIV and the baseline DTRIV, as well as for AdamDTRIV, RMSPropDTRIV, and SRDTRIV. We provide both our reproduced baseline results and those reported in [6]. All of our momentum-based models outperform the baseline DTRIV. When using $N = 512$ hidden units, SRDTRIV yields the best result.

Model	n	# params	PMNIST
DTRIV	170	$\approx 16K$	95.21 ± 0.10 (95.20 [6])
DTRIV	360	$\approx 69K$	96.45 ± 0.10 (96.50 [6])
DTRIV	512	$\approx 137K$	96.62 ± 0.12 (96.80 [6])
MomentumDTRIV	170	$\approx 16K$	$\mathbf{95.37 \pm 0.09}$
MomentumDTRIV	360	$\approx 69K$	$\mathbf{96.73 \pm 0.08}$
MomentumDTRIV	512	$\approx 137K$	$\mathbf{96.89 \pm 0.08}$
AdamDTRIV	512	$\approx 137K$	$\mathbf{96.77 \pm 0.21}$
RMSPropDTRIV	512	$\approx 137K$	$\mathbf{96.75 \pm 0.12}$
SRDTRIV	512	$\approx 137K$	$\mathbf{97.02 \pm 0.09}$

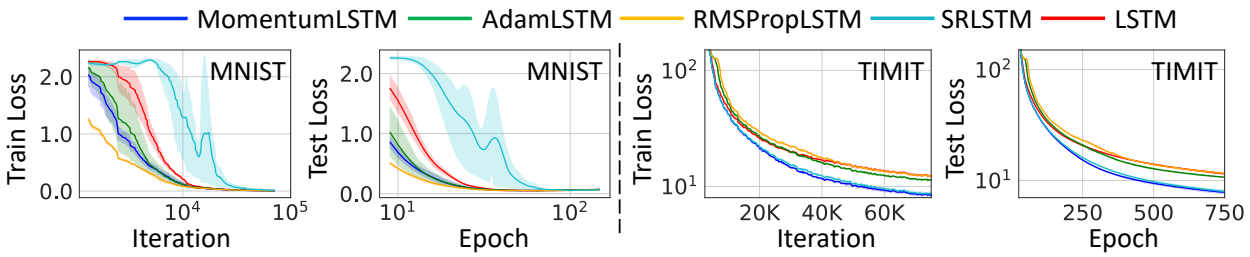

Figure 7: Train and test loss of MomentumLSTM (blue), AdamLSTM (green), RMSPropLSTM (orange), SRLSTM (cyan), and LSTM (red) using $N = 128$ hidden units for MNIST (left two panels) and using $N = 120$ hidden units for TIMIT (right two panels) tasks. MomentumLSTM converges faster than LSTM in both tasks. RMSPropLSTM and MomentumLSTM/SRLSTM converge the fastest for MNIST and TIMIT tasks, respectively.

Table 14: Model test perplexity at the end of the epoch with the lowest validation perplexity for the Penn TreeBank language modeling task (word level). Both MomentumLSTM and SRLSTM outperform the baseline LSTM.

Model	# params	Val. PPL	Test PPL
LSTM	$\approx 24M$	61.96 ± 0.83	59.71 ± 0.99 (58.80 [33])
MomentumLSTM	$\approx 24M$	$\mathbf{60.71 \pm 0.24}$	$\mathbf{58.62 \pm 0.22}$
SRLSTM	$\approx 24M$	61.12 ± 0.68	58.83 ± 0.62

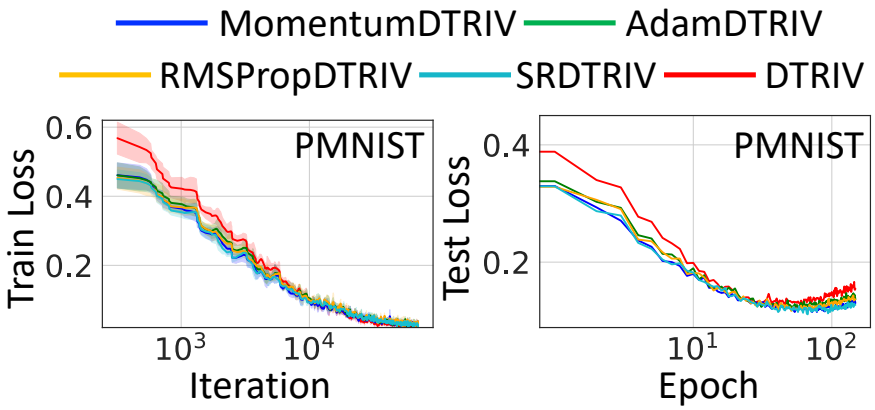

Figure 8: Train and test loss of MomentumDTRIV (blue), AdamDTRIV (green), RMSPropDTRIV (orange), SRDTRIV (cyan), and DTRIV (red) for PMNIST task. Our momentum-based models converge faster than the baseline DTRIV.

D Computational Time and Memory Cost: RNN vs. Momentum-RNN

We provide the computation time and memory cost per sample at training and evaluation of MomentumLSTM, AdamLSTM, RMSPropLSTM, and SRLSTM in comparison with LSTM for PMNIST classification task using 256 hidden units in Table 15 and 16, respectively.

