École Polytechnique Fédérale de Lausanne

Stochastic variance reduced gradient methods for training Deep Neural Networks

by Alexander Apostolov

Semester Project Report

Project advised by:

Prof. Dr. Martin Jaggi
Advising Professor

Dr. Tatjana Chavdarova
Supervisor

EPFL IC IINFCOM MLO
INJ 130 (Bâtiment INJ)
Station 14
CH-1015 Lausanne

February 1, 2021

Abstract

This project focuses on analyzing the effect of stochastic variance reduced gradient methods on training Deep Neural Networks (DNNs), with a major focus on *stochastic variance reduced gradient* [SVRG, Johnson and Zhang, 2013]. Despite their notable theoretical advantages in terms of convergence speed and their simplicity, relative to some widely used optimization methods (*e.g.* Adam–which lacks theoretical understanding and convergence proof), variance reduced methods yet remain unpopular in deep learning.

Their unpopularity mainly follows from the fact that these methods were empirically shown ineffective for training DNNs [Defazio and Bottou, 2019], which result, to our knowledge, lacks deeper understanding of the reasons of this failure. Moreover, modern DNNs incorporate various stabilizing techniques–such as Batch Normalization [Ioffe and Szegedy, 2015], which break some assumptions of these methods. This project represents an empirical approach to improve our understanding of why variance reduction methods fail to perform well on DNNs.

Apart from SVRG, we empirically investigate how different optimization methods perform on DNNs of varying depth, including the standard Stochastic Gradient Descent (SGD), SGD with momentum, *Adadelta* [Zeiler, 2012], *Adam* [Kingma and Ba, 2015] and a more recently proposed method named *STOchastic Recursive Momentum* [STORM, Cutkosky and Orabona, 2019].

Consistent with theoretical results on convex functions, we show that variance reduced gradient methods give good results on shallow convolutional neural networks (CNNs). We also show that in a setup where batch-normalization layers are omitted from a deeper architecture, stochastic variance reduced gradient methods outperform their respective stochastic counterpart.

Contents

Chapter 1

Introduction

Optimization problems are a centerpiece of many machine learning tasks and choosing an appropriate optimization algorithm is a crucial step to get good results. Stochastic variance reduced gradient methods[1] have been shown to theoretically have better convergence than stochastic gradient descent methods. However, they are not widely used as they seem to be outperformed on most standard deeper neural networks. These worse performances have been empirically described [Defazio and Bottou, 2019] but there are no theoretical insights on the reasons for that.

This project tries to build an empirical approach to improve our understanding on why variance reduction methods fail to perform well on DNNs. To do so, we look at **Stochastic Variance Reduced Gradient (SVRG)** [Johnson and Zhang, 2013] and **STOchastic Recursive Momentum (STORM)** [Cutkosky and Orabona, 2019], two variance reduced methods and compare them on different architectures. We compare them with **Stochastic Gradient Descent (SGD)** and **Adam** [Kingma and Ba, 2015].

We think that using batch-normalization layers in a neural network might be one of the reasons of the worse performance of variance reduced methods, so we perform different experiments using networks with and without these layers. Finally, we think that having better weight initialization can help in the cases when we are training on DNNs without batch-normalization layers, so we try to observe effects by using **MetaInit** [Dauphin and Schoenholz, 2019], an algorithm that helps finding weights before starting to train a model.

[1] Herein referred as *Variance Reduced*, for short.

Chapter 2

Background & Related Works

2.1 The finite-sum setting

In machine learning, the following optimization is often encountered. Let w denote the model parameters, and let $f_1, f_2, \ldots, f_n$ be a sequence of vector functions $\mathbb{R}^d \mapsto \mathbb{R}$ where each $f_i(\cdot)$ is the loss on a single training data point. The goal is to find a solution to the following "finite-sum" problem:

$$\min_w F(w), \quad F(w) = \frac{1}{n}\sum_{i=1}^{n} f_i(w). \tag{2.1}$$

There are many examples of such functions we need to minimize, they are often referred to as loss functions. When trying to do predictions in $\mathbb{R}$ using features very common ones are Mean Square Error (MSE) and Mean Absolute Error (MAE). Given a dataset $S = \{(y_i, x_i)\}_{i=1}^n$ where $y_i \in \mathbb{R}$ and $x_i \in \mathbb{R}^d$ and a prediction function $p_w : \mathbb{R}^d \mapsto \mathbb{R}$ depending on some weights or parameters w, the Mean Square Error is defined as:

