RICE UNIVERSITY

On the Momentum-based Methods for Training and Designing Deep Neural Networks

By

Tan Minh Nguyen

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Doctor of Philosophy

APPROVED, THESIS COMMITTEE

Richard G. Baraniuk
Victor E. Cameron Professor of Electrical and Computer Engineering

Santiago Segarra
W. M. Rice Trustee Assistant Professor of Electrical and Computer Engineering

Maarten de Hoop
Simons Chair in Computational and Applied Mathematics and Earth Science

Anima Anandkumar
Bren Professor of Computing + Mathematical Sciences

Stan J. Osher
Professor of Mathematics & Computer Science, Electrical Engineering & Chemical and Biomolecular Engineering

HOUSTON, TEXAS

September 2020
On the Momentum-based Methods for Training and Designing Deep Neural Networks

Tan M. Nguyen
ABSTRACT

On the Momentum-based Methods for Training and Designing Deep Neural Networks

by

Tan M. Nguyen

Training and designing deep neural networks (DNNs) are an art that often involves expensive search over candidate architectures and optimization algorithms. In my thesis, we develop novel momentum-based methods to speed up deep neural networks training and facilitate the process of designing them.*

For training DNNs, stochastic gradient descent (SGD) algorithms with constant momentum and its variants such as Adam are the optimization methods of choice for training DNNs. There is great interest in speeding up the convergence of these methods due to their high computational expense. Nesterov accelerated gradient (NAG) improves the convergence rate of gradient descent (GD) for convex optimization using a specially designed momentum; however, it accumulates error when an inexact gradient is used (such as in SGD), slowing convergence at best and diverging at worst. We propose scheduled restart SGD (SRSGD), a new NAG-style scheme for training

*My thesis work has resulted in the following papers:


DNNs. SRSGD replaces the constant momentum in SGD by the increasing momentum in NAG but stabilizes the iterations by resetting the momentum to zero according to a schedule. Using a variety of models and benchmarks for image classification, we demonstrate that, in training DNNs, SRSGD significantly improves convergence and generalization; for instance, in training ResNet-200 for ImageNet classification, SRSGD achieves an error rate of 20.93% vs. the benchmark of 22.13%. These improvements become more significant as the network grows deeper. Furthermore, on both CIFAR and ImageNet, SRSGD reaches similar or even better error rates with significantly fewer training epochs compared to the SGD baseline.

For designing DNNs, we focus on the recurrent neural networks (RNNs) and 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 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 NAG with a restart, can be easily incorporated into the MomentumRNN framework for designing new recurrent cells with even better performance.
Acknowledgments

First of all, I am very grateful to all members of the DSP group at Rice University for their great support during my Ph.D. Especially, I would like to thank my Ph.D. advisor, Dr. Richard G. Baraniuk, for being not only a great mentor but also a great friend who has always been supportive throughout these 2 + 6 amazing years (undergraduate research + Master/Ph.D.).

Furthermore, I would like to thank my collaborators in Professor Stanley J. Osher and Professor Andrea L. Bertozzi’s groups at UCLA, for providing a highly stimulating research environment to start our collaborative projects that lead to this thesis.

In addition, I would like to express my sincere thanks to Professor Anima Anandkumar at Caltech for her great help, support, and advice during my Ph.D. It was an absolute pleasure working with Anima, and I am so grateful for her time, effort and positive energy!

My friends at the NVIDIA Machine Learning Research team, the Amazon AI team, the Caltech TensorLab, Professor Maarten V. de Hoop’s research group, and the ECE department at Rice University have helped me by providing comments and suggestions on my work, as well as collaborating with me on various great projects. I wish to thank them all!

I would also like to thank Nguyen Ngoc Thanh Van and Adithya Pediredla for being such great friends over the years and for all their encouragement!

Finally, I would like to thank my parents for their unconditional support!

Funding Acknowledgements: My research has been generously funded by the NSF Graduate Research Fellowship, the Neuroengineering IGERT: From Cells to Systems Fellowship, the Texas Instruments Fellowship, and the Ken Kennedy Institute for
Information Technology Oil & Gas HPC Workshop Fellowship.
Contents

Abstract ii
Acknowledgments iv
List of Illustrations x
List of Tables xv

1 Introduction 1

2 Scheduled Restart Momentum for Accelerated Stochastic Gradient Descent 8

2.1 Review: Momentum in Gradient Descent .......................... 8
2.2 Scheduled Restart SGD (SRSGD) ................................ 10
  2.2.1 Uncontrolled Bound of Nesterov Accelerated SGD (NASGD) . 11
  2.2.2 SRSGD and Its Convergence ................................. 12
2.3 Experimental Results .................................................. 12
  2.3.1 CIFAR10 and CIFAR100 ..................................... 13
  2.3.2 ImageNet ......................................................... 16
  2.3.3 Empirical Analysis ............................................ 18
2.4 Related Work ......................................................... 20

3 MomentumRNN: Integrating Momentum into Recurrent Neural Networks 21

3.1 Review: Recurrent Neural Networks and Long Short Term Memory Networks ........................................... 21
3.2 Momentum Recurrent Neural Networks .................. 22
  3.2.1 Gradient Descent Analogy for RNN and MomentumRNN .. 22
  3.2.2 Analysis of the Vanishing Gradient Issue: Momentum Cell vs. Recurrent Cell .................. 24
  3.2.3 Beyond MomentumRNN: NAG and Adam Principled Recurrent Neural Nets .................. 25
3.3 Experimental Results ......................................... 27
  3.3.1 Pixel-by-Pixel MNIST .................................. 28
  3.3.2 TIMIT Speech Dataset .................................. 29
  3.3.3 Word-Level Penn TreeBank ............................ 31
  3.3.4 NAG and Adam Principled Recurrent Neural Nets .......... 32
3.4 Additional Results and Analysis ............................ 32
3.5 Related Work .................................................. 35

4 Conclusion and Broader Impact ............................. 36

A Additional Details and Results for Scheduled Restart

Stochastic Gradient Descent .................. 38
  A.1 Uncontrolled Bound of NASGD ............................ 38
    A.1.1 Preliminaries ................................. 39
    A.1.2 Uncontrolled Bound of NASGD .......................... 40
  A.2 NAG with δ-Inexact Oracle & Experimental Settings in Section 3.1 .... 45
  A.3 Convergence of SRSGD ................................. 47
  A.4 Datasets and Implementation Details .................. 50
    A.4.1 CIFAR .................................. 50
    A.4.2 ImageNet .................................. 51
    A.4.3 Training ImageNet in Fewer Number of Epochs: ........ 51
A.5 SRSGD vs. SGD and SGD + NM on ImageNet Classification and Other Tasks ........................................ 52
A.5.1 Comparing with SGD with Nesterov Momentum on ImageNet Classification .................................. 52
A.5.2 Long Short-Term Memory (LSTM) Training for Pixel-by-Pixel MNIST ......................................... 53
A.5.3 Wasserstein Generative Adversarial Networks (WGAN) Training on MNIST .................................. 54
A.6 Error Rate vs. Reduction in Training Epochs ........................................ 58
A.6.1 Implementation Details ........................................ 58
A.6.2 Additional Experimental Results ........................................ 59
A.7 Impact of Restarting Frequency for ImageNet and CIFAR100 ........................................ 61
A.7.1 Implementation Details ........................................ 61
A.7.2 Additional Experimental Results ........................................ 62
A.8 Full Training with Less Epochs at the Intermediate Learning Rates ........................................ 62
A.9 Visualization of SRSGD’s trajectory ........................................ 63
A.10 SRSGD Implementation in Pytorch ........................................ 68
A.11 SRSGD Implementation in Keras ........................................ 71

B Additional Details and Results for Momentum Recurrent Neural Networks ........................................ 75
B.1 Experimental Details ........................................ 75
B.1.1 Pixel-by-Pixel MNIST ........................................ 75
B.1.2 TIMIT Speech Dataset ........................................ 76
B.1.3 Word-Level Penn TreeBank ........................................ 77
B.1.4 Momentum Cells can Avoid Vanishing Gradient Issue ........................................ 77
B.2 Backpropagation Through Time – A Review ........................................ 78
B.3 More Experimental Results ........................................ 81
B.4 Computational Time and Memory Cost: RNN vs. MomentumRNN . 84
B.5 MomentumLSTM Cell Implementation in Pytorch . . . . . . . . . . 86

Bibliography 90
1.1 Error vs. depth of ResNet models trained with SRSGD and the baseline SGD with constant momentum. Advantage of SRSGD continues to grow with depth. ............................................. 5

1.2 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. 6

2.1 Comparison between different schemes in optimizing the quadratic function in (2.5) with (a) exact gradient, (b) gradient with constant variance Gaussian noise, and (c) gradient with decaying variance Gaussian noise. NAG, ARNAG, and SRNAG can speed up convergence remarkably when exact gradient is used. Also, SRNAG is more robust to noisy gradient than NAG and ARNAG. ................................................................. 10

2.2 (a) Training loss comparison between different schemes in training logistic regression for MNIST classification. NASGD is not robust to noisy gradient, ARSGD almost degenerates to SGD, and SRSGD performs the best in this case. (b, c) Training loss vs. training epoch of ResNet models trained with SRSGD (blue) and the SGD baseline with constant momentum (red). . . . 13
2.3 Test error vs. number of training epochs. Dashed lines are test errors of SGD trained in 200 epochs for CIFAR10 and 90 epochs for ImageNet. For CIFAR, SRSGD with fewer epochs achieves comparable results to SRSGD with 200 epochs. For ImageNet, training with less epochs slightly decreases the performance of SRSGD but still achieves comparable results to 200-epoch SGD.

2.4 Training loss (left) and test error (right) of Pre-ResNet-290 trained on CIFAR10 with different initial restarting frequencies $F_1$ (linear schedule). SRSGD with small $F_1$ approximates SGD without momentum, while SRSGD with large $F_1$ approximates NASGD.

3.1 $\ell_2$ norm of the gradients of the loss $\mathcal{L}$ w.r.t. the state vector $h_t$ at each time step $t$ for RNN (left) and MomentumRNN (right). MomentumRNN does not suffer from vanishing gradients.

3.2 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.

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

3.4 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 overfitting while MomentumDTRIV overfits less.
3.5 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.

A.1 Training loss vs. training iterations of LSTM trained with SGD (red), SGD + NM (green), and SRSGD (blue) for PMNIST classification tasks.

A.2 Earth Moving distance estimate (i.e. discriminator loss) vs. training epochs of WGAN with gradient penalty trained with SGD (red), SGD + NM (green), and SRSGD (blue) on MNIST.

A.3 MNIST digits generated by WGAN trained with gradient penalty by SGD (left), SGD + NM (middle), and SRSGD (right).

A.4 Test error vs. number of training epochs. Dashed lines are test errors of SGD trained in 200 epochs for CIFAR10 and 90 epochs for ImageNet. For CIFAR, SRSGD with fewer epochs achieves comparable results to SRSGD with 200 epochs. For ImageNet, training with less epochs slightly decreases the performance of SRSGD but still achieves comparable results to 200-epoch SGD.

A.5 Test error vs. number of epoch reduction in CIFAR100 training. The dashed lines are test errors of the SGD baseline. For CIFAR100, SRSGD training with fewer epochs can achieve comparable results to SRSGD training with full 200 epochs. In some cases, such as with Pre-ResNet-290 and 1001, SRSGD training with fewer epochs achieves even better results than SRSGD training with full 200 epochs.

A.6 Training loss (left) and test error (right) of Pre-ResNet-290 trained on CIFAR10 with different initial restarting frequencies $F_1$ (linear schedule). SRSGD with small $F_1$ approximates SGD without momentum, while SRSGD with large $F_1$ approximates NASGD.
A.7 Training loss and test error of Pre-ResNet-290 trained on CIFAR100 with different initial restarting frequencies $F_1$ (linear schedule). SRSGD with small $F_1$ approximates SGD without momentum, while SRSGD with large $F_1$ approximates NASGD. .................................................. 63

A.8 Training loss and test error of ResNet-101 trained on ImageNet with different initial restarting frequencies $F_1$. We use linear schedule and linearly decrease the restarting frequency to 1 at the last learning rate. SRSGD with small $F_1$ approximates SGD without momentum, while SRSGD with large $F_1$ approximates NASGD. .................................................. 64

A.9 Test error when using new learning rate schedules with less training epochs at the 2nd and 3rd learning rate for CIFAR10. We still train in full 200 epochs in this experiment. On the x-axis, 10, for example, means we reduce the number of training epochs by 10 at each intermediate learning rate, i.e. the 2nd and 3rd learning rate. The dashed lines are test errors of the SGD baseline. .................................................. 64

A.10 Test error when using new learning rate schedules with less training epochs at the 2nd and 3rd learning rate for CIFAR100. We still train in full 200 epochs in this experiment. On the x-axis, 10, for example, means we reduce the number of training epochs by 10 at each intermediate learning rate, i.e. the 2nd and 3rd learning rate. The dashed lines are test errors of the SGD baseline. .................................................. 66

A.11 Test error when using new learning rate schedules with less training epochs at the 2nd learning rate for ImageNet. We still train in full 90 epochs in this experiment. On the x-axis, 10, for example, means we reduce the number of training epochs by 10 at the 2nd learning rate. The dashed lines are test errors of the SGD baseline. .................................................. 67
A.12 Trajectory through bad minima of SGD, SGD with constant momentum, and SRSGD during the training: we train a neural net classifier and plot the iterates of SGD after each ten epoch (red dots). We also plot locations of nearby “bad” minima with poor generalization (blue dots). We visualize these using PCA and t-SNE embedding. The blue color bar is for the test accuracy of bad local minima while the red color bar is for the number of training epochs. All blue dots for SGD with constant momentum and SRSGD achieve near perfect train accuracy, but with test accuracy below 59%. All blue dots for SGD achieves average train accuracy of 73.11% and with test accuracy also below 59%. The final iterate (yellow star) of SGD, SGD with constant momentum, and SRSGD achieve 73.13%, 99.25%, and 100.0% test accuracy, respectively.

B.1 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.

B.2 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.

B.3 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.
2.1 Classification test error (%) on CIFAR10 using the SGD, SGD + NM, and SRSGD. We report the results of SRSGD with two different restarting schedules: linear (lin) and exponential (exp). The numbers of iterations after which we restart the momentum in the lin schedule are 30, 60, 90, 120 for the 1st, 2nd, 3rd, and 4th stage. Those numbers for the exp schedule are 40, 50, 63, 78. We also include the reported results from [1] (in parentheses) in addition to our reproduced results.

2.2 Classification test error (%) on CIFAR100 using the SGD, SGD + NM, and SRSGD. We report the results of SRSGD with two different restarting schedules: linear (lin) and exponential (exp). The numbers of iterations after which we restart the momentum in the lin schedule are 50, 100, 150, 200 for the 1st, 2nd, 3rd, and 4th stage. Those numbers for the exp schedule are 45, 68, 101, 152. We also include the reported results from [1] (in parentheses) in addition to our reproduced results.

2.3 Comparison of classification errors on CIFAR10/100 (%) between SRSGD training with only 100 epochs and SGD baseline training with 200 epochs. Using only half the number of training epochs, SRSGD achieves comparable results to SGD.
2.4 Single crop validation errors (%) on ImageNet of ResNets trained with SGD baseline and SRSGD. We report the results of SRSGD with the increasing restarting frequency in the first two learning rates. In the last learning rate, the restarting frequency is linearly decreased to 1. For baseline results, we also include the reported single-crop validation errors [2] (in parentheses). 17

2.5 Comparison of single crop validation errors on ImageNet (%) between SRSGD training with fewer epochs and SGD training with full 90 epochs. 18

3.1 Best test accuracy at the MNIST and PMNIST tasks (%). We use the baseline results reported in [3], [4], [5]. 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. 29

3.2 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. 31

3.3 Model test perplexity at the end of the epoch with the lowest validation perplexity for the Penn Treebank language modeling task (word level). 32

3.4 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. 33

3.5 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. 34

A.1 Restarting frequencies for CIFAR10 and CIFAR100 experiments. 50

A.2 Restarting frequencies for ImageNet experiments. 51
A.3 Learning rate and restarting frequency schedule for ImageNet short training, i.e. Table 2.5 in the main text.

A.4 Single crop validation errors (%) on ImageNet of ResNets trained with SGD + NM and SRSGD. We report the results of SRSGD with the increasing restarting frequency in the first two learning rates. In the last learning rate, the restarting frequency is linearly decreased from 70 to 1. For baseline results, we also include the reported single-crop validation errors [2] (in parentheses).

A.5 Test errors (%) on Permuted MNIST of trained with SGD, SGD + NM and SRSGD. The LSTM model has 128 hidden units. In all experiments, we use the initial learning rate of 0.01, which is reduced by a factor of 10 at epoch 200 and 300. All models are trained for 350 epochs. The momentum for SGD and SGD + NM is set to 0.9. The restart schedule in SRSGD is set to 90, 30, and 90.

A.6 Discriminator loss (i.e. Earth Moving distance estimate) of the WGAN with gradient penalty trained on MNIST with SGD, SGD + NM and SRSGD. In all experiments, we use the initial learning rate of 0.01, which is reduced by a factor of 10 at epoch 200 and 300. All models are trained for 350 epochs. The momentum for SGD and SGD + NM is set to 0.9. The restart schedule in SRSGD is set to 60, 120, and 180.

A.7 Learning rate (LR) schedule for the ablation study of error rate vs. reduction in training epochs for CIFAR10 experiments, i.e. Figure A.5 (left) in the main text and for CIFAR100 experiments, i.e. Figure A.5 in this Appendix.

A.8 Test error vs. number of training epochs for CIFAR10.

A.9 Top 1 single crop validation error vs. number of training epochs for ImageNet.

A.10 Test error vs. number of training epochs for CIFAR100.
A.11 Test error when using new learning rate schedules with less training epochs at the 2nd and 3rd learning rate for CIFAR10. We still train in full 200 epochs in this experiment. In the table, 80-90-100, for example, means we reduce the learning rate by factor of 10 at the 80th, 90th, and 100th epoch.

A.12 Test error when using new learning rate schedules with less training epochs at the 2nd and 3rd learning rate for CIFAR100. We still train in full 200 epochs in this experiment. In the table, 80-90-100, for example, means we reduce the learning rate by factor of 10 at the 80th, 90th, and 100th epoch.

A.13 Top 1 single crop validation error when using new learning rate schedules with less training epochs at the 2nd learning rate for ImageNet. We still train in full 90 epochs in this experiment. In the table, 30-40, for example, means we reduce the learning rate by factor of 10 at the 30th and 40th epoch.

B.1 Hyperparameters for the Baseline LSTM and DTRIV Training.

B.2 Hyperparameters for MomentumLSTM and MomentumDTRIV Training

B.3 Hyperparameters for AdamLSTM and AdamDTRIV Training

B.4 Hyperparameters for RMSPropLSTM and RMSPropDTRIV Training

B.5 Hyperparameters for SRLSTM and SRDTRIV Training

B.6 Best test accuracy at the MNIST and PMNIST tasks (%). We use the baseline results reported in [3], [4], [5]. 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.

B.7 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.
B.8 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. .......................... 83

B.9 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. .......................... 84

B.10 Computation time per sample at training and evaluation for PMNIST
classification task using models with 256 hidden units. .......................... 85

B.11 Memory cost per sample at training and evaluation for PMNIST
classification task using models with 256 hidden units. .......................... 86
Chapter 1

Introduction

Training many machine learning (ML) models reduces to solving the following optimization problem

\[
\min_w f(w) := \min_w \frac{1}{N} \sum_{i=1}^{N} f_i(w), \quad w \in \mathbb{R}^d, \tag{1.1}
\]

where \( f_i(w) := \mathcal{L}(g(x_i, w), y_i) \) is the loss between the ground-truth label \( y_i \) and the prediction by the model \( g(\cdot, w) \), parametrized by \( w \). This training loss is typically a cross-entropy loss for classification and a root mean squared error for regression. Here, \( \{x_i, y_i\}_{i=1}^{N} \) are the training samples, and problem (1.1) is known as empirical risk minimization (ERM). For many practical applications, \( f(w) \) is highly non-convex, and \( g(\cdot, w) \) is chosen among deep neural networks (DNNs) due to their preeminent performance across various tasks. These deep models are heavily overparametrized and require large amounts of training data. Thus, both \( N \) and the dimension of \( w \) can scale up to millions or even billions. These complications pose serious computational challenges.