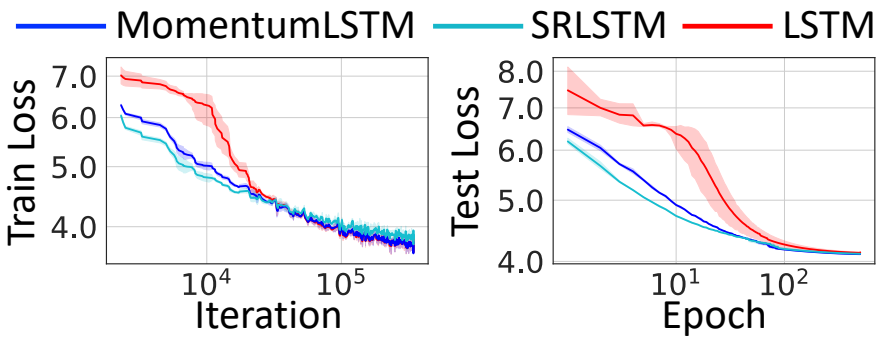

Figure 9: Train (left) and test loss (right) of MomentumLSTM (blue), SRLSTM (cyan), and LSTM (red) for the Penn Treebank language modeling tasks at word level. Both MomentumLSTM and SRLSTM converge faster than the baseline LSTM. SRLSTM converges the fastest.

Table 15: Computation time per sample at training and evaluation for PMNIST classification task using models with 256 hidden units.

Model	Training Time (μs/sample)	Evaluation Time (μs/sample)
LSTM	6.18	2.52
MomentumLSTM	7.43	3.16
AdamLSTM	10.34	4.07
RMSPropLSTM	9.94	3.96
SRLSTM	8.34	3.16

Table 16: Memory cost per sample at training and evaluation for PMNIST classification task using models with 256 hidden units.

Model	Training Memory (MB/sample)	Evaluation Memory (MB/sample)
LSTM	15.93	7.51
MomentumLSTM	15.95	7.51
AdamLSTM	25.13	7.52
RMSPropLSTM	25.13	7.52
SRLSTM	15.95	7.51

E Additional Information about the Figures in the Main Text

In Figure 3, the MNIST plots are for models with 256 hidden units, and the TIMIT plots are for models with 158 hidden units. In Figure 5, the PMNIST plots are for models with 512 hidden units, and the TIMIT plots are for models with 322 hidden units.

F MomentumLSTM Cell Implementation in Pytorch

```
import torch
import torch.nn as nn
from torch.nn import functional as F

class MomentumLSTMCell(nn.Module):

    """
    An implementation of MomentumLSTM Cell

    Args:
        input_size: The number of expected features in the input 'x'
        hidden_size: The number of features in the hidden state 'h'
        mu: momentum coefficient in MomentumLSTM Cell
        s: step size in MomentumLSTM Cell
        bias: If ''False'', then the layer does not use bias weights 'b_ih' and 'b_hh'.
            Default: ''True''

    Inputs: input, hidden0=(h_0, c_0), v0
        - input of shape '(batch, input_size)': tensor containing input features
        - h_0 of shape '(batch, hidden_size)': tensor containing the initial hidden state
            for each element in the batch.
        - c_0 of shape '(batch, hidden_size)': tensor containing the initial cell state for
            each element in the batch.
        - v0 of shape '(batch, hidden_size)': tensor containing the initial momentum state
            for each element in the batch

    Outputs: h1, (h_1, c_1), v1
        - h_1 of shape '(batch, hidden_size)': tensor containing the next hidden state for
            each element in the batch
        - c_1 of shape '(batch, hidden_size)': tensor containing the next cell state for
            each element in the batch
        - v_1 of shape '(batch, hidden_size)': tensor containing the next momentum state for
             each element in the batch
    """

    def __init__(self, input_size, hidden_size, mu, s, bias=True):
        super(MomentumLSTMCell, self).__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.bias = bias
        self.x2h = nn.Linear(input_size, 4 * hidden_size, bias=bias)
        self.h2h = nn.Linear(hidden_size, 4 * hidden_size, bias=bias)

        # for momentumnet
        self.mu = mu
        self.s = s

        self.reset_parameters(hidden_size)

    def reset_parameters(self, hidden_size):
        nn.init.orthogonal_(self.x2h.weight)
        nn.init.eye_(self.h2h.weight)
        nn.init.zeros_(self.x2h.bias)
        self.x2h.bias.data[hidden_size:(2 * hidden_size)].fill_(1.0)
        nn.init.zeros_(self.h2h.bias)
        self.h2h.bias.data[hidden_size:(2 * hidden_size)].fill_(1.0)

    def forward(self, x, hidden, v):

        hx, cx = hidden

        x = x.view(-1, x.size(1))
        v = v.view(-1, v.size(1))

        vy = self.mu * v + self.s * self.x2h(x)
```

```
gates = vy + self.h2h(hx)

gates = gates.squeeze()

ingate, forgetgate, cellgate, outgate = gates.chunk(4, 1)

ingate = F.sigmoid(ingate)
forgetgate = F.sigmoid(forgetgate)
cellgate = F.tanh(cellgate)
outgate = F.sigmoid(outgate)

cy = torch.mul(cx, forgetgate) +  torch.mul(ingate, cellgate)

hy = torch.mul(outgate, F.tanh(cy))

return hy, (hy, cy), vy
```