$$F_{MSE}(w) = \frac{1}{2n}\sum_{i=1}^{n} (y_i - p_w(x_i))^2$$

In the same setting, the Mean Absolute Error is defined as:

$$F_{MAE}(w) = \frac{1}{n}\sum_{i=1}^{n} |y_i - p_w(x_i)|$$

Another example of such loss functions we are trying to minimize in image segmentation tasks is the Jaccard Loss. Given some images where we want to detect some areas of interest (free rooftop areas for photovoltaic panels installations in satellite images, pedestrians on images from a car etc…) we can be given the set of correct labels $y_i \in \{0,1\}^{H \times W}$ for some images

$x_i \in \mathbb{R}^{H \times W}$ of height H and width W and we wish to find a function p_w which assigns a label 1 or 0 to each pixel to indicate weather it is in the area of interest or not: $p_w : \mathbb{R}^{H \times W} \mapsto \{0,1\}^{H \times W}$. Then the Jaccard loss is given by:

$$F_{Jaccard}(w) = \frac{1}{n} \sum_{i=1}^{n} \frac{p_w(x_i) \cap y_i}{p_w(x_i) \cup y_i}$$

2.2 Full–Batch Gradient Descent (GD)

A widely used and straightforward method to solve Equation 2.1 is to use **Gradient Descent (GD)**. This method relies on the fact that the gradient of a function, $\nabla F(w^{(t)})$, points towards the steepest ascent direction, so one can use the opposite direction of the gradient to go towards a minimum of a function. This method is iterative, at each iteration $t = 1, 2, \dots T$ we calculate new weights for $F(w)$ by:

$$w^{(t)} = w^{(t-1)} - \eta_t \nabla F(w^{(t-1)}), \tag{2.2}$$

where η_t is the learning rate at iteration t. A problem with Gradient Descent is that each update step requires computing the gradient over the whole dataset. This can require a lot of computations, making this algorithm inefficient for some tasks.

2.3 Stochastic Gradient Descent (SGD)

This drawback is why **Stochastic Gradient Descent (SGD)** has been widely adopted. Instead of computing the gradient over the whole dataset at each iteration, we only use the gradient with respect to one datapoint. The algorithm is given by the following update rule. At each iteration $t = 1, 2 \dots T$, sample one datapoint $i_t \in \{1, \dots n\}$ uniformly at random:

$$w^{(t)} = w^{(t-1)} - \eta_t \nabla f_{i_t}(w^{(t-1)}), \tag{2.3}$$

where η_t is the learning rate at iteration t.

The idea behind this trick is that it decreases the computational complexity of each iteration by a factor of $\theta(1/n)$ and one can show that in expectation the gradient in SGD is equal to the full gradient over the whole dataset:

$$\mathbb{E}[\nabla f_{i_t}(w)] = \nabla F(w). \tag{2.4}$$

A very similar method is the so called **Mini-batch SGD** where instead of choosing one datapoint x_{i_t}, we choose a subset B_t of all the datapoints uniformly at random at each iteration and then use the following rule. For each iteration $t = 1, 2 \dots T$:

$$w^{(t)} = w^{(t-1)} - \eta_t \frac{1}{|B_t|} \sum_{i \in B_t} \nabla f_i(w^{(t-1)}), \tag{2.5}$$

where η_t is the learning rate at iteration t.

Both Stochastic Gradient Descent and Mini-batch Stochastic Gradient Descent use an unbiased estimate of the gradient to have faster iterations, but using only a subset of the dataset **increases the variance** of the gradient estimates, which slows down convergence. There is thus a trade-off between fast per iteration computations and fast convergence. Variance reduced methods allow us to get a better deal from this trade-off by keeping fast per-iteration computations while maintaining a fast convergence.

2.4 Variance Reduced Gradient

In this project we will analyze two variance reduced algorithms: **Stochastic Variance Reduced Gradient (SVRG)** [Johnson and Zhang, 2013] and **STOchastic Recursive Momentum (STORM)** [Cutkosky and Orabona, 2019].