One of the simplest algorithms to solve (1.1) is gradient descent (GD), which updates \( w \) according to:

\[
w^{k+1} = w^k - s_k \frac{1}{N} \sum_{i=1}^{N} \nabla f_i(w^k), \tag{1.2}
\]

where \( s_k > 0 \) is the step size at the \( k \)-th iteration. Computing \( \nabla f(w^k) \) on the entire training set is memory intensive and often prohibitive for devices with limited random access memory (RAM) such as graphics processing units (GPUs) used for deep learning...
(DL). In practice, we sample a subset of the training set, of size \( m \) with \( m \ll N \), to approximate \( \nabla f(w^k) \) by the mini-batch gradient \( 1/m \sum_{j=1}^{m} \nabla f_{ij}(w^k) \). This results in the stochastic gradient descent (SGD) update

\[
w^{k+1} = w^k - s_k \frac{1}{m} \sum_{j=1}^{m} \nabla f_{ij}(w^k).
\]  

(1.3)

SGD and its accelerated variants are among the most used optimization algorithms in ML practice [7]. These gradient-based algorithms have a number of benefits. Their convergence rate can be independent of the dimension of the underlying problem [7]; their computational complexity is low and easy to parallelize, making them suitable for large scale and high dimensional problems [8, 9]. They have achieved, so far, the best performance for training DNNs [10].

Nevertheless, GD and SGD have convergence issues, especially when the problem is ill-conditioned. There are two common approaches to accelerate GD: adaptive step size [11, 12, 13] and momentum [14]. The integration of both adaptive step size and momentum with SGD leads to Adam [15], which is one of the most used optimizers for training DNNs. Many recent developments have improved Adam [16, 17, 18, 19].

GD with constant momentum leverages the previous step to accelerate GD according to:

\[
v^{k+1} = w^k - s_k \nabla f(w^k); \quad w^{k+1} = v^{k+1} + \mu(v^{k+1} - v^k),
\]  

(1.4)

where \( \mu > 0 \) is a constant. A similar acceleration can be achieved by the heavy-ball (HB) method [14]. The momentum update in both (1.4) and HB have the same convergence rate of \( O(1/k) \) as that of GD for convex smooth optimization. A breakthrough due to Nesterov [20] replaces the constant momentum \( \mu \) with \( (k - 1)/(k + 2) \) (aka, Nesterov accelerated gradient (NAG) momentum), and it can accelerate the convergence rate to \( O(1/k^2) \), which is optimal for convex and smooth loss functions [20, 21]. NAG can
also speed up the process of escaping saddle points [22]. In practice, NAG momentum and its variants such as Katyusha momentum [23] can accelerate GD for nonconvex optimization, especially when the underlying loss function is poorly conditioned [24]. However, NAG accumulates error when the gradient is inexact [25]. Until now, only constant momentum has been successfully used to train DNNs in practice [26]. Since NAG momentum has achieved a much better convergence rate than constant momentum methods with an exact gradient oracle, we study the following question:

Q1. Can we leverage NAG momentum to accelerate SGD in training DNNs and improve the final test accuracy of the trained models?

Furthermore, 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 [27] which has achieved state-of-the-art results in many applications, including speech recognition [28], acoustic modeling [29, 30], and language modeling [31]. There have been many efforts in improving LSTM: [32] introduces a forget gate into the original LSTM cell, which can forget information selectively; [33] further adds peephole connections to the LSTM cell to inspect its current internal states[34]; to reduce the computational cost, a gated recurrent unit (GRU) [35] uses a single update gate to replace the forget and input gates in LSTM. Phased LSTM [36] adds a new time gate to the LSTM cell and achieves faster convergence than the regular LSTM on learning long sequences. In addition, [37] and [38] 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. In this thesis, we also explore the following question:

Q2. Can we leverage momentum methods to facilitate the design of DNNs?
Contributions. We answer Q1 by proposing the first algorithm that integrates scheduled restart (SR) NAG momentum with plain SGD. Here, we restart the momentum, which is orthogonal to the learning rate restart [39]. We name the resulting algorithm scheduled restart SGD (SRSGD). Theoretically, we prove the error accumulation of Nesterov accelerated SGD (NASGD) and the convergence of SRSGD. The major practical benefits of SRSGD are fourfold:

- SRSGD significantly speeds up DNN training. For image classification, SRSGD significantly reduces the number of training epochs while preserving or even improving the network’s accuracy. In particular, on CIFAR10/100, the number of training epochs is reduced by half with SRSGD while on ImageNet the reduction in training epochs is also remarkable.

- DNNs trained by SRSGD generalize significantly better than the current benchmark optimizers. The improvement becomes more significant as the network grows deeper as shown in Fig. 1.1.

- SRSGD reduces overfitting in training very deep networks such as ResNet-200 for ImageNet classification, enabling the accuracy to keep increasing with depth.

- SRSGD is straightforward to implement and only requires changes in a few lines of the SGD code. There is also no additional computational or memory overhead.

We focus on image classification with DNNs, in which SGD with constant momentum is the choice.

We answer Q2 by developing 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. 1.2. We provide the design principle and detailed derivation of the momentum cell in Sections 3. 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 accuracy of the 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 [15] and Nesterov accelerated gradients with a restart [20, 40].

**Organization.** In Chapter 2, we first review and discuss momentum for accelerating GD in convex smooth optimization. We then present SRSGD algorithm and
we denote its norm of $k$ and more results are provided in the appendix.

each section. We end with concluding remarks. Technical proofs, experimental details, a GD analogy of the RNN and present details of the MomentumRNN with some DNNs for image classification on CIFAR and ImageNet. In Chapter 3, we establish a GD analogy of the RNN and present details of the MomentumRNN with some analysis. We then verify the advantages of the proposed MomentumRNN on several benchmark tasks ranging from small scale image classification to large scale language modeling. We perform empirical analysis of both SRSGD and MomentumRNN in each section. We end with concluding remarks. Technical proofs, experimental details, and more results are provided in the appendix.

**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 $\mathbf{x} = (x_1, \cdots, x_d) \in \mathbb{R}^d$, we denote its $\ell_p$ norm ($p \geq 1$) by $\|\mathbf{x}\|_p = (\sum_{i=1}^{d} |x_i|^p)^{1/p}$. For a matrix $\mathbf{A}$, we use $\|\mathbf{A}\|_p$ to denote its induced norm by the vector $\ell_p$ norm. Also, we denote the spectral norm of $\mathbf{A}$ as $\|\mathbf{A}\|$. We use $\mathbf{A}^T$ (T in roman type) and $\mathbf{A}^{-1}$ to denote its transpose and inverse, respectively. Given two sequences $\{a_n\}$ and $\{b_n\}$, we write $a_n = O(b_n)$ if there exists a positive constant s.t. $a_n \leq Cb_n$. We denote the interval $a$ to $b$ (included)
as \((a, b]\). For a function \(f(\mathbf{w}) : \mathbb{R}^d \to \mathbb{R}\), we denote its gradient as \(\nabla f(\mathbf{w})\) and its Hessian as \(\nabla^2 f(\mathbf{w})\). 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 \(\phi(\mathbf{x}) : \mathbb{R}^d \to \mathbb{R}\), we denote \(\phi^{-1}(\mathbf{x})\) as its inverse and \(\nabla \phi(\mathbf{x})\) as its gradient.
Chapter 2

Scheduled Restart Momentum for Accelerated Stochastic Gradient Descent

2.1 Review: Momentum in Gradient Descent

**GD.** Perhaps the simplest algorithm to solve (1.1) is GD (1.2), which dates back to [41]. If \( f(w) \) is convex and \( L \)-smooth (i.e., \( \|\nabla^2 f(w)\|_2 \leq L \)), then GD converges with rate \( O(1/k) \) by letting \( s_k \equiv 1/L \) (we use this \( s_k \) in all the discussion below), which is independent of the dimension of \( w \).

**HB.** HB (2.1) [14] accelerates GD by using the momentum \( w^k - w^{k-1} \), which gives

\[
\begin{align*}
\mathbf{w}^{k+1} &= \mathbf{w}^k - s_k \nabla f(\mathbf{w}^k) + \mu(\mathbf{w}^k - \mathbf{w}^{k-1}), \quad \mu > 0. \\
\end{align*}
\]  

(2.1)

Alternatively, we can accelerate GD by using the Nesterov momentum (aka, lookahead momentum), which leads to (1.4). Both HB and (1.4) have the same convergence rate of \( O(1/k) \) for solving convex smooth problems. Recently, several variants of (1.4) have been proposed for DL, e.g., [26] and [42].

**NAG.** NAG [20, 43] replaces \( \mu \) with \( (t_k - 1)/t_{k+1} \), where \( t_{k+1} = (1 + \sqrt{1 + 4t_k^2})/2 \) with \( t_0 = 1 \),

\[
\begin{align*}
\mathbf{v}^{k+1} &= \mathbf{w}^k - s_k \nabla f(\mathbf{w}^k); \quad \mathbf{w}^{k+1} = \mathbf{v}^{k+1} + \frac{t_k - 1}{t_{k+1}}(\mathbf{v}^{k+1} - \mathbf{v}^k). \\
\end{align*}
\]  

(2.2)

NAG achieves a convergence rate \( O(1/k^2) \) with the step size \( s_k = 1/L \).

**Remark 1.** Su et al. [21] show that \((k - 1)/(k + 2)\) is the asymptotic limit of
In the following presentation of NAG with restart, for the ease of notation, we will replace the momentum coefficient \((t_k - 1)/t_{k+1}\) with the form of \((k - 1)/(k + 2)\).

Adaptive Restart NAG (ARNAG). The sequences, \(\{f(w^k) - f(w^*)\}\) where \(w^*\) is the minimum of \(f(w)\), generated by GD and GD with constant momentum (GD + Momentum) converge monotonically to zero. However, that sequence generated by NAG oscillates, as illustrated in Fig. 2.1 (a) when \(f(w)\) is a quadratic function. [44] proposes ARNAG (2.3) to alleviate this oscillatory phenomenon

\[
\begin{align*}
v^{k+1} &= w^k - s_k \nabla f(w^k); \quad w^{k+1} = v^{k+1} + \frac{m(k) - 1}{m(k) + 2} (v^{k+1} - v^k),
\end{align*}
\]

where \(m(1) = 1; m(k+1) = m(k) + 1\) if \(f(w^{k+1}) \leq f(w^k)\), and \(m(k+1) = 1\) otherwise.

Scheduled Restart NAG (SRNAG). SR is another strategy to restart NAG. We first divide the total iterations \((0, T] (\text{integers only})\) into a few intervals \(\{I_i\}_{i=1}^m = (T_{i-1}, T_i]\), such that \((0, T] = \bigcup_{i=1}^m I_i\). In each \(I_i\) we restart the momentum after every \(F_i\) iterations. The update rule is then given by:

\[
\begin{align*}
v^{k+1} &= w^k - s_k \nabla f(w^k); \quad w^{k+1} = v^{k+1} + \frac{(k \mod F_i)}{(k \mod F_i) + 3} (v^{k+1} - v^k),
\end{align*}
\]

Both AR and SR accelerate NAG to linear convergence for convex problems with PL condition [45].

Case Study – Quadratic Function. Consider the following quadratic optimization

\[
\min_x f(x) = \frac{1}{2} x^T Lx - x^T b,
\]

where \(L \in \mathbb{R}^{d \times d}\) is the Laplacian of a cycle graph, and \(b\) is a \(d\)-dimensional vector whose first entry is 1 and all the other entries are 0. It is easy to see that \(f(x)\) is

*We take this example from [46].*
convex with Lipschitz constant 4. In particular, we set $d = 1K$ ($1K := 10^3$). We run $T = 50K$ iterations with step size $1/4$. In SRNAG, we restart, i.e., we set the momentum to 0, after every 1K iterations. As shown in Fig. 2.1 (a), GD + Momentum converges faster than GD, while NAG speeds up GD + Momentum dramatically and converges to the minimum in an oscillatory fashion. Both AR and SR accelerate NAG significantly.

### 2.2 Scheduled Restart SGD (SRSGD)

Computing gradient for ERM, (1.1), can be computational costly and memory intensive, especially when the training set is large. In many applications, such as training DNNs, SGD, (1.3), is used. In this section, we will first prove that NAG accumulates error in SGD, then we formulate our new SRSGD as a solution to accelerate the convergence of SGD using NAG momentum.
2.2.1 Uncontrolled Bound of Nesterov Accelerated SGD (NASGD)

Replacing $\nabla f(w^k) := 1/N \sum_{i=1}^{N} \nabla f_i(w^k)$ in (2.2) with the stochastic gradient $1/m \sum_{j=1}^{m} \nabla f_{ij}(w^k)$ for (1.1) accumulates error even for convex function. Theorem 1 formulates this observation for NASGD.

**Theorem 1.** Let $f(w)$ be a convex and $L$-smooth function. The sequence $\{w^k\}_{k \geq 0}$ generated by (2.2), with mini-batch stochastic gradient using any constant step size $s_k \equiv s \leq 1/L$, satisfies

$$\mathbb{E}(f(w^k) - f(w^*)) = O(k),$$

where $w^*$ is the minimum of $f$, and the expectation is taken over the random mini-batch samples.

In Appendix A.1, we provide the proof of Theorem 1. In [25], Devolder et al. prove a similar error accumulation result for the $\delta$-inexact gradient. In Appendix A.2, we provide a brief review of NAG with $\delta$-inexact gradient. We consider three different inexact gradients, namely, Gaussian noise with constant and decaying variance corrupted gradients for the quadratic optimization (2.5), and training logistic regression model for MNIST [47] classification. The detailed settings and discussion are provided in Appendix A.2. We denote SGD with NAG momentum as NASGD, and denote NASGD with AR and SR as ARSGD and SRSGD, respectively. The results shown in Fig. 2.1 (b) and (c) (iteration vs. optimal gap for quadratic optimization (2.5) ), and Fig. 2.2 (a) (iteration vs. loss for training logistic regression model) confirm Theorem 1. Moreover, for these cases SR can improve the performance of NAG with inexact gradients. When inexact gradient is used, GD performs almost the same as ARNAG asymptotically because ARNAG restarts too often and almost degenerates to GD.
2.2.2 SRSGD and Its Convergence

For ERM (1.1), SRSGD replaces $\nabla f(w)$ in (2.4) with stochastic gradient with batch size $m$ and gives

$$v^{k+1} = w^k - s_k \frac{1}{m} \sum_{j=1}^{m} \nabla f_{ij}(w^k); \quad w^{k+1} = v^{k+1} + \frac{(k \mod F_i)}{(k \mod F_i) + 3}(v^{k+1} - w^k), \quad (2.7)$$

where $F_i$ is the restart frequency used in the interval $I_i$. We implemented SRSGD in both PyTorch [48] and Keras [49], by changing just a few lines on top of the existing SGD optimizer. We provide a snippet of SRSGD code in Appendix A.10 (PyTorch) and A.11 (Keras). We formulate the convergence of SRSGD for general nonconvex problems in Theorem 2, and we provide its proof in Appendix A.3.

**Theorem 2.** Suppose $f(w)$ is $L$-smooth. Consider the sequence $\{w^k\}_{k \geq 0}$ generated by (2.7) with mini-batch stochastic gradient and any restart frequency $F$ using any constant step size $s_k := s \leq 1/L$. Assume that the set $A := \{k \in \mathbb{Z}^+ | \mathbb{E}f(w^{k+1}) \geq \mathbb{E}f(w^k)\}$ is finite, then we have

$$\min_{1 \leq k \leq K} \{\mathbb{E}\|\nabla f(w^k)\|_2^2\} = O(s + \frac{1}{sK}). \quad (2.8)$$

Therefore $\forall \epsilon > 0$, to obtain $\epsilon$ error, we just need to set $s = O(\epsilon)$ and $K = O(1/\epsilon^2)$.

2.3 Experimental Results

We evaluate SRSGD on a variety of benchmarks for image classification, including CIFAR10, CIFAR100, and ImageNet. In all experiments, we show the advantage of SRSGD over the widely used and well-calibrated SGD baselines with a constant momentum of 0.9 and decreasing learning rate at certain epochs, which we denote as SGD. We also compare SRSGD with the well-calibrated SGD in which we switch momentum to the Nesterov momentum of 0.9, and we denote this optimizer as SGD +
Figure 2.2: (a) Training loss comparison between different schemes in training logistic regression for MNIST classification. NASGD is not robust to noisy gradient, ARSGD almost degenerates to SGD, and SRSGD performs the best in this case. (b, c) Training loss vs. training epoch of ResNet models trained with SRSGD (blue) and the SGD baseline with constant momentum (red).

NM. We fine tune the SGD and SGD + NM baselines to obtain the best performance, and we then adopt the same set of parameters for training with SRSGD. In the SRSGD experiments, we tune the restart frequencies on small DNNs and apply the tuned restart frequencies to large DNNs. We provide the detailed description of datasets and experimental settings in Appendix A.4. Additional experimental results in training LSTMs [27] and WGANs [50, 51] with SRSGD, as well as the comparison between SRSGD and SGD + NM on ImageNet classification task, are provided in Appendix A.5.

2.3.1 CIFAR10 and CIFAR100

We summarize our results for CIFAR in Tables 2.1 and 2.2. We also explore two different restarting frequency schedules for SRSGD: linear and exponential schedule. These schedules are governed by two parameters: the initial restarting frequency $F_1$ and the growth rate $r$. In both scheduling schemes, the restarting frequency at the 1st learning rate stage is set to $F_1$ during training. Then the restarting frequency at
Table 2.1: Classification test error (%) on CIFAR10 using the SGD, SGD + NM, and SRSGD. We report the results of SRSGD with two different restarting schedules: linear (lin) and exponential (exp). The numbers of iterations after which we restart the momentum in the lin schedule are 30, 60, 90, 120 for the 1st, 2nd, 3rd, and 4th stage. Those numbers for the exp schedule are 40, 50, 63, 78. We also include the reported results from [1] (in parentheses) in addition to our reproduced results.

<table>
<thead>
<tr>
<th>Network</th>
<th># Params</th>
<th>SGD (baseline)</th>
<th>SGD+NM</th>
<th>SRSGD (lin)</th>
<th>SRSGD (exp)</th>
<th>Improve over SGD (lin/exp)</th>
<th>Improve over SGD+NM (lin/exp)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre-ResNet-110</td>
<td>1.1M</td>
<td>5.25 ± 0.14 (6.37)</td>
<td>5.24 ± 0.16</td>
<td>4.93 ± 0.13</td>
<td>5.00 ± 0.47</td>
<td>0.32/0.25</td>
<td>0.31/0.24</td>
</tr>
<tr>
<td>Pre-ResNet-290</td>
<td>3.0M</td>
<td>5.05 ± 0.23</td>
<td>5.04 ± 0.12</td>
<td>4.37 ± 0.15</td>
<td>4.50 ± 0.18</td>
<td>0.68/0.55</td>
<td>0.67/0.54</td>
</tr>
<tr>
<td>Pre-ResNet-470</td>
<td>4.9M</td>
<td>4.92 ± 0.10</td>
<td>4.97 ± 0.15</td>
<td>4.18 ± 0.09</td>
<td>4.49 ± 0.19</td>
<td>0.74/0.43</td>
<td>0.79/0.48</td>
</tr>
<tr>
<td>Pre-ResNet-650</td>
<td>6.7M</td>
<td>4.87 ± 0.14</td>
<td>4.80 ± 0.14</td>
<td>4.00 ± 0.07</td>
<td>4.40 ± 0.13</td>
<td>0.87/0.47</td>
<td>0.80/0.40</td>
</tr>
<tr>
<td>Pre-ResNet-1001</td>
<td>10.3M</td>
<td>4.84 ± 0.19 (4.92)</td>
<td>4.62 ± 0.14</td>
<td>3.87 ± 0.07</td>
<td>4.13 ± 0.10</td>
<td>0.97/0.71</td>
<td>0.75/0.49</td>
</tr>
</tbody>
</table>

the \((k + 1)\)-th learning rate stage is determined by:

\[
F_{k+1} = \begin{cases} 
F_1 \times r^k, & \text{exponential schedule} \\
F_1 \times (1 + (r - 1) \times k), & \text{linear schedule}.
\end{cases}
\]

We conducted a hyper-parameter search for \(F_1\) and \(r\) for both scheduling schemes. For CIFAR10, \((F_1 = 40, r = 1.25)\) and \((F_1 = 30, r = 2)\) are good initial restarting frequencies and growth rates for the exponential and linear schedules, respectively. For CIFAR100, those values are \((F_1 = 45, r = 1.5)\) for the exponential schedule and \((F_1 = 50, r = 2)\) for the linear schedule.

**Improvement in Accuracy Increases with Depth.** We observe that the linear schedule of restart yields better test error on CIFAR than the exponential schedule for most of the models except for Pre-ResNet-470 and Pre-ResNet-1001 on CIFAR100 (see Tables 2.1 and 2.2). SRSGD with either linear or exponential restart schedule outperforms SGD. Furthermore, the advantage of SRSGD over SGD is more significant for deeper networks. This observation holds strictly when using the linear schedule (see Fig. 1.1) and is generally true when using the exponential schedule with only a
Table 2.2: Classification test error (%) on CIFAR100 using the SGD, SGD + NM, and SRSGD. We report the results of SRSGD with two different restarting schedules: linear (lin) and exponential (exp). The numbers of iterations after which we restart the momentum in the lin schedule are 50, 100, 150, 200 for the 1st, 2nd, 3rd, and 4th stage. Those numbers for the exp schedule are 45, 68, 101, 152. We also include the reported results from [1] (in parentheses) in addition to our reproduced results.

<table>
<thead>
<tr>
<th>Network</th>
<th># Params</th>
<th>SGD (baseline)</th>
<th>SGD+NM</th>
<th>SRSGD (lin)</th>
<th>SRSGD (exp)</th>
<th>Improve over SGD (lin/exp)</th>
<th>Improve over SGD+NM (lin/exp)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre-ResNet-110</td>
<td>1.2M</td>
<td>23.75 ± 0.20</td>
<td>23.65 ± 0.36</td>
<td><strong>23.49 ± 0.23</strong></td>
<td>23.50 ± 0.39</td>
<td>0.26/0.25</td>
<td>0.16/0.15</td>
</tr>
<tr>
<td>Pre-ResNet-290</td>
<td>3.0M</td>
<td>21.78 ± 0.21</td>
<td>21.68 ± 0.21</td>
<td><strong>21.49 ± 0.27</strong></td>
<td>21.58 ± 0.20</td>
<td>0.29/0.20</td>
<td>0.19/0.10</td>
</tr>
<tr>
<td>Pre-ResNet-470</td>
<td>4.9M</td>
<td>21.43 ± 0.30</td>
<td>21.21 ± 0.30</td>
<td><strong>20.71 ± 0.32</strong></td>
<td>20.64 ± 0.18</td>
<td>0.72/0.79</td>
<td>0.50/0.57</td>
</tr>
<tr>
<td>Pre-ResNet-650</td>
<td>6.7M</td>
<td>21.27 ± 0.14</td>
<td>21.04 ± 0.38</td>
<td><strong>20.36 ± 0.25</strong></td>
<td>20.41 ± 0.21</td>
<td>0.91/0.86</td>
<td>0.68/0.63</td>
</tr>
<tr>
<td>Pre-ResNet-1001</td>
<td>10.4M</td>
<td>20.87 ± 0.20 (22.71)</td>
<td>20.13 ± 0.16</td>
<td><strong>19.75 ± 0.11</strong></td>
<td><strong>19.53 ± 0.19</strong></td>
<td><strong>1.12/1.34</strong></td>
<td><strong>0.38/0.60</strong></td>
</tr>
</tbody>
</table>

Few exceptions.

Faster Convergence Reduces the Training Time by Half. SRSGD also converges faster than SGD. This result is consistent with our MNIST case study in Section 2.2 and indeed expected since SRSGD can avoid the error accumulation when there is an inexact oracle. For CIFAR, Fig. 2.2 (b) shows that SRSGD yields smaller training loss than SGD during the training. Interestingly, SRSGD converges quickly to good loss values in the 2nd and 3rd stages. This suggests that the model can be trained with SRSGD in many fewer epochs compared to SGD while achieving a similar error rate.

Our numerical results in Table 2.3 confirm the hypothesis above. We train Pre-ResNet models with SRSGD in only 100 epochs, decreasing the learning rate by a factor of 10 at the 80th, 90th, and 95th epoch while using the same linear schedule for restarting frequency as before with \((F_1 = 30, r = 2)\) for CIFAR10 and \((F_1 = 50, r = 2)\) for CIFAR100. We compare the test error of the trained models with those trained by the SGD baseline in 200 epochs. We observe that SRSGD training consistently yields lower test errors than SGD except for the case of Pre-ResNet-110 even though
Table 2.3: Comparison of classification errors on CIFAR10/100 (%) between SRSGD training with only 100 epochs and SGD baseline training with 200 epochs. Using only half the number of training epochs, SRSGD achieves comparable results to SGD.

<table>
<thead>
<tr>
<th>Network</th>
<th>CIFAR10</th>
<th>Improvement</th>
<th>CIFAR100</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre-ResNet-110</td>
<td>5.43 ± 0.18</td>
<td>-0.18</td>
<td>23.85 ± 0.19</td>
<td>-0.10</td>
</tr>
<tr>
<td>Pre-ResNet-290</td>
<td>4.83 ± 0.11</td>
<td>0.22</td>
<td>21.77 ± 0.43</td>
<td>0.01</td>
</tr>
<tr>
<td>Pre-ResNet-470</td>
<td>4.64 ± 0.17</td>
<td>0.34</td>
<td>21.42 ± 0.19</td>
<td>0.02</td>
</tr>
<tr>
<td>Pre-ResNet-650</td>
<td>4.43 ± 0.14</td>
<td>0.44</td>
<td>21.04 ± 0.20</td>
<td>0.23</td>
</tr>
<tr>
<td>Pre-ResNet-1001</td>
<td>4.17 ± 0.20</td>
<td>0.67</td>
<td>20.27 ± 0.11</td>
<td>0.60</td>
</tr>
<tr>
<td>Pre-ResNet-110 (110 epochs)</td>
<td>5.25 ± 0.10</td>
<td>0.00</td>
<td>23.73 ± 0.23  (140 epochs)</td>
<td>0.02</td>
</tr>
</tbody>
</table>

the number of training epochs of our method is only half of the number of training epochs required by SGD. For Pre-ResNet-110, SRSGD training in 110 epochs with learning rate decreased at the 80th, 90th, and 100th epoch achieves the same error rate as the 200-epoch SGD training on CIFAR10. On CIFAR100, SRSGD training for Pre-ResNet-110 needs 140 epochs with learning rate decreased at the 80th, 100th and 120th epoch to achieve an 0.02% improvement in error rate over the 200-epoch SGD.

### 2.3.2 ImageNet

Next, we discuss our experimental results on the 1000-way ImageNet classification task [52]. We conduct our experiments on ResNet-50, 101, 152, and 200 with 5 different seeds. We use the official PyTorch implementation† for all of our ResNet models [48]. Following common practice, we train each model for 90 epochs and decrease the learning rate by a factor of 10 at the 30th and 60th epoch. We use an initial learning rate of 0.1, a momentum scaled by 0.9, and a weight decay value of 0.0001. Additional details and comparisons between SRSGD and SGD + NM are given in Appendix A.5.

†Implementation available at https://github.com/pytorch/examples/tree/master/imagenet
Table 2.4: Single crop validation errors (%) on ImageNet of ResNets trained with SGD baseline and SRSGD. We report the results of SRSGD with the increasing restarting frequency in the first two learning rates. In the last learning rate, the restarting frequency is linearly decreased to 1. For baseline results, we also include the reported single-crop validation errors [2] (in parentheses).

<table>
<thead>
<tr>
<th>Network</th>
<th># Params</th>
<th>SGD</th>
<th>SRSGD</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>top-1</td>
<td>top-5</td>
<td>top-1</td>
</tr>
<tr>
<td>ResNet-50</td>
<td>25.56M</td>
<td>24.11 ± 0.10 (24.70)</td>
<td>7.22 ± 0.14 (7.80)</td>
<td>23.85 ± 0.09</td>
</tr>
<tr>
<td>ResNet-101</td>
<td>44.55M</td>
<td>22.42 ± 0.03 (23.60)</td>
<td>6.22 ± 0.01 (7.10)</td>
<td>22.06 ± 0.10</td>
</tr>
<tr>
<td>ResNet-152</td>
<td>60.19M</td>
<td>22.03 ± 0.12 (23.00)</td>
<td>6.04 ± 0.07 (6.70)</td>
<td>21.46 ± 0.07</td>
</tr>
<tr>
<td>ResNet-200</td>
<td>64.67M</td>
<td>22.13 ± 0.12</td>
<td>6.00 ± 0.07</td>
<td>20.93 ± 0.13</td>
</tr>
</tbody>
</table>

We report single crop validation errors of ResNet models trained with SGD and SRSGD on ImageNet in Table 2.4. In contrast to our CIFAR experiments, we observe that for ResNets trained on ImageNet with SRSGD, linearly decreasing the restarting frequency to 1 at the last stage (i.e., after the 60th epoch) helps improve the generalization of the models. Thus, in our experiments, we use linear scheduling with \((F_1 = 40, r = 2)\). From epoch 60 to 90, the restarting frequency decays to 1 linearly.

**Advantage of SRSGD continues to grow with depth.** Similar to the CIFAR experiments, we observe that SRSGD outperforms the SGD baseline for all ResNet models that we study. As shown in Fig. 1.1, the advantage of SRSGD over SGD grows with network depth, just as in our CIFAR experiments with Pre-ResNet architectures.

**Avoiding Overfitting in ResNet-200.** ResNet-200 is an interesting model that demonstrates that SRSGD is better than the SGD baseline at avoiding overfitting\(^1\). The ResNet-200 trained with SGD has a top-1 error of 22.13\%, higher than the ResNet-152 trained with SGD, which achieves a top-1 error of 22.03\% (see Table 2.4). As pointed out in [1], it is because ResNet-200 suffers from overfitting. The ResNet-200 trained with our SRSGD has a top-1 error of 20.93\%, which is 1.2\% lower than the ResNet-200

\(^1\)By overfitting, we mean that the model achieves low training error but high test error.
Table 2.5: Comparison of single crop validation errors on ImageNet (%) between SRSGD training with fewer epochs and SGD training with full 90 epochs.

<table>
<thead>
<tr>
<th>Network</th>
<th>SRSGD</th>
<th>Reduction</th>
<th>Improvement</th>
<th>Network</th>
<th>SRSGD</th>
<th>Reduction</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>ResNet-50</td>
<td>24.30 ± 0.21</td>
<td>10</td>
<td>−0.19</td>
<td>ResNet-152</td>
<td>21.79 ± 0.07</td>
<td>15</td>
<td>0.24</td>
</tr>
<tr>
<td>ResNet-101</td>
<td>22.32 ± 0.06</td>
<td>10</td>
<td>0.1</td>
<td>ResNet-200</td>
<td>21.92 ± 0.17</td>
<td>30</td>
<td>0.21</td>
</tr>
</tbody>
</table>

trained with the SGD baseline and also lower than the ResNet-152 trained with both SRSGD and SGD, an improvement by 0.53% and 1.1%, respectively. Theoretical understanding of the fact that SRSGD is less overfitting in training deep networks is under our investigation.

**Training ImageNet in Fewer Number of Epochs.** As in the CIFAR experiments, we note that when training on ImageNet, SRSGD converges faster than SGD at the first and last learning rate while quickly reaching a good loss value at the second learning rate (see Fig. 2.2 (c)). This observation suggests that ResNets can be trained with SRSGD in fewer epochs while still achieving comparable error rates to the same models trained by the SGD baseline using all 90 epochs. We summarize the results in Table 2.5. On ImageNet, we note that SRSGD helps reduce the number of training epochs for very deep networks (ResNet-101, 152, 200). For smaller networks like ResNet-50, training with fewer epochs slightly decreases the accuracy.

### 2.3.3 Empirical Analysis

We find that SRSGD training using fewer epochs yields comparable error rates to both the SGD baseline and the SRSGD full training with 200 epochs on CIFAR. We conduct an ablation study to understand the impact of reducing the number of epochs on the final error rate when training with SRSGD on CIFAR10 and ImageNet. In the CIFAR10 experiments, we reduce the number of epochs from 15 to 90 while in the ImageNet experiments, we reduce the number of epochs from 10 to 30. We summarize
Figure 2.3: Test error vs. number of training epochs. Dashed lines are test errors of SGD trained in 200 epochs for CIFAR10 and 90 epochs for ImageNet. For CIFAR, SRSGD with fewer epochs achieves comparable results to SRSGD with 200 epochs. For ImageNet, training with less epochs slightly decreases the performance of SRSGD but still achieves comparable results to 200-epoch SGD.

Impact of Restarting Frequency. We examine the impact of restarting frequency on the network training. We choose a case study of training a Pre-ResNet-290 on CIFAR10 using SRSGD with a linear schedule scheme for the restarting frequency. We fix the growth rate \( r = 2 \) and vary the initial restarting frequency \( F_1 \) from 1 to 80.

As shown in Fig. 2.4, SRSGD with a large \( F_1 \), e.g. \( F_1 = 80 \), approximates NASGD (yellow). As discussed in Section 2.2, it suffers from error accumulation due to stochastic gradients and converges slowly. SRSGD with small \( F_1 \), e.g. \( F_1 = 1 \), approximates SGD without momentum (green). It converges faster initially but reaches a worse local
Figure 2.4: Training loss (left) and test error (right) of Pre-ResNet-290 trained on CIFAR10 with different initial restarting frequencies $F_1$ (linear schedule). SRSGD with small $F_1$ approximates SGD without momentum, while SRSGD with large $F_1$ approximates NASGD. minimum (i.e. larger loss). Typical SRSGD (blue) converges faster than NASGD and to a better local minimum than both NASGD and SGD without momentum. It also achieves the best test error. More analysis results are in Appendix A.6, A.7 and A.8.

2.4 Related Work

Momentum has long been used to accelerate SGD. SGD with scheduled momentum and a good initialization can handle the curvature issues in training DNNs and enable the trained models to generalize well [26]. The works [15, 17] integrate momentum with adaptive step size to accelerate SGD. These works all leverage constant momentum, while our work utilizes NAG momentum with a restart. AR and SR have been used to accelerate NAG with the exact gradient [53, 54, 55, 56, 57, 58, 59, 44, 60, 21]. These studies of restart NAG momentum are for convex optimization with the exact gradient. Our work focuses on SGD for nonconvex optimization with inexact gradients. Many efforts have also been devoted to accelerating first-order algorithms with noise-corrupted gradients [61, 62].
Chapter 3

MomentumRNN: Integrating Momentum into Recurrent Neural Networks

3.1 Review: Recurrent Neural Networks and Long Short Term Memory Networks

Recurrent cells are the building blocks of RNNs. A recurrent cell employs a cyclic connection to update the current hidden state \( h_t \) using the past hidden state \( h_{t-1} \) and the current input data \( x_t \) \([63]\); the dependence of \( h_t \) on \( h_{t-1} \) and \( x_t \) in a recurrent cell can be written as

\[
    h_t = \sigma(U h_{t-1} + W x_t + b), \quad x_t \in \mathbb{R}^d, \quad h_t \in \mathbb{R}^h, \quad t = 1, 2, \ldots, T, \tag{3.1}
\]

where \( U \in \mathbb{R}^{h \times h} \), \( W \in \mathbb{R}^{h \times d} \), and \( b \in \mathbb{R}^h \) 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 \([64]\). Thus RNNs may fail to learn long term dependencies. Several approaches exist to improve RNNs’ performance, including enforcing unitary weight matrices \([65, 4, 66, 5, 67, 3] \) and leveraging LSTM cells.
LSTM cells augment the recurrent cell with “gates” [27] and can be formulated as

\[
\begin{align*}
  i_t &= \sigma(U_{ih} h_{t-1} + W_{ix} x_t + b_i), \quad (i_t : \text{input gate}) \\
  \tilde{c}_t &= \tanh(U_{ch} h_{t-1} + W_{cx} x_t + b_c), \quad (\tilde{c}_t : \text{cell input}) \\
  c_t &= c_{t-1} + i_t \odot \tilde{c}_t, \quad (c_t : \text{cell state}) \\
  o_t &= \sigma(U_{oh} h_{t-1} + W_{ox} x_t + b_o), \quad (o_t : \text{output gate}) \\
  h_t &= o_t \odot \tanh(c_t), \quad (h_t : \text{hidden state})
\end{align*}
\]

where \( U_s \in \mathbb{R}^{h \times h}, W_s \in \mathbb{R}^{h \times d}, \) and \( b_s \in \mathbb{R}^h \) 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.

### 3.2 Momentum Recurrent Neural Networks

#### 3.2.1 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 \( \widetilde{W} = [W, b] \) and \( \tilde{x}_t = [x_t, 1]^T \) in (3.1), then we have \( h_t = \sigma(U h_{t-1} + \widetilde{W} \tilde{x}_t). \) For the ease of notation, without ambiguity we denote \( W := \widetilde{W} \) and \( x_t := \tilde{x}_t. \) Then the recurrent cell can be reformulated as

\[
    h_t = \sigma(U h_{t-1} + W x_t). \tag{3.3}
\]

Moreover, let \( \phi(\cdot) := \sigma(U(\cdot)) \) and \( u_t := U^{-1} W x_t, \) we can rewrite (3.3) as

\[
    h_t = \phi(h_{t-1} + u_t). \tag{3.4}
\]

If we regard \(-u_t\) as the “gradient” at the \( t \)-th iteration, then we can consider (3.4) as the dynamical system which updates the hidden state by the gradient and then
transforms the updated hidden state by the nonlinear activation function $\phi$. We propose the following accelerated dynamical system to accelerate the dynamics of (3.4), which is principled by the accelerated gradient descent theory (see subsection ??):

$$
 p_t = \mu p_{t-1} - su_t; \quad h_t = \phi(h_{t-1} - p_t),
$$

(3.5)

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 $v_t := -Up_t$, we arrive at the following dynamical system:

$$
 v_t = \mu v_{t-1} + sWx_t; \quad h_t = \sigma(Uh_{t-1} + v_t).
$$

(3.6)

The architecture of the momentum cell that corresponds to the dynamical system (3.6) is plotted in Fig. 1.2 (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 $\mu$ and $s$.

**Remark 2.** Different parameterizations of (3.5) can result in different momentum cell architectures. For instance, if we let $v_t = -p_t$, we end up with the following dynamical system:

$$
 v_t = \mu v_{t-1} + s\hat{W}x_t; \quad h_t = \sigma(Uh_{t-1} + Uv_t),
$$

(3.7)

where $\hat{W} := U^{-1}W$ is the trainable weight matrix. Even though (3.6) and (3.7) are mathematically equivalent, the training procedure might cause the MomentumRNNs that are derived from different parameterizations to have different performances.

**Remark 3.** We put the nonlinear activation in the second equation of (3.5) to ensure that the value of $h_t$ is in the same range as the original recurrent cell.
Remark 4. 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.

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

Let \( h_T \) and \( h_t \) 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

\[
\frac{\partial \mathcal{L}}{\partial h_t} = \frac{\partial \mathcal{L}}{\partial h_T} \cdot \frac{\partial h_T}{\partial h_t} = \frac{\partial \mathcal{L}}{\partial h_T} \cdot \prod_{k=t}^{T-1} \frac{\partial h_{k+1}}{\partial h_k} = \frac{\partial \mathcal{L}}{\partial h_T} \cdot \prod_{k=t}^{T-1} (D_k U^T),
\]

(3.8)

where \( U^T \) is the transpose of \( U \) and \( D_k = \text{diag}(\sigma'(U h_k + W x_{k+1})) \) is a diagonal matrix with \( \sigma'(U h_k + W x_{k+1}) \) being its diagonal entries. \( \| \prod_{k=t}^{T-1} (D_k U^T) \|_2 \) tends to either vanish or explode [64]. 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 [68]. We can rewrite (3.6) as

\[
h_t = \sigma \left( U(h_{t-1} - \mu h_{t-2}) + \mu \sigma^{-1}(h_{t-1}) + s W x_t \right),
\]

(3.9)

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

\[
\frac{\partial \mathcal{L}}{\partial h_t} = \frac{\partial \mathcal{L}}{\partial h_T} \cdot \frac{\partial h_T}{\partial h_t} = \frac{\partial \mathcal{L}}{\partial h_T} \cdot \prod_{k=t}^{T-1} \frac{\partial h_{k+1}}{\partial h_k} = \frac{\partial \mathcal{L}}{\partial h_T} \cdot \prod_{k=t}^{T-1} \hat{D}_k [U^T + \mu \Sigma_k],
\]

(3.10)

where \( \hat{D}_k = \text{diag}(\sigma'(U(h_k - \mu h_{k-1}) + \mu \sigma^{-1}(h_k) + s W x_{k+1})) \) and \( \Sigma = \text{diag}((\sigma^{-1})'(h_k)) \).