2.4.1 Stochastic Variance Reduced Gradient (SVRG)

SVRG is an iterative algorithm whose goal is to solve Equation 2.1 by using elements of Stochastic Gradient Descent to ensure fast iterations in terms of computational complexity and elements of Gradient Descent to get low variance. The main idea behind this algorithm, presented by Rie Johnson and Tong Zhang in 2013, is that we do a snapshot $\tilde{w}$ of the parameters every m iterations and compute the gradient $\tilde{\mu}$ over the whole dataset with respect to $\tilde{w}$ to get an estimate of the true gradient. Similarly to SGD, we take a datapoint x_{i_t} (or more generally a mini-batch) uniformly at random at each iteration $t = 1, 2, \dots T$ and apply the following update rule:

$$w^{(t)} = w^{(t-1)} - \eta(\nabla f_{i_t}(w^{(t-1)}) - \nabla f_{i_t}(\tilde{w}) + \tilde{\mu}), \tag{2.6}$$

where $\tilde{w}$ is snapshot of w taken when $\tilde{\mu}$ is computed every m iterations, and:

$$\tilde{\mu} = \nabla F(\tilde{w}) = \frac{1}{n} \sum_{i=1}^{n} \nabla f_i(\tilde{w}). \tag{2.7}$$

The hyperparameter m is usually in the order of a few epochs. The original paper suggests setting m to 3 or 5 epochs for convex objectives and non-convex objectives such as Neural Networks respectively.

An important thing to note is that $\nabla f_{i_t}(w^{(t-1)}) - \nabla f_{i_t}(\tilde{w}) + \tilde{\mu}$ is an unbiased estimate of $\nabla F(w)$, so we have:

$$\mathbb{E}[w^{(t)}|w^{(t-1)}] = w^{(t-1)} - \eta_t \nabla F(w^{(t-1)}). \tag{2.8}$$

So in expectation the updates are the same as in GD.

One can also show that variance is reduced with SVRG. When $\tilde{w}$ and $w^{(t)}$ both converge to w_*, then $\tilde{\mu} \to 0$. If $\nabla f_{i_t}(\tilde{w}) \to \nabla f_i(w_*)$, then:

$$\nabla f_{i_t}(w^{(t-1)}) - \nabla f_{i_t}(\tilde{w}) + \tilde{\mu} \to \nabla f_{i_t}(w^{(t-1)}) - \nabla f_{i_t}(w_*) \to 0 \tag{2.9}$$

For completeness, the pseudocode of the SVRG procedure is given in algorithm 1

Algorithm 1: SVRG PROCEDURE

Input: learning rate η and update frequency m
Initialize $\tilde{w}_0$
for $epoch \leftarrow 1, 2, \ldots$ **do**
  $\tilde{w} \leftarrow \tilde{w}_{epoch-1}$
  $\tilde{\mu} \leftarrow \frac{1}{n}\sum_{i=1}^{n} \nabla f_i(\tilde{w})$
  $w^{(0)} \leftarrow \tilde{w}$
  for $t \leftarrow 1, 2 \ldots, T$ **do**
    Randomly pick $i_t \in \{1, \ldots, n\}$
    $w^{(t)} \leftarrow w^{(t-1)} - \eta(\nabla f_{i_t}(w^{(t-1)}) - \nabla f_{i_t}(\tilde{w}) + \tilde{\mu})$
  end
  Save snapshot $\tilde{w}_{epoch} \leftarrow w_T$
end

2.5 STOchastic Recursive Momentum (STORM)

The second variance reduction method we are considering is STOchastic Recursive Momentum (STORM) recently proposed by Cutkosky and Orabona. This is an iterative algorithm to solve Equation 2.1 similarly to SGD with momentum. At each iteration t a datapoint x_{i_t} (or more generally a mini-batch) is chosen uniformly at random and the following update rule is applied:

$$d^{(t)} = (1-a)d^{(t-1)} + a\nabla f_{i_t}(w^{(t)}) + (1-a)(\nabla f_{i_t}(w^{(t)}) + \nabla f_{i_t}(w^{(t-1)})), \tag{2.10}$$

with

$$w^{(t+1)} = w^{(t)} - \eta_t d_t \,. \tag{2.11}$$

The complete procedure is given in algorithm 2.

Two thing to notice here are that the learning rate changes at each iteration and that Equation 2.10 is very similar to SGD with momentum. In fact the first two terms are the same and when $w^{(t)} \approx w^{(t-1)}$ the last term of the equality tends to 0. Then the update is exactly the same as SGD with momentum.