For mostly used \( \sigma \), e.g., sigmoid and tanh, \( (\sigma^{-1}(\cdot))' > 1 \) and \( \mu \Sigma_k \) dominants \( U^T \).*

Therefore, with an appropriate choice of \( \mu \), the momentum cell can alleviate vanishing gradient and accelerate training.

*In the vanishing gradient scenario, \( \|U\|_2 \) is small; also it can be controlled by regularizing the loss function.
3.2.3 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 are derived from the Nesterov accelerated gradient (NAG)-style momentum with restart [20, 40] and Adam [15].

NAG Principled RNNs. The momentum-accelerated GD can be further accelerated by replacing the constant momentum coefficient $\mu$ in (3.6) with the NAG-style momentum, i.e. setting $\mu$ 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 $F$ [40]. 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 [40]. 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 [15] leverages the moving average of historical gradients and entry-wise squared gradients to accelerate the stochastic gradient dynamics. We use Adam to accelerate (3.4) and end up with the following iteration

$$p_t = \mu p_{t-1} - (1-\mu)u_t; \quad m_t = \beta m_{t-1} + (1-\beta)u_t \odot u_t; \quad h_t = \phi(h_{t-1} - s \frac{p_t}{\sqrt{r_t} + \epsilon}),$$

(3.11)

where $\mu, s, \beta > 0$ are hyperparameters, $\epsilon$ is a small constant and chosen to be $10^{-8}$ by default, and $\odot/\sqrt{}$ denotes the entrywise product/square root†. Again, let $v_t = -U p_t$, we rewrite (3.11) as follows

$$v_t = \mu v_{t-1} + (1-\mu)W x_t; \quad m_t = \beta m_{t-1} + (1-\beta)u_t \odot u_t; \quad h_t = \sigma(U h_{t-1} + s \frac{v_t}{\sqrt{m_t} + \epsilon}).$$

As before, here $u_t := U^{-1} W x_t$. Computing $U^{-1}$ is expensive. Our experiments suggest that replacing $u_t \odot u_t$ by $W x_t \odot W x_t$ is sufficient and more efficient to compute. In our implementation, we also relax $v_t = \mu v_{t-1} + (1-\mu)W x_t$ to $v_t = \mu v_{t-1} + sW x_t$ that follows the momentum in the MomentumRNN (3.6) for better performance. Therefore,†

†In contrast to Adam, we do not normalize $p_t$ and $m_t$ since they can be absorbed in the weight matrices.
we propose the AdamRNN that is given by

\[ v_t = \mu v_{t-1} + s W x_t; \quad m_t = \beta m_{t-1} + (1 - \beta)(W x_t \odot W x_t); \quad h_t = \sigma(U h_{t-1} + \frac{v_t}{\sqrt{m_t} + \epsilon}). \]

(3.12)

In AdamRNN, if \( \mu \) is set to 0, we achieve another new RNN, which obeys the RMSProp gradient update rule \([69]\). We call this new model the RMSPropRNN.

**Remark 5.** Both AdamRNN and RMSPropRNN can also be derived by letting \( v_t = -p_t \) and \( \hat{W} := U^{-1}W \) as in Remark 2. This parameterization yields the following formulation for AdamRNN

\[ v_t = \mu v_{t-1} + s \hat{W} x_t; \quad m_t = \beta m_{t-1} + (1 - \beta)(\hat{W} x_t \odot \hat{W} x_t); \quad h_t = \sigma(U h_{t-1} + \frac{U v_t}{\sqrt{m_t} + \epsilon}). \]

Here, we simply need to learn \( \hat{W} \) and \( U \) without any relaxation. In contrast, we relaxed \( U^{-1} \) to an identity matrix in (3.12). Our experiments suggest that both parameterizations yield similar results.

### 3.3 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 \([27]\) in the following tasks: 1) the object classification task on pixel-permuted MNIST \([70]\), 2) the speech prediction task on the TIMIT dataset \([65, 3, 4, 67, 71]\), and 3) the language modeling task on the Penn TreeBank (PTB) dataset \([72]\). 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 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 B.4. 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 B.1. For MNIST and TIMIT experiments, we use the baseline codebase provided by [73]. For PTB experiments, we use the baseline codebase provided by [74].

### 3.3.1 Pixel-by-Pixel MNIST

In this task, we classify image samples of hand-written digits from the MNIST dataset [75] into one of the ten classes. Following the implementation of [70], we flatten the image of original size $28 \times 28$ 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 3.1. 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 3.2
Table 3.1: Best test accuracy at the MNIST and PMNIST tasks (%). We use the baseline results reported in [3], [4], [5]. 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.

<table>
<thead>
<tr>
<th>Model</th>
<th>N</th>
<th># Params</th>
<th>MNIST</th>
<th>PMNIST</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSTM</td>
<td>128</td>
<td>$\approx 68K$</td>
<td>98.70 [3], 97.30 [5]</td>
<td>92.00 [3], 92.62 [5]</td>
</tr>
<tr>
<td>LSTM</td>
<td>256</td>
<td>$\approx 270K$</td>
<td>98.90 [3], 98.50 [4]</td>
<td>92.29 [3], 92.10 [4]</td>
</tr>
<tr>
<td>MomentumLSTM</td>
<td>128</td>
<td>$\approx 68K$</td>
<td><strong>99.04 ± 0.04</strong></td>
<td><strong>93.40 ± 0.25</strong></td>
</tr>
<tr>
<td>MomentumLSTM</td>
<td>256</td>
<td>$\approx 270K$</td>
<td><strong>99.08 ± 0.05</strong></td>
<td><strong>94.72 ± 0.16</strong></td>
</tr>
<tr>
<td>AdamLSTM</td>
<td>256</td>
<td>$\approx 270K$</td>
<td>99.09 ± 0.03</td>
<td>95.05 ± 0.37</td>
</tr>
<tr>
<td>RMSPropLSTM</td>
<td>256</td>
<td>$\approx 270K$</td>
<td><strong>99.15 ± 0.06</strong></td>
<td><strong>95.38 ± 0.19</strong></td>
</tr>
<tr>
<td>SR LSTM</td>
<td>256</td>
<td>$\approx 270K$</td>
<td>99.01 ± 0.07</td>
<td>93.82 ± 1.85</td>
</tr>
</tbody>
</table>

(left two panels) corroborates this observation when using $N = 256$ hidden units.

### 3.3.2 TIMIT Speech Dataset

We study how MomentumLSTM performs on audio data with speech prediction experiments on the TIMIT speech dataset [76], which is a collection of real-world speech recordings. As first proposed by [4], 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 [4, 77, 6], 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 3.2. Results are
Figure 3.2: 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 3.3: Train (left) and test loss (right) of MomentumLSTM (blue) and LSTM (red) for the Penn Treebank language modeling tasks at word level.

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 3.2 (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 [4, 77, 6] to generate the preprocessed data for our experiments and did our best to reproduce the baseline results. In Table 3.2 and 3.5, we include both our reproduced results and the ones reported from previous works.
Table 3.2: 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.

<table>
<thead>
<tr>
<th>Model</th>
<th>$n$</th>
<th># params</th>
<th>Val. MSE</th>
<th>Test MSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSTM</td>
<td>84</td>
<td>$\approx 83K$</td>
<td>14.87 ± 0.15 (15.42 [3, 77])</td>
<td>14.94 ± 0.15 (14.30 [3, 77])</td>
</tr>
<tr>
<td>LSTM</td>
<td>120</td>
<td>$\approx 135K$</td>
<td>11.77 ± 0.14 (13.93 [3, 77])</td>
<td>11.83 ± 0.12 (12.95 [3, 77])</td>
</tr>
<tr>
<td>LSTM</td>
<td>158</td>
<td>$\approx 200K$</td>
<td>9.33 ± 0.14 (13.66 [3, 77])</td>
<td>9.37 ± 0.14 (12.62 [3, 77])</td>
</tr>
<tr>
<td>MomentumLSTM</td>
<td>84</td>
<td>$\approx 83K$</td>
<td>10.90 ± 0.19</td>
<td>10.98 ± 0.18</td>
</tr>
<tr>
<td>MomentumLSTM</td>
<td>120</td>
<td>$\approx 135K$</td>
<td>8.00 ± 0.30</td>
<td>8.04 ± 0.30</td>
</tr>
<tr>
<td>MomentumLSTM</td>
<td>158</td>
<td>$\approx 200K$</td>
<td>5.86 ± 0.14</td>
<td>5.87 ± 0.15</td>
</tr>
<tr>
<td>AdamLSTM</td>
<td>158</td>
<td>$\approx 200K$</td>
<td>8.66 ± 0.15</td>
<td>8.69 ± 0.14</td>
</tr>
<tr>
<td>RMSPropLSTM</td>
<td>158</td>
<td>$\approx 200K$</td>
<td>9.13 ± 0.33</td>
<td>9.17 ± 0.33</td>
</tr>
<tr>
<td>SRLSTM</td>
<td>158</td>
<td>$\approx 200K$</td>
<td>5.81 ± 0.10</td>
<td>5.83 ± 0.10</td>
</tr>
</tbody>
</table>

3.3.3 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 [72], 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 3.3 shows the test and validation perplexity (PPL) using the model parameters that yield the best validation loss. Again, MomentumLSTM achieves better perplexities and
Table 3.3 : Model test perplexity at the end of the epoch with the lowest validation perplexity for the Penn Treebank language modeling task (word level).

<table>
<thead>
<tr>
<th>Model</th>
<th># params</th>
<th>Val. PPL</th>
<th>Test PPL</th>
</tr>
</thead>
<tbody>
<tr>
<td>lstm</td>
<td>≈ 24M</td>
<td>61.96 ± 0.83</td>
<td>59.71 ± 0.99 (58.80 [78])</td>
</tr>
<tr>
<td>MomentumLSTM</td>
<td>≈ 24M</td>
<td>60.71 ± 0.24</td>
<td>58.62 ± 0.22</td>
</tr>
</tbody>
</table>

converges faster than the baseline LSTM (see Figure 3.3).

3.3.4 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 3.1 and 3.2 and provide the plots of train and test losses in Figure 3.2.

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 MomentumLSTM 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.

3.4 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
Table 3.4: 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.

<table>
<thead>
<tr>
<th>N</th>
<th># Params</th>
<th>PMNIST (DTRIV)</th>
<th>PMNIST (MomentumDTRIV)</th>
</tr>
</thead>
<tbody>
<tr>
<td>170</td>
<td>≈ 16K</td>
<td>95.21 ± 0.10 (95.20 [6])</td>
<td>95.37 ± 0.09</td>
</tr>
<tr>
<td>360</td>
<td>≈ 69K</td>
<td>96.45 ± 0.10 (96.50 [6])</td>
<td>96.73 ± 0.08</td>
</tr>
<tr>
<td>512</td>
<td>≈ 137K</td>
<td>96.62 ± 0.12 (96.80 [6])</td>
<td>96.89 ± 0.08</td>
</tr>
</tbody>
</table>

Figure 3.4: 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 overfitting while MomentumDTRIV overfits less.

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 3.4 and 3.5 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 3.4).
Table 3.5: 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.

<table>
<thead>
<tr>
<th>Model</th>
<th>N</th>
<th># Params</th>
<th>Val. MSE</th>
<th>Test MSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>DTRIV</td>
<td>224</td>
<td>83K</td>
<td>4.74 ± 0.06 (4.75 [6])</td>
<td>4.70 ± 0.07 (4.71 [6])</td>
</tr>
<tr>
<td>DTRIV</td>
<td>322</td>
<td>135K</td>
<td>1.92 ± 0.17 (3.39 [6])</td>
<td>1.87 ± 0.17 (3.76 [6])</td>
</tr>
<tr>
<td>MomentumDTRIV</td>
<td>224</td>
<td>83K</td>
<td>3.10 ± 0.09</td>
<td>3.06 ± 0.09</td>
</tr>
<tr>
<td>MomentumDTRIV</td>
<td>322</td>
<td>135K</td>
<td>1.21 ± 0.05</td>
<td>1.17 ± 0.05</td>
</tr>
</tbody>
</table>

Figure 3.5: 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 3.5. The result in each cell is averaged over 5 runs.
3.5 Related Work

**Dynamical system viewpoint of RNNs.** Leveraging the theory of dynamical system to improve RNNs has been an interesting research direction: [79] proposes a gated RNN, which is principled by non-chaotical dynamical systems and achieves comparable performance to GRUs and LSTMs. [80] 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 [81, 82, 83, 84]. Our 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 [14, 24, 26, 15, 42, 48] and sampling algorithms [85, 86] A particularly interesting momentum is the iteration-dependent one in NAG [20, 53, 43], 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 [40].

**Momentum in DNNs.** Momentum has also been used in designing DNN architectures. [87] 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 [43], in which momentum can be used in the underpinning optimizer [88, 89, 90, 91, 92].
Chapter 4

Conclusion and Broader Impact

In this thesis, for speeding up DNN’s training, we propose the Scheduled Restart SGD (SRSGD), with two major changes from the widely used SGD with constant momentum. First, we replace the momentum in SGD with the iteration-dependent momentum that used in Nesterov accelerated gradient (NAG). Second, we restart the NAG momentum according to a schedule to prevent error accumulation when the stochastic gradient is used. For image classification, SRSGD can significantly improve the accuracy of the trained DNNs. Also, compared to the SGD baseline, SRSGD requires fewer training epochs to reach the same trained model’s accuracy. There are numerous avenues for future work: 1) deriving the optimal restart scheduling and the corresponding convergence rate of SRSGD, 2) integrating the scheduled restart NAG momentum with adaptive learning rate algorithms, e.g., Adam [15], and 3) integrating SRSGD with optimizers that remove noise on the fly, e.g., Laplacian smoothing SGD [93].

Furthermore, for facilitating the process of designing a new RNN architecture, 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 momen-
tum 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.

**Broader Impact.** Nesterov acceleration is a well-established technique for accelerating gradient descent; it is used in many applications, including compressed sensing, image reconstruction, and inverse problems. The development of robust Nesterov accelerated stochastic gradient descent provides a solution not only for training deep neural nets but also for ultra-large-scale scientific computing. One particular example is the potential use of this algorithm to enhance the single-particle CryoEM reconstruction [94], an important technique for understanding biomolecular structures.

Recurrent neural net (RNN) is among the most important classes of deep learning models. Improving training efficiency and generalization performance of RNNs not only advances image classification and language modeling but also benefits epidemiological models for pandemic disease prediction. RNNs have also been successfully used for the molecular generation [95]. Developing better RNNs that enable modeling of long term dependency, such as our Momentum RNN, has the potential to facilitate life science research.
Appendix A

Additional Details and Results for Scheduled Restart Stochastic Gradient Descent

A.1 Uncontrolled Bound of NASGD

Consider the following optimization problem

\[ \min_w f(w), \quad (A.1) \]

where \( f(w) \) is \( L \)-smooth and convex.

Start from \( w^k \), GD update, with step size \( \frac{1}{r} \), can be obtained based on the minimization of the functional

\[ \min_v Q_r(v, w^k) := \langle v - w^k, \nabla f(w^k) \rangle + \frac{r}{2} \| v - w^k \|_2^2. \quad (A.2) \]

With direct computation, we can get that

\[ Q_r(v^{k+1}, w^k) - \min_v Q_r(v, w^k) = \frac{\| g^k - \nabla f(w^k) \|_2^2}{2r}, \]

where \( g^k := \frac{1}{m} \sum_{j=1}^{m} \nabla f_j(w^k) \). We assume the variance is bounded, which gives The stochastic gradient rule, \( R_s \), satisfies \( \mathbb{E}[Q_r(v^{k+1}, w^k) - \min_v Q_r(v, w^k) | \chi^k] \leq \delta \), with \( \delta \) being a constant and \( \chi^k \) being the sigma algebra generated by \( w^1, w^2, \cdots, w^k \), i.e.,

\[ \chi^k := \sigma(w^1, w^2, \cdots, w^k). \]

NASGD can be reformulated as
\( v^{k+1} \approx \min_{v} Q_{r}(v, w^k) \) with rule \( R_s \),

\[
\begin{align*}
    w^{k+1} &= v^{k+1} + \frac{t_k - 1}{t_{k+1}} (v^{k+1} - v^k), \\
    t_0 &= 1 \text{ and } t_{k+1} = (1 + \sqrt{1 + 4t_k^2})/2.
\end{align*}
\]  

where \( t_0 = 1 \) and \( t_{k+1} = (1 + \sqrt{1 + 4t_k^2})/2 \).  

\textbf{A.1.1 Preliminaries}  

To proceed, we introduce several definitions and some useful properties in variational and convex analysis. More detailed background can be found at [96, 97, 98, 99].  

Let \( f \) be a convex function, we say that \( f \) is \( L \)-smooth (gradient Lipschitz) if \( f \) is differentiable and  

\[
\|\nabla f(v) - \nabla f(w)\|_2 \leq L \|v - w\|_2,
\]

and we say \( f \) is \( \nu \)-strongly convex if for any \( w, v \in \text{dom}(J) \)  

\[
f(w) \geq f(v) + \langle \nabla f(v), w - v \rangle + \frac{\nu}{2} \|w - v\|^2_2.
\]

Below of this subsection, we list several basic but useful lemmas, the proof can be found in [97].  

\textbf{Lemma 1.} If \( f \) is \( \nu \)-strongly convex, then for any \( v \in \text{dom}(J) \) we have  

\[
f(v) - f(v^*) \geq \frac{\nu}{2} \|v - v^*\|^2_2,
\]

where \( v^* \) is the minimizer of \( f \).  

\textbf{Lemma 2.} If \( f \) is \( L \)-smooth, for any \( w, v \in \text{dom}(f) \),  

\[
f(w) \leq f(v) + \langle \nabla f(v), w - v \rangle + \frac{L}{2} \|w - v\|^2_2.
\]
A.1.2 Uncontrolled Bound of NASGD

In this part, we denote
\[ \tilde{v}^{k+1} := \min_v Q_r(v, w^k). \]  \hfill (A.5)

**Lemma 3.** If the constant \( r > 0 \), then
\[ \mathbb{E} \left( \|v^{k+1} - \tilde{v}^{k+1}\|_2^2 | x^k \right) \leq \frac{2\delta}{r}. \]  \hfill (A.6)

**Proof.** Note that \( Q_r(v, w^k) \) is strongly convex with constant \( r \), and \( \tilde{v}^{k+1} \) in (A.5) is the minimizer of \( Q_r(v, w^k) \). With Lemma 1 we have
\[ Q_r(v^{k+1}, w^k) - Q_r(\tilde{v}^{k+1}, w^k) \geq \frac{r}{2} \|v^{k+1} - \tilde{v}^{k+1}\|_2^2. \]  \hfill (A.7)

Notice that
\[ \mathbb{E} \left[ Q_r(v^{k+1}, w^k) - Q_r(\tilde{v}^{k+1}, w^k) \right] = \mathbb{E} \left[ Q_r(v^{k+1}, w^k) - \min_v Q_r(v, w^k) \right] \leq \delta. \]

The inequality (A.6) can be established by combining the above two inequalities.

**Lemma 4.** If the constant satisfy \( r > L \), then we have
\[ \mathbb{E} \left( f(\tilde{v}^{k+1}) + \frac{r}{2} \|\tilde{v}^{k+1} - w^k\|_2^2 - f(v^{k+1}) + \frac{r}{2} \|v^{k+1} - w^k\|_2^2 \right) \geq -\tau \delta - \frac{r - L}{2} \mathbb{E} \left[ \|w^k - \tilde{v}^{k+1}\|_2^2 \right], \]  \hfill (A.8)

where \( \tau = \frac{L^2}{r(r-L)}. \)

**Proof.** The convexity of \( f \) gives us
\[ 0 \leq \langle \nabla f(v^{k+1}), v^{k+1} - \tilde{v}^{k+1} \rangle + f(\tilde{v}^{k+1}) - f(v^{k+1}), \]  \hfill (A.9)
From the definition of the stochastic gradient rule $\mathcal{R}_s$, we have

$$ -\delta \leq \mathbb{E}(Q_s(\tilde{v}^{k+1}, w^k) - Q_s(v^{k+1}, w^k)) \leq \mathbb{E}\left[\langle \tilde{v}^{k+1} - w^k, \nabla f(w^k) \rangle + \frac{r}{2}\|v^{k+1} - w^k\|_2^2 \right] - \mathbb{E}\left[\langle v^{k+1} - w^k, \nabla f(w^k) \rangle + \frac{r}{2}\|v^{k+1} - w^k\|_2^2 \right]. $$

(A.10)

With (A.9) and (A.10), we have

$$ -\delta \leq \left( f(v^{k+1}) + \frac{r}{2}\|v^{k+1} - w^k\|_2^2 \right) - \left( f(\tilde{v}^{k+1}) + \frac{r}{2}\|\tilde{v}^{k+1} - w^k\|_2^2 \right) + \mathbb{E}\langle \nabla f(w^k) - \nabla f(\tilde{v}^{k+1}), v^{k+1} - \tilde{v}^{k+1} \rangle. $$

(A.11)

With the Schwarz inequality $\langle a, b \rangle \leq \frac{\|a\|^2}{2\mu} + \frac{\mu}{2}\|b\|^2$ with $\mu = \frac{L^2}{r-L}$, $a = \nabla f(v^{k+1}) - \nabla f(\tilde{v}^{k+1})$ and $b = w^k - \tilde{v}^{k+1}$,

$$ \langle \nabla f(w^k) - \nabla f(\tilde{v}^{k+1}), v^{k+1} - \tilde{v}^{k+1} \rangle \leq \frac{(r-L)}{2L^2}\|\nabla f(w^k) - \nabla f(\tilde{v}^{k+1})\|_2^2 + \frac{L^2}{2(r-L)}\|v^{k+1} - \tilde{v}^{k+1}\|_2^2 $$

$$ \leq \frac{(r-L)}{2}\|w^k - \tilde{v}^{k+1}\|_2^2 + \frac{L^2}{2(r-L)}\|v^{k+1} - \tilde{v}^{k+1}\|_2^2. $$

(A.12)

Combining (A.11) and (A.12), we have

$$ -\delta \leq \mathbb{E}\left( f(v^{k+1}) + \frac{r}{2}\|v^{k+1} - w^k\|_2^2 \right) - \mathbb{E}\left( f(\tilde{v}^{k+1}) + \frac{r}{2}\|\tilde{v}^{k+1} - w^k\|_2^2 \right) + \frac{L^2}{2(r-L)}\mathbb{E}\|v^{k+1} - \tilde{v}^{k+1}\|_2^2 + \frac{r-L}{2}\mathbb{E}\|w^k - \tilde{v}^{k+1}\|_2^2. $$

(A.13)

By rearrangement of the above inequality (A.13) and using Lemma 3, we obtain the result.

Lemma 5. If the constants satisfy $r > L$, then we have the following bounds

$$ \mathbb{E}(f(v^k) - f(v^{k+1})) \geq \frac{r}{2}\mathbb{E}\|w^k - v^{k+1}\|_2^2 + r\mathbb{E}\langle w^k - v^k, \tilde{v}^{k+1} - v^k \rangle - \tau\delta, $$

(A.14)
\[ \mathbb{E} \left( f(v^*) - f(v^{k+1}) \right) \geq \frac{r}{2} \mathbb{E} \| w^k - v^{k+1} \|_2^2 + r \mathbb{E} \langle w^k - v^*, v^{k+1} - w^k \rangle - \tau \delta, \tag{A.15} \]

where \( \tau := \frac{L^2}{r(\rho - L)} \) and \( v^* \) is the minimum.

Proof. With Lemma 2, we have

\[ -f(\tilde{v}^{k+1}) \geq -f(w^k) - \langle \tilde{v}^{k+1} - w^k, \nabla f(w^k) \rangle - \frac{L}{2} \| \tilde{v}^{k+1} - w^k \|_2^2. \tag{A.16} \]

Using the convexity of \( f \), we have

\[ f(v^k) - f(w^k) \geq \langle v^k - w^k, \nabla f(w^k) \rangle, \]

i.e.,

\[ f(v^k) \geq f(w^k) + \langle v^k - w^k, \nabla f(w^k) \rangle. \tag{A.17} \]

According to the definition of \( \tilde{v}^{k+1} \) in (A.2), i.e.,

\[ \tilde{v}^{k+1} = \min_v Q_r(v, w^k) = \min_v \langle v - w^k, \nabla f(w^k) \rangle + \frac{r}{2} \| v - w^k \|_2^2, \]

and the optimization condition gives

\[ \tilde{v}^{k+1} = w^k - \frac{1}{r} \nabla f(w^k). \tag{A.18} \]

Substituting (A.18) into (A.17), we obtain

\[ f(v^k) \geq f(w^k) + \langle v^k - w^k, r(w^k - \tilde{v}^{k+1}) \rangle. \tag{A.19} \]

Direct summation of (A.16) and (A.19) gives

\[ f(v^k) - f(\tilde{v}^{k+1}) \geq \left( r - \frac{L}{2} \right) \| \tilde{v}^{k+1} - w^k \|_2^2 + r \langle w^k - v^k, \tilde{v}^{k+1} - w^k \rangle. \tag{A.20} \]

Summing (A.20) and (A.8), we obtain the inequality (A.14)

\[ \mathbb{E} \left[ f(v^k) - f(v^{k+1}) \right] \geq \frac{r}{2} \mathbb{E} \| w^k - v^{k+1} \|_2^2 + r \mathbb{E} \langle w^k - v^k, \tilde{v}^{k+1} - w^k \rangle - \tau \delta. \tag{A.21} \]
On the other hand, with the convexity of $f$, we have

$$f(v^*) - f(w^k) \geq \langle v^* - w^k, \nabla f(w^k) \rangle = \langle v^* - w^k, r(w^k - \tilde{v}^{k+1}) \rangle. \quad (A.22)$$

The summation of (A.16) and (A.22) resulting in

$$f(v^*) - f(\tilde{v}^{k+1}) \geq \left( r - \frac{L}{2} \right) \|w^k - \tilde{v}^{k+1}\|_2^2 + r\langle w^k - v^*, \tilde{v}^{k+1} - w^k \rangle. \quad (A.23)$$

Summing (A.23) and (A.8), we obtain

$$\mathbb{E} \left( f(v^*) - f(v^{k+1}) \right) \geq \frac{r}{2} \mathbb{E}\|w^k - v^{k+1}\|_2^2 + r\mathbb{E}\langle w^k - v^*, \tilde{v}^{k+1} - w^k \rangle - \tau \delta, \quad (A.24)$$

which is the same as (A.15).

**Theorem 3** (Uncontrolled Bound of NASGD (Theorem 1 restate)). Let the constant satisfies $r < L$ and the sequence $\{v^k\}_{k \geq 0}$ be generated by NASGD, then we have

$$\mathbb{E}[f(v^k) - \min_v f(v)] = O(k). \quad (A.25)$$

**Proof.** We denote

$$F^k := \mathbb{E}(f(v^k) - f(v^*)).$$

By (A.14) $\times (t_k - 1) + (A.15)$, we have

$$\frac{2[t_k - 1]F^k - t_k F^{k+1}}{r} \geq t_k \mathbb{E}\|v^{k+1} - w^k\|_2^2 \quad (A.26)$$

$$+ 2\mathbb{E}\langle \tilde{v}^{k+1} - w^k, t_k w^k - (t_k - 1)v^k - v^* \rangle - \frac{2\tau t_k \delta}{r}.$$ 

With $t_{k-1}^2 = t_k^2 - t_k$, (A.26) $\times t_k$ yields

$$\frac{2[t_{k-1}^2 F^k - t_k^2 F^{k+1}]}{r} \geq \mathbb{E}\|t_k v^{k+1} - t_k w^k\|_2^2 \quad (A.27)$$

$$+ 2t_k \mathbb{E}\langle \tilde{v}^{k+1} - w^k, t_k w^k - (t_k - 1)v^k - v^* \rangle - \frac{2\tau t_k^2 \delta}{r}$$
Substituting \(a = t_k v^{k+1} - (t_k - 1) v^k - v^*\) and \(b = t_k w^k - (t_k - 1) v^k - v^*\) into identity

\[
\|a - b\|_2^2 + 2(a - b, b) = \|a\|_2^2 - \|b\|_2^2.
\]  \phantom{.} (A.28)

It follows that

\[
\begin{align*}
\mathbb{E}\|t_k v^{k+1} - t_k w^k\|_2^2 + 2t_k \mathbb{E}\langle \tilde{v}^{k+1} - w^k, t_k w^k - (t_k - 1) v^k - v^* \rangle & = \\
\mathbb{E}\|t_k v^{k+1} - t_k w^k\|_2^2 + 2t_k \mathbb{E}\langle v^{k+1} - w^k, t_k w^k - (t_k - 1) v^k - v^* \rangle + 2t_k \mathbb{E}\langle \tilde{v}^{k+1} - v^{k+1}, t_k w^k - (t_k - 1) v^k - v^* \rangle & = \\
\mathbb{E}\|t_k v^{k+1} - (t_k - 1) v^k - v^*\|_2^2 - \|t_k w^k - (t_k - 1) v^k - v^*\|_2^2 + 2t_k \mathbb{E}\langle \tilde{v}^{k+1} - v^{k+1}, t_k w^k - (t_k - 1) v^k - v^* \rangle & = \\
+ 2t_k \mathbb{E}\langle \tilde{v}^{k+1} - v^{k+1}, t_k v^k - (t_k - 1) v^{k-1} - v^* \rangle.
\end{align*}
\]

In the third identity, we used the fact \(t_k w^k = t_k v^k + (t_k - 1)(v^k - v^{k-1})\). If we denote \(u^k = \mathbb{E}\|t_{k-1} v^k - (t_{k-1} - 1)v^{k-1} - v^*\|_2^2\), (A.27) can be rewritten as

\[
\frac{2t_k^2 F^{k+1}}{r} + u^{k+1} \leq \frac{2t_{k-1}^2 F^k}{r} + u^k + \frac{2\tau t_k^2 \delta}{r} + 2t_k \mathbb{E}\langle \tilde{v}^{k+1} - v^{k+1}, t_{k-1} v^k - (t_{k-1} - 1)v^{k-1} - v^* \rangle \leq \frac{2t_k^2 F^k}{r} + u^k + \frac{2\tau t_k^2 \delta}{r} + t_{k-1}^2 R^2,
\]  \phantom{.} (A.30)

where we used

\[
2t_k \mathbb{E}\langle \tilde{v}^{k+1} - v^{k+1}, t_{k-1} v^k - (t_{k-1} - 1)v^{k-1} - v^* \rangle \leq t_k^2 \mathbb{E}\|v^{k+1} - \tilde{v}^{k+1}\|_2^2 + \mathbb{E}\|t_{k-1} v^k - (t_{k-1} - 1)v^{k-1} - v^*\|_2^2 \leq 2t_k^2 \delta/r + t_{k-1}^2 R^2.
\]
Denoting
\[ \xi_k := \frac{2t_{k-1}^2 F^k}{r} + u^k, \]
then, we have
\[ \xi_{k+1} \leq \xi_0 + \left( \frac{2\tau \delta}{r} + R^2 \right) \sum_{i=1}^{k} t_i^2 = O(k^3). \]  
(A.31)

With the fact, \( \xi_k := \frac{2t_{k-1}^2 F^k}{r} \geq \Omega(k^2)F^k \), we then proved the result.

**A.2 NAG with \( \delta \)-Inexact Oracle & Experimental Settings in Section 3.1**

In [25], the authors define the \( \delta \)-inexact gradient oracle for convex smooth optimization as follows:

**Definition 1 (\( \delta \)-Inexact Oracle).** [25] For a convex \( L \)-smooth function \( f : \mathbb{R}^d \rightarrow \mathbb{R} \). For all \( w \in \mathbb{R}^d \) and exact first-order oracle returns a pair \((f(w), \nabla f(w)) \) \( \in \mathbb{R} \times \mathbb{R}^d \) so that for all \( v \in \mathbb{R}^d \) we have
\[
0 \leq f(v) - (f(w) + \langle \nabla f(w), v - w \rangle) \leq \frac{L}{2} \|v - w\|^2.
\]

A \( \delta \)-inexact oracle returns a pair \((f^\delta(w), \nabla f^\delta(w)) \) \( \in \mathbb{R} \times \mathbb{R}^d \) so that all \( v \in \mathbb{R}^d \) we have
\[
0 \leq f(v) - (f^\delta(w) + \langle \nabla f^\delta(w), v - w \rangle) \leq \frac{L}{2} \|v - w\|^2 + \delta.
\]

We have the following convergence results of GD and NAG under a \( \delta \)-Inexact Oracle for convex smooth optimization.

**Theorem 4.** [25]* Consider
\[
\min f(w), \ w \in \mathbb{R}^d,
\]

*We adopt the result from [46].
where $f(w)$ is convex and $L$-smooth with $w^*$ being the minimum. Given access to $\delta$-inexact oracle, GD with step size $1/L$ returns a point $w^k$ after $k$ steps so that

$$f(w^k) - f(w^*) = O\left(\frac{L}{k}\right) + \delta.$$  

On the other hand, NAG, with step size $1/L$ returns

$$f(w^k) - f(w^*) = O\left(\frac{L}{k^2}\right) + O(k\delta).$$

Theorem 4 says that NAG may not robust to a $\delta$-inexact gradient. In the following, we will study the numerical behavior of a variety of first-order algorithms for convex smooth optimizations with the following different inexact gradients.

**Constant Variance Gaussian Noise:** We consider the inexact oracle where the true gradient is contaminated with a Gaussian noise $N(0, 0.001^2)$. We run 50K iterations of different algorithms. For SRNAG, we restart after every 200 iterations. Fig. 2.1 (b) shows the iteration vs. optimal gap, $f(x^k) - f(x^*)$, with $x^*$ being the minimum. NAG with the inexact gradient due to constant variance noise does not converge. GD performs almost the same as ARNAG asymptotically, because ARNAG restarts too often and almost degenerates into GD. GD with constant momentum outperforms the three schemes above, and SRNAG slightly outperforms GD with constant momentum.

**Decaying Variance Gaussian Noise:** Again, consider minimizing (2.5) with the same experimental setting as before except that $\nabla f(x)$ is now contaminated with a decaying Gaussian noise $N(0, \left(\frac{0.1}{[t/100]+1}\right)^2)$. For SRNAG, we restart every 200 iterations in the first 10k iterations, and restart every 400 iterations in the remaining 40K iterations. Fig. 2.1 (c) shows the iteration vs. optimal gap by different schemes. ARNAG still performs almost the same as GD. The path of NAG is oscillatory. GD
with constant momentum again outperforms the previous three schemes. Here SRNAG significantly outperforms all the other schemes.

**Logistic Regression for MNIST Classification:** We apply the above schemes with stochastic gradient to train a logistic regression model for MNIST classification [47]. We consider five different schemes, namely, SGD, SGD + (constant) momentum, NASGD, ASGD, and SRSGD. In ARSGD, we perform restart based on the loss value of the mini-batch training data. In SRSGD, we restart the NAG momentum after every 10 iterations. We train the logistic regression model with a $\ell_2$ weight decay of $10^{-4}$ by running 20 epochs using different schemes with batch size of 128. The step sizes for all the schemes are set to 0.01. Fig. 2.2 (a) plots the training loss vs. iteration. In this case, NASGD does not converge, and SGD with momentum does not speed up SGD. ARSGD’s performance is on par with SGD’s. Again, SRSGD gives the best performance with the smallest training loss among these five schemes.

### A.3 Convergence of SRSGD

We prove the convergence of Nesterov accelerated SGD with scheduled restart, i.e., the convergence of SRSGD. We denote that $\theta^k := \frac{t_k - 1}{t_k + 1}$ in the Nesterov iteration and $\hat{\theta}^k$ is its use in the restart version, i.e., SRSGD. For any restart frequency $F$ (positive integer), we have $\hat{\theta}^k = \theta^{k-\lfloor k/F \rfloor F}$. In the restart version, we can see that

$$\hat{\theta}^k \leq \theta^F =: \bar{\theta} < 1.$$  

**Lemma 6.** Let the constant satisfies $r > L$ and the sequence $\{v^k\}_{k \geq 0}$ be generated by the SRSGD with restart frequency $F$ (any positive integer), we have

$$\sum_{i=1}^{k} \|v^i - v^{i-1}\|_2^2 \leq \frac{r^2 k R^2}{(1 - \bar{\theta})^2},$$  

(A.32)

where $\bar{\theta} := \theta^F < 1$ and $R := \sup_x \{\|\nabla f(x)\|\}$. 


Proof. It holds that
\[
\|v^{k+1} - w^k\|_2 = \|v^{k+1} - v^k + v^k - w^k\|_2 \tag{A.33}
\]
\[
\geq \|v^{k+1} - v^k\|_2 - \|v^k - w^k\|_2 \\
\geq \|v^{k+1} - v^k\|_2 - \bar{\theta}\|v^k - v^{k-1}\|_2.
\]
Thus,
\[
\|v^{k+1} - v^k\|_2^2 \geq (\|v^{k+1} - v^k\|_2 - \bar{\theta}\|v^k - v^{k-1}\|_2)^2 \tag{A.34}
\]
\[
= \|v^{k+1} - v^k\|_2^2 - 2\bar{\theta}\|v^k - v^{k-1}\|_2\|v^k - v^{k-1}\|_2 + \bar{\theta}^2\|v^k - v^{k-1}\|_2^2 \\
\geq (1 - \bar{\theta})\|v^{k+1} - v^k\|_2^2 - \bar{\theta}(1 - \bar{\theta})\|v^{k+1} - v^k\|_2^2.
\]
Summing (A.34) from \(k = 1\) to \(K\), we get
\[
(1 - \bar{\theta})^2 \sum_{k=1}^{K} \|v^k - v^{k-1}\|_2^2 \leq \sum_{k=1}^{K} \|v^{k+1} - w^k\|_2^2 \leq r^2KR^2. \tag{A.35}
\]

In the following, we denote
\[
\mathcal{A} := \{k \in \mathbb{Z}^+ | \mathbb{E}f(v^k) \geq \mathbb{E}f(v^{k-1})\}.
\]

Theorem 5 (Convergence of SRSBD). (Theorem 2 restate) For any \(L\)-smooth function \(f\), let the constant satisfies \(r > L\) and the sequence \(\{v^k\}_{k \geq 0}\) be generated by the SRSBD with restart frequency \(F\) (any positive integer). Assume that \(\mathcal{A}\) is finite, then we have
\[
\min_{1 \leq k \leq K} \{\mathbb{E}\|\nabla f(w^k)\|_2^2\} = O\left(\frac{1}{r} + \frac{r}{K}\right). \tag{A.36}
\]
Therefore for \(\forall \epsilon > 0\), to get \(\epsilon\) error bound, we just need to set \(r = O\left(\frac{1}{\epsilon}\right)\) and \(K = O\left(\frac{1}{\epsilon^2}\right)\).

Proof. \(L\)-smoothness of \(f\), i.e., Lipschitz gradient continuity, gives us
\[
f(v^{k+1}) \leq f(w^k) + \langle \nabla f(w^k), v^{k+1} - w^k \rangle + \frac{L}{2}\|v^{k+1} - w^k\|_2^2 \tag{A.37}
\]
Taking expectation, we get
\[
\mathbb{E} f(v^{k+1}) \leq \mathbb{E} f(w^k) - r\mathbb{E} \|\nabla f(w^k)\|^2 + \frac{r^2 LR^2}{2}. \tag{A.38}
\]

On the other hand, we have
\[
f(w^k) \leq f(v^k) + \hat{\theta}^k \langle \nabla f(v^k), v^k - v^{k-1} \rangle + \frac{L(\hat{\theta}^k)^2}{2} \|v^k - v^{k-1}\|^2. \tag{A.39}
\]

Then, we have
\[
\mathbb{E} f(v^{k+1}) \leq \mathbb{E} f(v^k) + \hat{\theta}^k \mathbb{E} \langle \nabla f(v^k), v^k - v^{k-1} \rangle
+ \frac{L(\hat{\theta}^k)^2}{2} \mathbb{E} \|v^k - v^{k-1}\|^2 - r\mathbb{E} \|\nabla f(w^k)\|^2 + \frac{r^2 LR^2}{2}. \tag{A.40}
\]

We also have
\[
\hat{\theta}^k \langle \nabla f(v^k), v^k - v^{k-1} \rangle \leq \hat{\theta}^k \left( f(v^k) - f(v^{k-1}) + \frac{L}{2} \|v^k - v^{k-1}\|^2 \right). \tag{A.41}
\]

We then get that
\[
\mathbb{E} f(v^{k+1}) \leq \mathbb{E} f(v^k) + \hat{\theta}^k \left( \mathbb{E} f(v^k) - \mathbb{E} f(v^{k-1}) \right) - r\mathbb{E} \|\nabla f(w^k)\|^2 + A_k, \tag{A.42}
\]

where
\[
A_k := \mathbb{E} \left[ \frac{L}{2} \|v^k - v^{k-1}\|^2 + \frac{L(\hat{\theta}^k)^2}{2} \mathbb{E} \|v^k - v^{k-1}\|^2 + \frac{r^2 LR^2}{2} \right].
\]

Summing the inequality gives us
\[
\mathbb{E} f(v^{K+1}) \leq \mathbb{E} f(v^0) + \hat{\theta} \sum_{k \in A} \left( \mathbb{E} f(v^k) - \mathbb{E} f(v^{k-1}) \right) \tag{A.43}
- r \sum_{k=1}^{K} \mathbb{E} \|\nabla f(w^k)\|^2 + \sum_{k=1}^{K} A_k.
\]

It is easy to see that
\[
\sum_{k \in A} \left( \mathbb{E} f(v^k) - \mathbb{E} f(v^{k-1}) \right) = \bar{R} < +\infty,
\]
Table A.1: Restarting frequencies for CIFAR10 and CIFAR100 experiments.