Algorithm 2: STORM PROCEDURE

Input: Hyperparameters k, w, c
Initialize w_1
$G_1 \leftarrow |\nabla f_{i_1}(w_1)|$
$d_1 \leftarrow \nabla f_{i_1}(w_1)$
for $t \leftarrow 1, 2, \ldots$ **do**
  $\eta_t \leftarrow \frac{k}{(w_t + \sum_{i=1}^{t} G_i^2)^{1/3}}$
  $w_{t+1} \leftarrow w_t - \eta_t d_t$
  $a_{t+1} \leftarrow c\eta_t^2$
  $G_{t+1} \leftarrow |\nabla f_{i_{t+1}}(w_{t+1})|$
  $d_{t+1} \leftarrow \nabla f_{i_{t+1}}(w_{t+1}) + (1 - a_{t+1})(d_t - \nabla f_{i_{t+1}}(w_t))$
end

One can reflect on the differences between SVRG and STORM. A disadvantage of STORM is that is not straightforward to get clear insights about it and it has more hyperparameters than SVRG that are not self-evident how to fix as discussed with one of the authors of the paper. On the other hand a disadvantage of SVRG is that it needs to calculate the full gradient every m epochs which makes for a significantly slower iteration. One can also note that these two algorithms both need at least two gradient calculations per iteration compared to one for regular SGD.

2.6 Adam

Throughout the experiments of this project we also use the optimization algorithm Adam [Kingma and Ba, 2015]. This method presented by Diederik P. Kingma and Jimmy Lei Ba is widely used for optimization of stochastic objectives as it is straightforward to implement, is computationally efficient, has little memory requirements, shows very good results for various tasks and present in many standard libraries for machine learning. Adam uses adaptive estimates of first and second order moments to adapt the learning rates of every parameter

of the network separately as opposed to just one global learning rate in GD or SGD. These adaptive learning rates are computed from moving averages of the 1^{st} and 2^{nd} order moments of the loss function and help reach better results during training. The procedure is presented in algorithm 3.

Algorithm 3: ADAM PROCEDURE

Input: learning rate η, exponential decay rates for the moment estimates $\beta_1, \beta_2 \in [0,1)$
Initialize w_0 Parameter vector
Initialize m_0 (1^{st} moment vector)
Initialize v_0 (2^{nd} moment vector)
Initialize t (timestamp)
while w_t *not converged* **do**
$t \leftarrow t+1$
Randomly pick $i_t \in 1,\dots,n$
$g_t \leftarrow \nabla_w f_{i_t}(w_{t-1})$
$m_t \leftarrow \beta_1 * m_{t-1} + (1-\beta_1) * g_t$ (update biased 1^{st} moment estimate)
$v_t \leftarrow \beta_2 * v_{t-1} + (1-\beta_2) * g_t^2$ (update biased 2^{nd} moment estimate)
$\hat{m}_t \leftarrow \frac{m_t}{1-\beta_1^t}$ (compute bias-corrected 1^{st} moment estimate)
$\hat{v}_t \leftarrow \frac{v_t}{1-\beta_2^t}$ (compute bias-corrected 2^{nd} moment estimate)
$w_t \leftarrow w_{t-1} - \eta * \frac{\hat{m}_t}{\sqrt{\hat{v}_t}+\epsilon}$
end

2.7 Batch-normalization layers

Another important theoretical aspect for this project are batch-normalization layers.The goal of these is to normalize the input of layers of the network with the following formula:

$$y = \frac{x - \mathbb{E}[x]}{\sqrt{\mathbb{V}ar[x] + \epsilon}}, \tag{2.12}$$

where x is the output of the previous layer (i.e. the input to the batch-normalization layer), y is the output of the batch-normalization layer and ϵ is a small number added for numerical stability in case of 0 variance. The batch-normalization layers however do not have a direct access to the value of $\mathbb{E}[x]$ and $\mathbb{V}ar[x]$ as $x's$ are only seen throughout the training, so the batch-normalization layer keeps running estimates for these two values by using average computations with momentum.

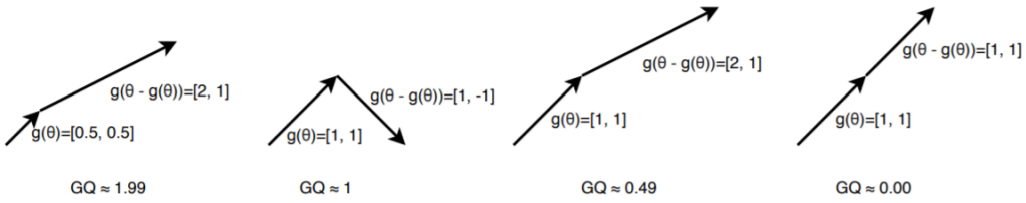

Figure 2.1: Gradient quotient for different scenarios of initial gradient $g(\theta)$ and gradient after one update $g(\theta - g(\theta))$. A situation where the optimization is locally linear with minimal second order effects (see last example) has a small change between the gradients and thus a small gradient quotient.