<table>
<thead>
<tr>
<th>Schedule</th>
<th>CIFAR10</th>
<th>CIFAR100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear schedule</td>
<td>$F_1 = 30, F_2 = 60, F_3 = 90, F_4 = 120$ (r = 2)</td>
<td>$F_1 = 50, F_2 = 100, F_3 = 150, F_4 = 200$ (r = 2)</td>
</tr>
<tr>
<td>Exponential</td>
<td>$F_1 = 40, F_2 = 50, F_3 = 63, F_4 = 75$ (r = 1.0)</td>
<td>$F_1 = 45, F_2 = 68, F_3 = 101, F_4 = 152$ (r = 1.50)</td>
</tr>
</tbody>
</table>

for the finiteness of $A$, and

$$\sum_{k=1}^{K} A_k = O(r^2 K).$$

\[\square\]

A.4 Datasets and Implementation Details

A.4.1 CIFAR

The CIFAR10 and CIFAR100 datasets [100] consist of 50K training images and 10K test images from 10 and 100 classes, respectively. Both training and test data are color images of size $32 \times 32$. We run our CIFAR experiments on Pre-ResNet-110, 290, 470, 650, and 1001 with 5 different seeds [1]. We train each model for 200 epochs with batch size of 128 and initial learning rate of 0.1, which is decayed by a factor of 10 at the 80th, 120th, and 160th epoch. The weight decay rate is $5 \times 10^{-5}$ and the momentum for the SGD baseline is 0.9. Random cropping and random horizontal flipping are applied to training data. Our code is modified based on the Pytorch classification project [101],‡ which was also used by Liu et al. [19]. We provide the restarting frequencies for the exponential and linear scheme for CIFAR10 and CIFAR100 in Table A.1 below. Using the same notation as in the main text, we denote $F_i$ as the restarting frequency at the $i$-th learning rate.

‡Implementation available at https://github.com/bearpaw/pytorch-classification
Table A.2: Restarting frequencies for ImageNet experiments.

| Linear schedule | $F_1 = 40$, $F_2 = 80$, $F_3$: linearly decayed from 80 to 1 in the last 30 epochs |

A.4.2 ImageNet

The ImageNet dataset contains roughly 1.28 million training color images and 50K validation color images from 1000 classes [52]. We run our ImageNet experiments on ResNet-50, 101, 152, and 200 with 5 different seeds. Following [102, 1], we train each model for 90 epochs with a batch size of 256 and decrease the learning rate by a factor of 10 at the 30th and 60th epoch. The initial learning rate is 0.1, the momentum is 0.9, and the weight decay rate is $1 \times 10^{-5}$. Random $224 \times 224$ cropping and random horizontal flipping are applied to training data. We use the official Pytorch ResNet implementation [48],‡ and run our experiments on 8 Nvidia V100 GPUs. We report single-crop top-1 and top-5 errors of our models. In our experiments, we set $F_1 = 40$ at the 1st learning rate, $F_2 = 80$ at the 2nd learning rate, and $F_3$ is linearly decayed from 80 to 1 at the 3rd learning rate (see Table A.2).

A.4.3 Training ImageNet in Fewer Number of Epochs:

Table A.3 contains the learning rate and restarting frequency schedule for our experiments on training ImageNet in fewer number of epochs, i.e. the reported results in Table 2.5 in the main text. Other settings are the same as in the full-training ImageNet experiments described in Section A.4.2 above.

‡Implementation available at https://github.com/pytorch/examples/tree/master/imagenet
Table A.3: Learning rate and restarting frequency schedule for ImageNet short training, i.e. Table 2.5 in the main text.

<table>
<thead>
<tr>
<th>Model</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ResNet-50</td>
<td>Decrease the learning rate by a factor of 10 at the 30th and 56th epoch. Train for a total of 80 epochs. ( F_1 = 60, F_2 = 105, F_3 ): linearly decayed from 105 to 1 in the last 24 epochs</td>
</tr>
<tr>
<td>ResNet-101</td>
<td>Decrease the learning rate by a factor of 10 at the 30th and 56th epoch. Train for a total of 80 epochs. ( F_1 = 40, F_2 = 80, F_3 ): linearly decayed from 80 to 1 in the last 24 epochs</td>
</tr>
<tr>
<td>ResNet-152</td>
<td>Decrease the learning rate by a factor of 10 at the 30th and 51th epoch. Train for a total of 75 epochs. ( F_1 = 40, F_2 = 80, F_3 ): linearly decayed from 80 to 1 in the last 24 epochs</td>
</tr>
<tr>
<td>ResNet-200</td>
<td>Decrease the learning rate by a factor of 10 at the 30th and 46th epoch. Train for a total of 60 epochs. ( F_1 = 40, F_2 = 80, F_3 ): linearly decayed from 80 to 1 in the last 24 epochs</td>
</tr>
</tbody>
</table>

Additional Implementation Details: Implementation details for the ablation study of error rate vs. reduction in epochs and the ablation study of impact of restarting frequency are provided in Section A.6 and A.7 below.

A.5 SRSGD vs. SGD and SGD + NM on ImageNet Classification and Other Tasks

A.5.1 Comparing with SGD with Nesterov Momentum on ImageNet Classification

In this section, we compare SRSGD with SGD with Nesterov constant momentum (SGD + NM) in training ResNets for ImageNet classification. All hyper-parameters of SGD with constant Nesterov momentum used in our experiments are the same as those of SGD described in section A.4.2. We list the results in Table A.4. Again, SRSGD remarkably outperforms SGD + NM in training ResNets for ImageNet classification,
Table A.4: Single crop validation errors (%) on ImageNet of ResNets trained with SGD + NM and SRSGD. We report the results of SRSGD with the increasing restarting frequency in the first two learning rates. In the last learning rate, the restarting frequency is linearly decreased from 70 to 1. For baseline results, we also include the reported single-crop validation errors [2] (in parentheses).

<table>
<thead>
<tr>
<th>Network</th>
<th># Params</th>
<th>SGD + NM</th>
<th>SRSGD</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>top-1</td>
<td>top-5</td>
<td>top-1</td>
</tr>
<tr>
<td>ResNet-50</td>
<td>25.56M</td>
<td>24.27 ± 0.07</td>
<td>7.17 ± 0.07</td>
<td>23.85 ± 0.09</td>
</tr>
<tr>
<td>ResNet-101</td>
<td>44.55M</td>
<td>22.32 ± 0.05</td>
<td>6.18 ± 0.05</td>
<td>22.06 ± 0.10</td>
</tr>
<tr>
<td>ResNet-152</td>
<td>60.19M</td>
<td>21.77 ± 0.14</td>
<td>5.86 ± 0.09</td>
<td>21.46 ± 0.07</td>
</tr>
<tr>
<td>ResNet-200</td>
<td>64.67M</td>
<td>21.98 ± 0.22</td>
<td>5.99 ± 0.20</td>
<td>20.93 ± 0.13</td>
</tr>
</tbody>
</table>

and as the network goes deeper the improvement becomes more significant.

A.5.2 Long Short-Term Memory (LSTM) Training for Pixel-by-Pixel MNIST

In this task, we examine the advantage of SRSGD over SGD and SGD with Nesterov Momentum in training recurrent neural networks. In our experiments, we use an LSTM with different numbers of hidden units (128, 256, and 512) to classify samples from the well-known MNIST dataset [47]. We follow the implementation of [70] and feed each pixel of the image into the RNN sequentially. In addition, we choose a random permutation of $28 \times 28 = 784$ elements at the beginning of the experiment. This fixed permutation is applied to training and testing sequences. This task is known as permuted MNIST classification, which has become standard to measure the performance of RNNs and their ability to capture long term dependencies.

Implementation and Training Details: For the LSTM model, we initialize the forget bias to 1 and other biases to 0. All weights matrices are initialized orthogonally except for the hidden-to-hidden weight matrices, which are initialized to be identity matrices. We train each model for 350 epochs with the initial learning rate of 0.01.
Table A.5 : Test errors (%) on Permuted MNIST of trained with SGD, SGD + NM and SRSGD. The LSTM model has 128 hidden units. In all experiments, we use the initial learning rate of 0.01, which is reduced by a factor of 10 at epoch 200 and 300. All models are trained for 350 epochs. The momentum for SGD and SGD + NM is set to 0.9. The restart schedule in SRSGD is set to 90, 30, and 90.

<table>
<thead>
<tr>
<th>Network</th>
<th>No. Hidden Units</th>
<th>SGD</th>
<th>SGD + NM</th>
<th>SRSGD</th>
<th>Improvement over SGD/SGD + NM</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSTM</td>
<td>128</td>
<td>10.10 ± 0.57</td>
<td>9.75 ± 0.69</td>
<td><strong>8.61 ± 0.30</strong></td>
<td>1.49/1.14</td>
</tr>
<tr>
<td>LSTM</td>
<td>256</td>
<td>10.42 ± 0.63</td>
<td>10.09 ± 0.61</td>
<td><strong>9.03 ± 0.23</strong></td>
<td>1.39/1.06</td>
</tr>
<tr>
<td>LSTM</td>
<td>512</td>
<td>10.04 ± 0.35</td>
<td>9.55 ± 1.09</td>
<td><strong>8.49 ± 1.59</strong></td>
<td>1.55/1.06</td>
</tr>
</tbody>
</table>

The learning rate was reduced by a factor of 10 at epoch 200 and 300. The momentum is set to 0.9 for SGD with standard and Nesterov constant momentum. The restart schedule for SRSGD is set to 90, 30, 90. The restart schedule changes at epoch 200 and 300. In all experiments, we use batch size 128 and the gradients are clipped so that their L2 norm are at most 1. Our code is based on the code from the exponential RNN’s Github.

**Results:** Our experiments corroborate the superiority of SRSGD over the two baselines. SRSGD yields much smaller test error and converges faster than SGD with standard and Nesterov constant momentum across all settings with different number of LSTM hidden units. We summarize our results in Table A.5 and Figure A.1.

### A.5.3 Wasserstein Generative Adversarial Networks (WGAN) Training on MNIST

We investigate the advantage of SRSGD over SGD with standard and Nesterov momentum in training deep generative models. In our experiments, we train a WGAN with gradient penalty [51] on MNIST. We evaluate our models using the discriminator’s

---

§Implementation available at [https://github.com/Lezcano/expRNN](https://github.com/Lezcano/expRNN)
Figure A.1: Training loss vs. training iterations of LSTM trained with SGD (red), SGD + NM (green), and SRSGD (blue) for PMNIST classification tasks.

loss, i.e. the Earth Moving distance estimate, since in WGAN lower discriminator loss and better sample quality are correlated [50].

Implementation and Training Details: The detailed implementations of our generator and discriminator are given below. For the generator, we set latent_dim to 100 and d to 32. For the discriminator, we set d to 32. We train each model for 350 epochs with the initial learning rate of 0.01. The learning rate was reduced by a factor of 10 at epoch 200 and 300. The momentum is set to 0.9 for SGD with standard and Nesterov constant momentum. The restart schedule for SRSGD is set to 60, 120, 180. The restart schedule changes at epoch 200 and 300. In all experiments, we use batch size 64. Our code is based on the code from the Pytorch WGAN-GP Github.★

```python
import torch
import torch.nn as nn

class Generator(nn.Module):
    def __init__(self, latent_dim, d=32):
```

★Implementation available at https://github.com/arturml/pytorch-wgan-gp
super() \_\_init\_\_()
self.net = nn.Sequential(
    nn.ConvTranspose2d(latent_dim, d \times 8, 4, 1, 0),
    nn.BatchNorm2d(d \times 8),
    nn.ReLU(True),

    nn.ConvTranspose2d(d \times 8, d \times 4, 4, 2, 1),
    nn.BatchNorm2d(d \times 4),
    nn.ReLU(True),

    nn.ConvTranspose2d(d \times 4, d \times 2, 4, 2, 1),
    nn.BatchNorm2d(d \times 2),
    nn.ReLU(True),

    nn.ConvTranspose2d(d \times 2, 1, 4, 2, 1),
    nn.Tanh()
)
def forward(self, x):
    return self.net(x)

class Discriminator(nn.Module):
    def \_\_init\_\_(self, d=32):
        super() \_\_init\_\_(self, d=32):
        self.net = nn.Sequential(
            nn.Conv2d(1, d, 4, 2, 1),
            nn.InstanceNorm2d(d),
            nn.LeakyReLU(0.2),

            nn.Conv2d(d, d \times 2, 4, 2, 1),
            nn.InstanceNorm2d(d \times 2),
nn.LeakyReLU(0.2),

nn.Conv2d(d * 2, d * 4, 4, 2, 1),
nn.InstanceNorm2d(d * 4),
nn.LeakyReLU(0.2),

nn.Conv2d(d * 4, 1, 4, 1, 0),
)

def forward(self, x):
    outputs = self.net(x)
    return outputs.squeeze()

Results: Our SRSGD is still better than both the baselines. SRSGD achieves smaller discriminator loss, i.e. Earth Moving distance estimate, and converges faster than SGD with standard and Nesterov constant momentum. We summarize our results in Table A.6 and Figure A.2. We also demonstrate the digits generated by the trained WGAN in Figure A.3. By visually evaluation, we observe that samples generated by the WGAN trained with SRSGD look slightly better than those generated by the WGAN trained with SGD with standard and Nesterov constant momentum.

Table A.6: Discriminator loss (i.e. Earth Moving distance estimate) of the WGAN with gradient penalty trained on MNIST with SGD, SGD + NM and SRSGD. In all experiments, we use the initial learning rate of 0.01, which is reduced by a factor of 10 at epoch 200 and 300. All models are trained for 350 epochs. The momentum for SGD and SGD + NM is set to 0.9. The restart schedule in SRSGD is set to 60, 120, and 180.

<table>
<thead>
<tr>
<th>Task</th>
<th>SGD</th>
<th>SGD + NM</th>
<th>SRSGD</th>
<th>Improvement over SGD/SGD + NM</th>
</tr>
</thead>
<tbody>
<tr>
<td>MNIST</td>
<td>0.71 ± 0.10</td>
<td>0.58 ± 0.03</td>
<td>0.44 ± 0.06</td>
<td>0.27/0.14</td>
</tr>
</tbody>
</table>
Figure A.2: Earth Moving distance estimate (i.e. discriminator loss) vs. training epochs of WGAN with gradient penalty trained with SGD (red), SGD + NM (green), and SRSGD (blue) on MNIST.

Figure A.3: MNIST digits generated by WGAN trained with gradient penalty by SGD (left), SGD + NM (middle), and SRSGD (right).

A.6 Error Rate vs. Reduction in Training Epochs

A.6.1 Implementation Details

CIFAR10 (Figure 2.3, left, in the main text) and CIFAR100 (Figure A.5 in this Appendix): Except for learning rate schedule, we use the same setting
Table A.7: Learning rate (LR) schedule for the ablation study of error rate vs. reduction in training epochs for CIFAR10 experiments, i.e. Figure A.5 (left) in the main text and for CIFAR100 experiments, i.e. Figure A.5 in this Appendix.

<table>
<thead>
<tr>
<th>#of Epoch Reduction</th>
<th>LR Schedule</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Decrease the LR by a factor of 10 at the 80th, 120th and 160th epoch. Train for a total of 200 epochs.</td>
</tr>
<tr>
<td>15</td>
<td>Decrease the LR by a factor of 10 at the 80th, 115th and 150th epoch. Train for a total of 185 epochs.</td>
</tr>
<tr>
<td>30</td>
<td>Decrease the LR by a factor of 10 at the 80th, 110th and 140th epoch. Train for a total of 170 epochs.</td>
</tr>
<tr>
<td>45</td>
<td>Decrease the LR by a factor of 10 at the 80th, 105th and 130th epoch. Train for a total of 155 epochs.</td>
</tr>
<tr>
<td>60</td>
<td>Decrease the LR by a factor of 10 at the 80th, 100th and 120th epoch. Train for a total of 140 epochs.</td>
</tr>
<tr>
<td>75</td>
<td>Decrease the LR by a factor of 10 at the 80th, 95th and 110th epoch. Train for a total of 125 epochs.</td>
</tr>
<tr>
<td>90</td>
<td>Decrease the LR by a factor of 10 at the 80th, 90th and 100th epoch. Train for a total of 110 epochs.</td>
</tr>
</tbody>
</table>

described in Section A.4.1 above and Section 4.1 in the main text. Table A.7 contains the learning rate schedule for each number of epoch reduction in Figure 2.3 (left) in the main text and Figure A.5 below.

**ImageNet (Figure 2.3, right, in the main text):** Except for the total number of training epochs, other settings are similar to experiments for training ImageNet in fewer number of epochs described in Section A.4.3. In particular, the learning rate and restarting frequency schedule still follow those in Table A.3 above. We examine different numbers of training epochs: 90 (0 epoch reduction), 80 (10 epochs reduction), 75 (15 epochs reduction), 70 (20 epochs reduction), 65 (25 epochs reduction), and 60 (30 epochs reduction).

### A.6.2 Additional Experimental Results

Figure A.4 shows error rate vs. reduction in epochs for all models trained on CIFAR10 and ImageNet. It is a more complete version of Figure 2.3 in the main text. Table A.8
Figure A.4: Test error vs. number of training epochs. Dashed lines are test errors of SGD trained in 200 epochs for CIFAR10 and 90 epochs for ImageNet. For CIFAR, SRSGD with fewer epochs achieves comparable results to SRSGD with 200 epochs. For ImageNet, training with less epochs slightly decreases the performance of SRSGD but still achieves comparable results to 200-epoch SGD.

Table A.8: Test error vs. number of training epochs for CIFAR10.

<table>
<thead>
<tr>
<th>Network</th>
<th>110 (90 less)</th>
<th>125 (75 less)</th>
<th>140 (60 less)</th>
<th>155 (45 less)</th>
<th>170 (30 less)</th>
<th>185 (15 less)</th>
<th>200 (full trainings)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre-ResNet-110</td>
<td>5.37 ± 0.11</td>
<td>5.27 ± 0.17</td>
<td>5.15 ± 0.09</td>
<td>5.09 ± 0.14</td>
<td>4.96 ± 0.14</td>
<td>4.96 ± 0.13</td>
<td>4.93 ± 0.13</td>
</tr>
<tr>
<td>Pre-ResNet-290</td>
<td>4.80 ± 0.14</td>
<td>4.71 ± 0.13</td>
<td>4.58 ± 0.11</td>
<td>4.45 ± 0.09</td>
<td>4.43 ± 0.09</td>
<td>4.44 ± 0.11</td>
<td>4.37 ± 0.15</td>
</tr>
<tr>
<td>Pre-ResNet-470</td>
<td>4.52 ± 0.16</td>
<td>4.43 ± 0.12</td>
<td>4.29 ± 0.11</td>
<td>4.33 ± 0.07</td>
<td>4.23 ± 0.12</td>
<td><strong>4.18 ± 0.09</strong></td>
<td><strong>4.18 ± 0.09</strong></td>
</tr>
<tr>
<td>Pre-ResNet-650</td>
<td>4.35 ± 0.10</td>
<td>4.24 ± 0.05</td>
<td>4.22 ± 0.15</td>
<td>4.10 ± 0.15</td>
<td>4.12 ± 0.14</td>
<td>4.02 ± 0.05</td>
<td><strong>4.00 ± 0.07</strong></td>
</tr>
<tr>
<td>Pre-ResNet-1001</td>
<td><strong>4.23 ± 0.19</strong></td>
<td><strong>4.13 ± 0.12</strong></td>
<td><strong>4.08 ± 0.15</strong></td>
<td><strong>4.10 ± 0.09</strong></td>
<td><strong>3.93 ± 0.11</strong></td>
<td><strong>4.06 ± 0.14</strong></td>
<td><strong>3.87 ± 0.07</strong></td>
</tr>
</tbody>
</table>

and Table A.9 provide detailed test errors vs. number of training epoch reduction reported in Figure 2.3 and Figure A.4. We also conduct an additional ablation study of error rate vs. reduction in epochs for CIFAR100 and include the results in Figure A.5 and Table A.10 below.
Table A.9: Top 1 single crop validation error vs. number of training epochs for ImageNet.

<table>
<thead>
<tr>
<th>Network</th>
<th>60 (30 less)</th>
<th>65 (25 less)</th>
<th>70 (20 less)</th>
<th>75 (15 less)</th>
<th>80 (10 less)</th>
<th>90 (full trainings)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ResNet-50</td>
<td>25.42 ± 0.42</td>
<td>25.02 ± 0.15</td>
<td>24.77 ± 0.14</td>
<td>24.38 ± 0.01</td>
<td>24.30 ± 0.21</td>
<td>23.85 ± 0.50</td>
</tr>
<tr>
<td>ResNet-101</td>
<td>23.11 ± 0.10</td>
<td>22.79 ± 0.01</td>
<td>22.71 ± 0.21</td>
<td>22.56 ± 0.10</td>
<td>22.44 ± 0.03</td>
<td>22.06 ± 0.10</td>
</tr>
<tr>
<td>ResNet-152</td>
<td>22.28 ± 0.20</td>
<td>22.12 ± 0.04</td>
<td>21.97 ± 0.04</td>
<td>21.79 ± 0.07</td>
<td>21.70 ± 0.07</td>
<td>21.46 ± 0.07</td>
</tr>
<tr>
<td>ResNet-200</td>
<td>21.92 ± 0.17</td>
<td>21.69 ± 0.20</td>
<td>21.64 ± 0.03</td>
<td>21.45 ± 0.06</td>
<td>21.30 ± 0.03</td>
<td>20.93 ± 0.13</td>
</tr>
</tbody>
</table>

Table A.10: Test error vs. number of training epochs for CIFAR100.

<table>
<thead>
<tr>
<th>Network</th>
<th>110 (90 less)</th>
<th>125 (75 less)</th>
<th>140 (60 less)</th>
<th>155 (45 less)</th>
<th>170 (30 less)</th>
<th>185 (15 less)</th>
<th>200 (full trainings)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre-ResNet-110</td>
<td>24.06 ± 0.26</td>
<td>23.82 ± 0.24</td>
<td>23.82 ± 0.28</td>
<td>23.58 ± 0.18</td>
<td>23.69 ± 0.21</td>
<td>23.73 ± 0.34</td>
<td>23.49 ± 0.23</td>
</tr>
<tr>
<td>Pre-ResNet-290</td>
<td>21.96 ± 0.45</td>
<td>21.77 ± 0.21</td>
<td>21.67 ± 0.37</td>
<td>21.56 ± 0.33</td>
<td>21.38 ± 0.44</td>
<td>21.47 ± 0.32</td>
<td>21.49 ± 0.27</td>
</tr>
<tr>
<td>Pre-ResNet-470</td>
<td>21.35 ± 0.17</td>
<td>21.25 ± 0.17</td>
<td>21.21 ± 0.18</td>
<td>21.09 ± 0.28</td>
<td>20.87 ± 0.28</td>
<td>20.81 ± 0.32</td>
<td>20.71 ± 0.32</td>
</tr>
<tr>
<td>Pre-ResNet-650</td>
<td>21.18 ± 0.27</td>
<td>20.95 ± 0.13</td>
<td>20.77 ± 0.31</td>
<td>20.61 ± 0.19</td>
<td>20.57 ± 0.13</td>
<td>20.47 ± 0.07</td>
<td>20.36 ± 0.25</td>
</tr>
<tr>
<td>Pre-ResNet-1001</td>
<td>20.27 ± 0.17</td>
<td>20.03 ± 0.13</td>
<td>20.05 ± 0.22</td>
<td>19.74 ± 0.18</td>
<td>19.71 ± 0.22</td>
<td>19.67 ± 0.22</td>
<td>19.75 ± 0.11</td>
</tr>
</tbody>
</table>

A.7 Impact of Restarting Frequency for ImageNet and CIFAR100

A.7.1 Implementation Details

For the CIFAR10 experiments on Pre-ResNet-290 in Figure 2.4 in the main text, as well as the CIFAR100 and ImageNet experiments in Figure A.7 and A.8 in this Appendix, we vary the initial restarting frequency $F_1$. Other settings are the same as described in Section A.4 above.
Figure A.5: Test error vs. number of epoch reduction in CIFAR100 training. The dashed lines are test errors of the SGD baseline. For CIFAR100, SRSGD training with fewer epochs can achieve comparable results to SRSGD training with full 200 epochs. In some cases, such as with Pre-ResNet-290 and 1001, SRSGD training with fewer epochs achieves even better results than SRSGD training with full 200 epochs.

A.7.2 Additional Experimental Results

To complete our study on the impact of restarting frequency in Section 5.2 in the main text, we examine the case of CIFAR100 and ImageNet in this section. We summarize our results in Figure A.7 and A.8 below. Also, Figure A.6 is a more detailed version of Figure 2.4 in the main text.

A.8 Full Training with Less Epochs at the Intermediate Learning Rates

We explore SRSGD full training (200 epochs on CIFAR and 90 epochs on ImageNet) with less number of epochs at the intermediate learning rates and report the results in Table A.11, A.12, A.13 and Figure A.9, A.10, A.11 below. The settings and implementation details here are similar to those in Section A.6 of this Appendix, but using all 200 epochs for CIFAR experiments and 90 epochs for ImageNet experiments.
Figure A.6: Training loss (left) and test error (right) of Pre-ResNet-290 trained on CIFAR10 with different initial restarting frequencies $F_1$ (linear schedule). SRSGD with small $F_1$ approximates SGD without momentum, while SRSGD with large $F_1$ approximates NASGD.

Figure A.7: Training loss and test error of Pre-ResNet-290 trained on CIFAR100 with different initial restarting frequencies $F_1$ (linear schedule). SRSGD with small $F_1$ approximates SGD without momentum, while SRSGD with large $F_1$ approximates NASGD.

A.9 Visualization of SRSGD’s trajectory

Here we visualize the training trajectory through bad minima of SRSGD, SGD with constant momentum, and SGD. In particular, we train a neural net classifier on a swiss roll data as in [103] and find bad minima along its training. Each red dot in Figure A.12 represents the trained model after each 10 epochs in the training. From each red dot, we search for nearby bad local minima, which are the blue dots. Those bad local minima achieve good training error but bad test error. We plots the trained
Figure A.8: Training loss and test error of ResNet-101 trained on ImageNet with different initial restarting frequencies $F_1$. We use linear schedule and linearly decrease the restarting frequency to 1 at the last learning rate. SRSGD with small $F_1$ approximates SGD without momentum, while SRSGD with large $F_1$ approximates NASGD.

Figure A.9: Test error when using new learning rate schedules with less training epochs at the 2nd and 3rd learning rate for CIFAR10. We still train in full 200 epochs in this experiment. On the x-axis, 10, for example, means we reduce the number of training epochs by 10 at each intermediate learning rate, i.e. the 2nd and 3rd learning rate. The dashed lines are test errors of the SGD baseline.

models and bad local minima using PCA [104] and t-SNE [105] embedding. The blue color bar is for the test accuracy of bad local minima; the red color bar is for the number of training epochs.
Table A.11: Test error when using new learning rate schedules with less training epochs at the 2nd and 3rd learning rate for CIFAR10. We still train in full 200 epochs in this experiment. In the table, 80-90-100, for example, means we reduce the learning rate by factor of 10 at the 80th, 90th, and 100th epoch.

<table>
<thead>
<tr>
<th>Network</th>
<th>80 - 90 - 100</th>
<th>80 - 95 - 110</th>
<th>80 - 100 - 120</th>
<th>80 - 105 - 130</th>
<th>80 - 110 - 140</th>
<th>80 - 115 - 150</th>
<th>80 - 120 - 160</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre-ResNet-110</td>
<td>5.32 ± 0.14</td>
<td>5.24 ± 0.17</td>
<td>5.11 ± 0.13</td>
<td>5.04 ± 0.15</td>
<td><strong>4.92 ± 0.15</strong></td>
<td>4.95 ± 0.12</td>
<td>4.93 ± 0.13</td>
</tr>
<tr>
<td>Pre-ResNet-290</td>
<td>4.73 ± 0.13</td>
<td>4.67 ± 0.12</td>
<td>4.53 ± 0.10</td>
<td>4.40 ± 0.11</td>
<td>4.42 ± 0.09</td>
<td>4.42 ± 0.10</td>
<td><strong>4.37 ± 0.15</strong></td>
</tr>
<tr>
<td>Pre-ResNet-470</td>
<td>4.48 ± 0.16</td>
<td>4.34 ± 0.10</td>
<td>4.25 ± 0.12</td>
<td>4.28 ± 0.10</td>
<td>4.19 ± 0.10</td>
<td><strong>4.14 ± 0.07</strong></td>
<td>4.18 ± 0.09</td>
</tr>
<tr>
<td>Pre-ResNet-650</td>
<td>4.25 ± 0.13</td>
<td>4.12 ± 0.06</td>
<td>4.13 ± 0.09</td>
<td>4.03 ± 0.11</td>
<td>4.04 ± 0.11</td>
<td>4.04 ± 0.04</td>
<td><strong>4.00 ± 0.07</strong></td>
</tr>
<tr>
<td>Pre-ResNet-1001</td>
<td>4.14 ± 0.18</td>
<td>4.06 ± 0.12</td>
<td>4.04 ± 0.15</td>
<td>4.08 ± 0.09</td>
<td>3.92 ± 0.13</td>
<td>4.05 ± 0.14</td>
<td><strong>3.87 ± 0.07</strong></td>
</tr>
</tbody>
</table>

Table A.12: Test error when using new learning rate schedules with less training epochs at the 2nd and 3rd learning rate for CIFAR100. We still train in full 200 epochs in this experiment. In the table, 80-90-100, for example, means we reduce the learning rate by factor of 10 at the 80th, 90th, and 100th epoch.

<table>
<thead>
<tr>
<th>Network</th>
<th>80 - 90 - 100</th>
<th>80 - 95 - 110</th>
<th>80 - 100 - 120</th>
<th>80 - 105 - 130</th>
<th>80 - 110 - 140</th>
<th>80 - 115 - 150</th>
<th>80 - 120 - 160</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre-ResNet-110</td>
<td>23.65 ± 0.14</td>
<td>23.96 ± 0.26</td>
<td>23.97 ± 0.31</td>
<td>23.65 ± 0.13</td>
<td>23.57 ± 0.36</td>
<td>23.68 ± 0.24</td>
<td><strong>23.49 ± 0.23</strong></td>
</tr>
<tr>
<td>Pre-ResNet-290</td>
<td>21.94 ± 0.44</td>
<td>21.71 ± 0.27</td>
<td>21.55 ± 0.40</td>
<td>21.44 ± 0.31</td>
<td><strong>21.37 ± 0.45</strong></td>
<td>21.47 ± 0.32</td>
<td>21.49 ± 0.27</td>
</tr>
<tr>
<td>Pre-ResNet-470</td>
<td>21.29 ± 0.11</td>
<td>21.21 ± 0.14</td>
<td>21.17 ± 0.18</td>
<td>20.99 ± 0.28</td>
<td>20.81 ± 0.22</td>
<td>20.80 ± 0.31</td>
<td><strong>20.71 ± 0.32</strong></td>
</tr>
<tr>
<td>Pre-ResNet-650</td>
<td>21.11 ± 0.24</td>
<td>20.91 ± 0.17</td>
<td>20.66 ± 0.33</td>
<td>20.52 ± 0.18</td>
<td>20.51 ± 0.16</td>
<td>20.43 ± 0.10</td>
<td><strong>20.36 ± 0.25</strong></td>
</tr>
<tr>
<td>Pre-ResNet-1001</td>
<td>20.21 ± 0.15</td>
<td>20.00 ± 0.11</td>
<td>19.86 ± 0.19</td>
<td><strong>19.55 ± 0.19</strong></td>
<td>19.69 ± 0.21</td>
<td>19.60 ± 0.17</td>
<td>19.75 ± 0.11</td>
</tr>
</tbody>
</table>
Figure A.10 : Test error when using new learning rate schedules with less training epochs at the 2nd and 3rd learning rate for CIFAR100. We still train in full 200 epochs in this experiment. On the x-axis, 10, for example, means we reduce the number of training epochs by 10 at each intermediate learning rate, i.e. the 2nd and 3rd learning rate. The dashed lines are test errors of the SGD baseline.

Table A.13 : Top 1 single crop validation error when using new learning rate schedules with less training epochs at the 2nd learning rate for ImageNet. We still train in full 90 epochs in this experiment. In the table, 30-40, for example, means we reduce the learning rate by factor of 10 at the 30th and 40th epoch.

<table>
<thead>
<tr>
<th>Network</th>
<th>30 - 40</th>
<th>30 - 45</th>
<th>30 - 50</th>
<th>30 - 55</th>
<th>30 - 60</th>
</tr>
</thead>
<tbody>
<tr>
<td>ResNet-50</td>
<td>24.44 ± 0.16</td>
<td>24.06 ± 0.15</td>
<td>24.05 ± 0.09</td>
<td>23.89 ± 0.14</td>
<td>23.85 ± 0.09</td>
</tr>
<tr>
<td>ResNet-101</td>
<td>22.49 ± 0.09</td>
<td>22.51 ± 0.05</td>
<td>22.24 ± 0.01</td>
<td>22.20 ± 0.01</td>
<td>22.06 ± 0.10</td>
</tr>
<tr>
<td>ResNet-152</td>
<td>22.02 ± 0.01</td>
<td>21.84 ± 0.03</td>
<td>21.65 ± 0.14</td>
<td>21.55 ± 0.06</td>
<td>21.46 ± 0.07</td>
</tr>
<tr>
<td>ResNet-200</td>
<td>21.65 ± 0.02</td>
<td>21.27 ± 0.14</td>
<td>21.12 ± 0.02</td>
<td>21.07 ± 0.01</td>
<td>20.93 ± 0.13</td>
</tr>
</tbody>
</table>
Figure A.11: Test error when using new learning rate schedules with less training epochs at the 2nd learning rate for ImageNet. We still train in full 90 epochs in this experiment. On the x-axis, 10, for example, means we reduce the number of training epochs by 10 at the 2nd learning rate. The dashed lines are test errors of the SGD baseline.

Figure A.12: Trajectory through bad minima of SGD, SGD with constant momentum, and SRSGD during the training: we train a neural net classifier and plot the iterates of SGD after each ten epoch (red dots). We also plot locations of nearby “bad” minima with poor generalization (blue dots). We visualize these using PCA and t-SNE embedding. The blue color bar is for the test accuracy of bad local minima while the red color bar is for the number of training epochs. All blue dots for SGD with constant momentum and SRSGD achieve near perfect train accuracy, but with test accuracy below 59%. All blue dots for SGD achieves average train accuracy of 73.11% and with test accuracy also below 59%. The final iterate (yellow star) of SGD, SGD with constant momentum, and SRSGD achieve 73.13%, 99.25%, and 100.0% test accuracy, respectively.
A.10 SRSGD Implementation in Pytorch

```python
import torch

from .optimizer import Optimizer, required

class SRSGD(Optimizer):
    """
    Scheduled Restart SGD.
    Args:
    params (iterable): iterable of parameters to optimize
        or dict defining parameter groups.
    lr (float): learning rate.
    weight_decay (float, optional): weight decay (L2 penalty) (default: 0)
    iter_count (integer): count the iterations mod 200
    Example:
    >>> optimizer = torch.optim.SRSGD(model.parameters(), lr=0.1,
    >>>                                 weight_decay=5e-4, iter_count=1)
    >>> optimizer.zero_grad()
    >>> loss_fn(model(input), target).backward()
    >>> optimizer.step()
    >>> iter_count = optimizer.update_iter()

    Formula:
    \[ v_{t+1} = p_t - lr \times g_t \]
    \[ p_{t+1} = v_{t+1} + \frac{\text{iter\_count}}{\text{iter\_count}+3} \times (v_{t+1} - v_t) \]
    """

    def __init__(self, params, lr=required, weight_decay=0.,
                 iter_count=1, restarting_iter=100):
        if lr is not required and lr < 0.0:
            raise ValueError("Invalid learning rate: \{\}".format(lr))
```

if weight_decay < 0.0:
    raise ValueError("Invalid weight_decay value: {}\".format(weight_decay))

if iter_count < 1:
    raise ValueError("Invalid iter_count: {}\".format(iter_count))

if restarting_iter < 1:
    raise ValueError("Invalid restarting_iter: {}\".format(restarting_iter))

def _setstate__(self, state):
    super(SRSGD, self)._setstate__(state)

def update_iter(self):
    idx = 1
    for group in self.param_groups:
        if idx == 1:
            group[\'iter_count\'] += 1
            if group[\'iter_count\'] >= group[\'restarting_iter\']:
                group[\'iter_count\'] = 1
            idx += 1
    return group[\'iter_count\'], group[\'restarting_iter\']

def step(self, closure=None):
    
    """
Perform a single optimization step.
Arguments: closure (callable, optional): A closure that reevaluates the model and returns the loss.

```python
loss = None
if closure is not None:
    loss = closure()

for group in self.param_groups:
    weight_decay = group['weight_decay']
    momentum = (group['iter_count'] - 1.)/(group['iter_count'] + 2.)
    for p in group['params']:
        if p.grad is None:
            continue
d_p = p.grad.data
if weight_decay != 0:
    d_p.add_(weight_decay, p.data)

param_state = self.state[p]

if 'momentum_buffer' not in param_state:
    buf0 = param_state['momentum_buffer'] = torch.clone(p.data).detach()
else:
    buf0 = param_state['momentum_buffer']

buf1 = p.data - group['lr'] * d_p
p.data = buf1 + momentum*(buf1 - buf0)
param_state['momentum_buffer'] = buf1
```
iter_count, iter_total = self.update_iter()

return loss

A.11 SRSGD Implementation in Keras

import numpy as np
import tensorflow as tf
from keras import backend as K
from keras.optimizers import Optimizer
from keras.legacy import interfaces
if K.backend() == 'tensorflow':
    import tensorflow as tf

class SRSGD(Optimizer):
    """Scheduled Restart Stochastic gradient descent optimizer.
    Includes support for Nesterov momentum and learning rate decay.
    # Arguments
    learning_rate: float >= 0. Learning rate.
    """

def __init__(self, learning_rate=0.01, iter_count=1, restarting_iter = 40, **kwargs):
    learning_rate = kwargs.pop('lr', learning_rate)
    self.initial_decay = kwargs.pop('decay', 0.0)
    super(SRSGD, self).__init__(**kwargs)
    with K.name_scope(self.__class__.__name__):
        self.iterations = K.variable(0, dtype='int64', name='')
iterations)
self.learning_rate = K.variable(learning_rate, name='learning_rate')
self.decay = K.variable(self.initial_decay, name='decay')

# for srgd
self.iter_count = K.variable(iter_count, dtype='int64', name='iter_count')

self.restarting_iter = K.variable(restarting_iter, dtype='int64', name='restarting_iter')
self.nesterov = nesterov

@interfaces.legacy_get_updates_support
@K.symbolic
def get_updates(self, loss, params):
    grads = self.get_gradients(loss, params)
    self.updates = [K.update_add(self.iterations, 1)]

    momentum = (K.cast(self.iter_count, dtype=K.dtype(self.decay)) - 1.) / (K.cast(self.iter_count, dtype=K.dtype(self.decay)) + 2.)

    lr = self.learning_rate
    if self.initial_decay > 0:
        lr = lr * (1. / (1. + self.decay * K.cast(self.iterations, K.dtype(self.decay))))

    # momentum
    shapes = [K.int_shape(p) for p in params]

    moments = [K.variable(value=K.get_value(p), dtype=K.dtype(self.decay)) for p in params]

    return updates
decay), name='moment_' + str(i))
    for (i, p) in enumerate(params):

    self.weights = [self.iterations] + moments + [self.iter_count]

    for p, g, m in zip(params, grads, moments):
        v = p - lr * g
        new_p = v + momentum * (v - m)
        self.updates.append(K.update(m, v))

        # Apply constraints.
        if getattr(p, 'constraint', None) is not None:
            new_p = p.constraint(new_p)

        self.updates.append(K.update(p, new_p))

    condition = K.all(K.less(self.iter_count, self.restarting_iter))
    new_iter_count = K.switch(condition, self.iter_count + 1, self.
                                iter_count - self.restarting_iter + 1)
    self.updates.append(K.update(self.iter_count, new_iter_count))

    return self.updates

def get_config(self):
    config = {'learning_rate': float(K.get_value(self.learning_rate))},
              'decay': float(K.get_value(self.decay)),
              'iter_count': int(K.get_value(self.iter_count)),
              'restarting_iter': int(K.get_value(self.
                                      restarting_iter))}
    base_config = super(SRSGD, self).get_config()
return dict(list(base_config.items()) + list(config.items()))
Appendix B

Additional Details and Results for Momentum Recurrent Neural Networks

B.1 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 $\mu$, step size $s$, scheduled restart $F$, and the coefficient $\beta$ 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 B.2, B.3, B.4, and B.5.

B.1.1 Pixel-by-Pixel MNIST

MNIST dataset [75] 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 $28 \times 28$. As mentioned in Section 3.3.1, 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 [77, 6] to train LSTM models for the MNIST and PMNIST tasks. Gradient norms are clipped to 1 during training, and the smoothing constant $\alpha$ for the RMSProp optimizer is set to 0.9. We provide other details on hyperparameters for the LSTM training on (P)MNIST in Table B.1 (top).

**DTRIV.** We use the best DTRIV models for each (P)MNIST task reported in [6] with Cayley initialization [3]. The gradient norms are clipped to 1 during training. Other hyperparameter details are provided in Table B.1 (bottom).

### B.1.2 TIMIT Speech Dataset

TIMIT speech dataset is a collection of real-world speech recordings [76] 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 [4, 6, 77, 3]. 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 [77, 6] to train LSTM models for the TIMIT tasks. We use the standard Adam optimizer in PyTorch [48] to train the models without using gradient clipping. We provide other details on hyperparameters for the LSTM training on TIMIT in Table B.1 (top).

**DTRIV.** We use the best DTRIV models for each TIMIT task reported in [6] with Henaff initialization [71]. Other hyperparameter details are provided in Table B.1 (bottom).

### B.1.3 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 [72].

**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 [78]. We summarize some important details in Table B.1 (top).

### B.1.4 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 $\|\partial \mathcal{L}/\partial h_t\|_2$ for each time step $t$ at each training iteration, as shown in Figure 3.1. Both MomentumDTRIV and DTRIV models used in this experiment contains one cell of 170 hidden units. The model implementation and training details are similar to those in Section B.1.1 above. Note that DTRIV is also an RNN with additional orthogonality constraint.
Table B.1: Hyperparameters for the Baseline LSTM and DTRIV Training.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Optimizer</th>
<th>Learning Rate</th>
<th>Batch Size</th>
<th>#Epochs</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSTM</td>
<td>RMSProp</td>
<td>0.001</td>
<td>128</td>
<td>150</td>
</tr>
<tr>
<td>PMNIST</td>
<td>RMSProp</td>
<td>0.001</td>
<td>128</td>
<td>150</td>
</tr>
<tr>
<td>TIMIT</td>
<td>Adam</td>
<td>0.0001</td>
<td>32</td>
<td>700</td>
</tr>
<tr>
<td>PTB</td>
<td>SGD</td>
<td>0.0003</td>
<td>128</td>
<td>700</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Size</th>
<th>DTRIV</th>
<th>Optimizer</th>
<th>Learning Rate</th>
<th>Orthogonal</th>
<th>Optimizer</th>
<th>Learning Rate</th>
<th>Size</th>
<th>#Epochs</th>
</tr>
</thead>
<tbody>
<tr>
<td>MNIST</td>
<td>170</td>
<td>1</td>
<td>RMSProp</td>
<td>0.0001</td>
<td>0.0001</td>
<td>128</td>
<td>150</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MNIST</td>
<td>360</td>
<td>∞</td>
<td>RMSProp</td>
<td>0.0007</td>
<td>0.0002</td>
<td>128</td>
<td>150</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MNIST</td>
<td>512</td>
<td>100</td>
<td>RMSProp</td>
<td>0.0005</td>
<td>0.0001</td>
<td>128</td>
<td>150</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PMNIST</td>
<td>170</td>
<td>1</td>
<td>RMSProp</td>
<td>0.0007</td>
<td>0.0005</td>
<td>128</td>
<td>150</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PMNIST</td>
<td>360</td>
<td>∞</td>
<td>RMSProp</td>
<td>0.0007</td>
<td>0.0005</td>
<td>128</td>
<td>150</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PMNIST</td>
<td>512</td>
<td>∞</td>
<td>RMSProp</td>
<td>0.0003</td>
<td>0.0007</td>
<td>128</td>
<td>150</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TIMIT</td>
<td>224</td>
<td>∞</td>
<td>Adam</td>
<td>0.001</td>
<td>0.0002</td>
<td>128</td>
<td>700</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TIMIT</td>
<td>322</td>
<td>∞</td>
<td>Adam</td>
<td>0.001</td>
<td>0.0002</td>
<td>128</td>
<td>700</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### B.2 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 (3.1), and for any given training sample \((x, y)\) with \(x = (x_1, \ldots, x_T)\) being an input sequence of length \(T\) and \(y = (y_1, \ldots, y_T)\) being the sequence of labels \(^*\). Let \(\mathcal{L}_t\) be the loss at the time step \(t\) and the total loss on the whole sequence is

\[
\mathcal{L} = \sum_{t=1}^{T} \mathcal{L}_t.
\]  

\(^*\)Without loss of generality, we consider the sequence to sequence modeling.
Table B.2: Hyperparameters for MomentumLSTM and MomentumDTRIV Training

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Momentum $\mu$</th>
<th>Step Size $s$</th>
<th>Optimizer</th>
<th>Learning Rate</th>
<th>Batch Size</th>
<th>#Epochs</th>
</tr>
</thead>
<tbody>
<tr>
<td>MNIST</td>
<td>0.6</td>
<td>0.6</td>
<td>RMSProp</td>
<td>0.001</td>
<td>128</td>
<td>150</td>
</tr>
<tr>
<td>PMNIST</td>
<td>0.6</td>
<td>1.0</td>
<td>RMSProp</td>
<td>0.001</td>
<td>128</td>
<td>150</td>
</tr>
<tr>
<td>TIMIT</td>
<td>0.3</td>
<td>0.1</td>
<td>Adam</td>
<td>0.0001</td>
<td>32</td>
<td>700</td>
</tr>
<tr>
<td>PTB</td>
<td>0.0</td>
<td>0.6</td>
<td>SGD</td>
<td>30 (initial learning rate)</td>
<td>20</td>
<td>500</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Size</th>
<th>DTRIV</th>
<th>Momentum $\mu$</th>
<th>Step Size $s$</th>
<th>Optimizer</th>
<th>Learning Rate</th>
<th>Orthogonal Rate</th>
<th>Orthogonal Size</th>
<th>Batch</th>
<th>#Epochs</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMNIST</td>
<td>170</td>
<td>$\infty$</td>
<td>0.6</td>
<td>0.9</td>
<td>RMSProp</td>
<td>0.0007</td>
<td>0.00002</td>
<td>128</td>
<td>150</td>
<td></td>
</tr>
<tr>
<td>PMNIST</td>
<td>360</td>
<td>$\infty$</td>
<td>0.3</td>
<td>0.3</td>
<td>RMSProp</td>
<td>0.0007</td>
<td>RMSProp</td>
<td>0.00005</td>
<td>128</td>
<td>150</td>
</tr>
<tr>
<td>PMNIST</td>
<td>512</td>
<td>$\infty$</td>
<td>0.3</td>
<td>0.3</td>
<td>RMSProp</td>
<td>0.0003</td>
<td>0.00007</td>
<td>128</td>
<td>150</td>
<td></td>
</tr>
<tr>
<td>TIMIT</td>
<td>224</td>
<td>$\infty$</td>
<td>0.3</td>
<td>0.1</td>
<td>Adam</td>
<td>0.001</td>
<td>RMSProp</td>
<td>0.0002</td>
<td>128</td>
<td>700</td>
</tr>
<tr>
<td>TIMIT</td>
<td>322</td>
<td>$\infty$</td>
<td>0.3</td>
<td>0.1</td>
<td>Adam</td>
<td>0.001</td>
<td>0.0002</td>
<td>128</td>
<td>700</td>
<td></td>
</tr>
</tbody>
</table>

Table B.3: Hyperparameters for AdamLSTM and AdamDTRIV Training

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Optimizer</th>
<th>Momentum $\mu$</th>
<th>Step Size $s$</th>
<th>$\beta$</th>
<th>Learning Rate</th>
<th>Batch Size</th>
<th>#Epochs</th>
</tr>
</thead>
<tbody>
<tr>
<td>MNIST</td>
<td>RMSProp</td>
<td>0.6</td>
<td>0.6</td>
<td>0.1</td>
<td>0.001</td>
<td>128</td>
<td>150</td>
</tr>
<tr>
<td>PMNIST</td>
<td>RMSProp</td>
<td>0.6</td>
<td>1.0</td>
<td>0.01</td>
<td>0.001</td>
<td>128</td>
<td>150</td>
</tr>
<tr>
<td>TIMIT</td>
<td>Adam</td>
<td>0.3</td>
<td>0.1</td>
<td>0.999</td>
<td>0.0001</td>
<td>32</td>
<td>700</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Size</th>
<th>DTRIV</th>
<th>Momentum $\mu$</th>
<th>Step Size $s$</th>
<th>$\beta$</th>
<th>Optimizer</th>
<th>Learning Rate</th>
<th>Orthogonal Rate</th>
<th>Orthogonal Size</th>
<th>Batch</th>
<th>#Epochs</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMNIST</td>
<td>512</td>
<td>$\infty$</td>
<td>0.3</td>
<td>0.3</td>
<td>0.8</td>
<td>RMSProp</td>
<td>0.0003</td>
<td>RMSProp</td>
<td>0.00007</td>
<td>128</td>
<td>150</td>
</tr>
</tbody>
</table>
Table B.4 : Hyperparameters for RMSPropLSTM and RMSPropDTRIV Training

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Optimizer</th>
<th>Momentum $\mu$</th>
<th>Step Size $s$</th>
<th>$\beta$</th>
<th>Learning Rate</th>
<th>Batch Size</th>
<th>#Epochs</th>
</tr>
</thead>
<tbody>
<tr>
<td>MNIST</td>
<td>RMSProp</td>
<td>0.0</td>
<td>0.6</td>
<td>0.9 (size $N = 256$), 0.99 (size $N = 128$)</td>
<td>0.001</td>
<td>128</td>
<td>150</td>
</tr>
<tr>
<td>PMNIST</td>
<td>RMSProp</td>
<td>0.0</td>
<td>1.0</td>
<td>0.01</td>
<td>0.001</td>
<td>128</td>
<td>150</td>
</tr>
<tr>
<td>TIMIT</td>
<td>Adam</td>
<td>0.0</td>
<td>0.1</td>
<td>0.999</td>
<td>0.0001</td>
<td>32</td>
<td>700</td>
</tr>
</tbody>
</table>

RMSPropDTRIV

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Size</th>
<th>DTRIV</th>
<th>Momentum</th>
<th>Step Size $s$</th>
<th>$\beta$</th>
<th>Optimizer</th>
<th>Learning</th>
<th>Orthogonal</th>
<th>Orthogonal</th>
<th>Batch</th>
<th>#Epochs</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMNIST</td>
<td>512</td>
<td>$\infty$</td>
<td>0.0</td>
<td>0.3</td>
<td>0.9</td>
<td>RMSProp</td>
<td>0.0003</td>
<td>RMSProp</td>
<td>0.00007</td>
<td>128</td>
<td>150</td>
</tr>
</tbody>
</table>

Table B.5 : Hyperparameters for SRLSTM and SRDTRIV Training

SRLSTM

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

SRDTRIV

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Size</th>
<th>DTRIV</th>
<th>Scheduled Restart (F)</th>
<th>Step Size $s$</th>
<th>Optimizer</th>
<th>Learning</th>
<th>Orthogonal</th>
<th>Orthogonal</th>
<th>Batch</th>
<th>#Epochs</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMNIST</td>
<td>512</td>
<td>$\infty$</td>
<td>2</td>
<td>0.3</td>
<td>RMSProp</td>
<td>0.0003</td>
<td>RMSProp</td>
<td>0.00007</td>
<td>128</td>
<td>150</td>
</tr>
</tbody>
</table>

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

$$
\frac{\partial \mathcal{L}_t}{\partial U} = \sum_{k=1}^{t} \frac{\partial h_k}{\partial U} \cdot \frac{\partial \mathcal{L}_t}{\partial h_t} \cdot \frac{\partial h_t}{\partial h_k} = \sum_{k=1}^{t} \frac{\partial h_k}{\partial U} \cdot \frac{\partial \mathcal{L}_t}{\partial h_t} \cdot \prod_{k=1}^{t-1} \frac{\partial h_{k+1}}{\partial h_k}, \quad (B.2)
$$
where \( \frac{\partial h_{k+1}}{\partial h_k} = D_k U^T \) with \( D_k = \text{diag}(\sigma'(Uh_k + Wx_{k+1} + b)) \). Similarly, we can compute \( \frac{\partial L_t}{\partial W} \) and \( \frac{\partial L_t}{\partial b} \).

### B.3 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 B.6 and B.7, as well as in Figure B.1. 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 B.8 and Figure B.2. Hyperparameter values for this experiment can be found in Table B.3, B.4, and B.5 (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 MomentumLSTM in terms of test accuracy. SRLSTM also converges the fastest. We summarize our results in Table B.9 and Figure B.3. Hyperparameter values for this experiment can be found in Table B.5 (top).
Table B.6: Best test accuracy at the MNIST and PMNIST tasks (%). We use the baseline results reported in [3], [4], [5]. 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.

<table>
<thead>
<tr>
<th>Model</th>
<th>$N$</th>
<th># params</th>
<th>MNIST</th>
<th>PMNIST</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSTM</td>
<td>128</td>
<td>$\approx 68K$</td>
<td>98.70 [3], 97.30 [5]</td>
<td>92.00 [3], 92.62 [5]</td>
</tr>
<tr>
<td>MomentumLSTM</td>
<td>128</td>
<td>$\approx 68K$</td>
<td>99.04 ± 0.04</td>
<td>93.40 ± 0.25</td>
</tr>
<tr>
<td>AdamLSTM</td>
<td>128</td>
<td>$\approx 68K$</td>
<td>98.98 ± 0.08</td>
<td>93.75 ± 0.25</td>
</tr>
<tr>
<td>RMSPropLSTM</td>
<td>128</td>
<td>$\approx 68K$</td>
<td>99.09 ± 0.05</td>
<td>94.32 ± 0.43</td>
</tr>
<tr>
<td>SRLSTM</td>
<td>128</td>
<td>$\approx 68K$</td>
<td>98.89 ± 0.08</td>
<td>93.65 ± 0.56</td>
</tr>
</tbody>
</table>

Table B.7: 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.

<table>
<thead>
<tr>
<th>Model</th>
<th>$N$</th>
<th># params</th>
<th>Val. MSE</th>
<th>Test MSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSTM</td>
<td>120</td>
<td>$\approx 135K$</td>
<td>11.77 ± 0.14 (13.93 [3, 77])</td>
<td>11.83 ± 0.12 (12.95 [3, 77])</td>
</tr>
<tr>
<td>MomentumLSTM</td>
<td>120</td>
<td>$\approx 135K$</td>
<td>8.00 ± 0.30</td>
<td>8.04 ± 0.30</td>
</tr>
<tr>
<td>AdamLSTM</td>
<td>120</td>
<td>$\approx 135K$</td>
<td>10.91 ± 0.08</td>
<td>10.96 ± 0.08</td>
</tr>
<tr>
<td>RMSPropLSTM</td>
<td>120</td>
<td>$\approx 135K$</td>
<td>11.83 ± 0.20</td>
<td>11.90 ± 0.19</td>
</tr>
<tr>
<td>SRLSTM</td>
<td>120</td>
<td>$\approx 135K$</td>
<td>8.15 ± 0.26</td>
<td>8.21 ± 0.26</td>
</tr>
</tbody>
</table>
Table B.8: 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.

<table>
<thead>
<tr>
<th>Model</th>
<th>( N )</th>
<th># params</th>
<th>PMNIST</th>
</tr>
</thead>
<tbody>
<tr>
<td>DTRIV</td>
<td>170</td>
<td>( \approx 16K )</td>
<td>95.21 ( \pm 0.10 ) (95.20 [6])</td>
</tr>
<tr>
<td>DTRIV</td>
<td>360</td>
<td>( \approx 69K )</td>
<td>96.45 ( \pm 0.10 ) (96.50 [6])</td>
</tr>
<tr>
<td>DTRIV</td>
<td>512</td>
<td>( \approx 137K )</td>
<td>96.62 ( \pm 0.12 ) (96.80 [6])</td>
</tr>
<tr>
<td>MomentumDTRIV</td>
<td>170</td>
<td>( \approx 16K )</td>
<td>95.37 ( \pm 0.09 )</td>
</tr>
<tr>
<td>MomentumDTRIV</td>
<td>360</td>
<td>( \approx 69K )</td>
<td>96.73 ( \pm 0.08 )</td>
</tr>
<tr>
<td>MomentumDTRIV</td>
<td>512</td>
<td>( \approx 137K )</td>
<td>96.89 ( \pm 0.08 )</td>
</tr>
<tr>
<td>AdamDTRIV</td>
<td>512</td>
<td>( \approx 137K )</td>
<td>96.77 ( \pm 0.21 )</td>
</tr>
<tr>
<td>RMSPropDTRIV</td>
<td>512</td>
<td>( \approx 137K )</td>
<td>96.75 ( \pm 0.12 )</td>
</tr>
<tr>
<td>SRDTRIV</td>
<td>512</td>
<td>( \approx 137K )</td>
<td>97.02 ( \pm 0.09 )</td>
</tr>
</tbody>
</table>

Figure B.1: 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 B.9: 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.

<table>
<thead>
<tr>
<th>Model</th>
<th># params</th>
<th>Val. PPL</th>
<th>Test PPL</th>
</tr>
</thead>
<tbody>
<tr>
<td>lstm</td>
<td>≈ 24M</td>
<td>61.96 ± 0.83</td>
<td>59.71 ± 0.99 (58.80 [78])</td>
</tr>
<tr>
<td>MomentumLSTM</td>
<td>≈ 24M</td>
<td><strong>60.71 ± 0.24</strong></td>
<td><strong>58.62 ± 0.22</strong></td>
</tr>
<tr>
<td>SRLSTM</td>
<td>≈ 24M</td>
<td>61.12 ± 0.68</td>
<td>58.83 ± 0.62</td>
</tr>
</tbody>
</table>

Figure B.2: 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.

B.4 Computational Time and Memory Cost: RNN vs. MomentumRNN

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 B.10 and B.11, respectively.
Figure B.3: 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 B.10: Computation time per sample at training and evaluation for PMNIST classification task using models with 256 hidden units.

<table>
<thead>
<tr>
<th>Model</th>
<th>Training Time (µs/sample)</th>
<th>Evaluation Time (µs/sample)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSTM</td>
<td>6.18</td>
<td>2.52</td>
</tr>
<tr>
<td>MomentumLSTM</td>
<td>7.43</td>
<td>3.16</td>
</tr>
<tr>
<td>AdamLSTM</td>
<td>10.34</td>
<td>4.07</td>
</tr>
<tr>
<td>RMSPropLSTM</td>
<td>9.94</td>
<td>3.96</td>
</tr>
<tr>
<td>SRLSTM</td>
<td>8.34</td>
<td>3.16</td>
</tr>
</tbody>
</table>
Table B.11: Memory cost per sample at training and evaluation for PMNIST classification task using models with 256 hidden units.

<table>
<thead>
<tr>
<th>Model</th>
<th>Training Memory (MB/sample)</th>
<th>Evaluation Memory (MB/sample)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSTM</td>
<td>15.93</td>
<td>7.51</td>
</tr>
<tr>
<td>MomentumLSTM</td>
<td>15.95</td>
<td>7.51</td>
</tr>
<tr>
<td>AdamLSTM</td>
<td>25.13</td>
<td>7.52</td>
</tr>
<tr>
<td>RMSPropLSTM</td>
<td>25.13</td>
<td>7.52</td>
</tr>
<tr>
<td>SRLSTM</td>
<td>15.95</td>
<td>7.51</td>
</tr>
</tbody>
</table>

B.5 MomentumLSTM Cell Implementation in Pytorch

```python
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

```python
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
Bibliography


[49] F. Chollet et al., “Keras.” [https://keras.io](https://keras.io), 2015.


[64] Y. Bengio, P. Simard, and P. Frasconi, “Learning long-term dependencies with gradient descent is difficult,” *IEEE Transactions on Neural Networks*, vol. 5,