2.8 Meta Initialization

Another concept we try in this project is initializing weights of the model before training with a method called **MetaInit**. The method was presented in *MetaInit: Initializing learning by learning to initialize* [Dauphin and Schoenholz, 2019] in 2019. Deep Neural networks can be difficult to train and their performance depends heavily on how weights are initialized. The hypothesis made by Yann N. Dauphin, Samuel Schoenholz, the authors of the MetaInit paper, is that it is better to start training in regions of the optimization landscape that look locally linear with minimal second order effects. To measure this they use a metric which they call gradient quotient (GQ) defined in Equation 2.13. The goal of the MetaInit algorithm is to minimize this gradient quotient before training using Gradient Descent and by doing so, MetaInit selects the initial weights for training.

$$GQ(L,\theta) = \frac{1}{N}\left\| \frac{\mathbf{g}(\theta - \mathbf{g}(\theta))}{\mathbf{g}(\theta) + \epsilon} - 1 \right\|_1 \tag{2.13}$$

Where $\theta \in \mathbb{R}^N$ are the parameters of the network, $l(x,\theta)$ is the loss function, $L(\theta)$ the mean over a mini-batch, $\mathbf{g}(\theta) = \nabla L$, $\epsilon = \epsilon_0(2_{\mathbf{g}(\theta)\geq 0} - 1)$ computes a damping factor with the right sign for each element and ϵ_0 is a small constant.

Examples of different gradient quotient are given in Figure 2.1 taken from the original paper describing MetaInit.

Chapter 3

Towards Understanding the Failure of VR Methods on DNNs: An Empirical Study

The first question we pose in this project is the following:

> Is the use of batch-normalization layers related to the failure of the stochastic variance reduced gradient (SVRG) method on deeper architectures?

A paper from 2019 by Defazio and Bottou *On the Ineffectiveness of Variance Reduced Optimization for Deep Learning* suggests that applying these techniques on hard non-convex problems generally fails to give good results. However, we note that their analysis does not give any theoretical insights on why this would be the case. Moreover, we see that when testing their assumptions on deeper networks, the architectures they use include batch-normalization layers [Ioffe and Szegedy, 2015]. These layers, because they use averages throughout training as explained in Equation 2.12 break the assumptions that the gradient estimates in Equation 2.6 for SVRG and Equation 2.10 for STORM are not unbiased. To test it we compare the results on networks with and without batch-normalization layers.

Another hypothesis we want to test is the following:

> Can initialization methods help give better results to variance reduced methods in a setting without batch-normalization layers, by selecting better initial weights for training?

Indeed, batch-normalization layers are helping a lot with problems of vanishing and ex-

ploding gradients and make it easier to train bigger models, so we think that removing the batch-normalization layers can make training harder but we want to see whether this can be counterbalanced by proper weight initialization.

Chapter 4

Experiments

4.1 Benchmark on shallow neural networks

To get insights on how variance reduced methods work we first perform a benchmark on a shallow neural network with different algorithms. We choose to use SVRG, STORM, SGD, Adam and AdaGrad [Duchi et al., 2011]. The first two are the only ones that have a variance reduction effect, SGD and Adam are chosen since they are very popular optimization algorithms and AdaGrad is also benchmarked as it is compared with STORM in its original paper.

To perform all subsequent experiments, we need to implement the two variance reduced methods, SVRG and STORM, as there is no official and available pytorch implementation. The code can be found in the repository of this project.

4.1.1 Setup

Dataset. We compare them on a classification task, recognizing hand-written digits from the MNIST dataset [LeCun et al., 2010]. You can see example digits in Figure 4.1.

Architecture. The used architecture is LeNet [LeCun et al., 1989]. The network is presented in Figure 4.2. The Network is composed of two convolutions with a 5×5 kernel, each followed by an average-pooling layer with a 2×2 kernel and 3 dense layers at then end.

Hyper-parameter selection procedure. To compare the algorithm we performed a **4-fold cross validation** procedure. We did a grid search for the hyperparameters for the algorithms of

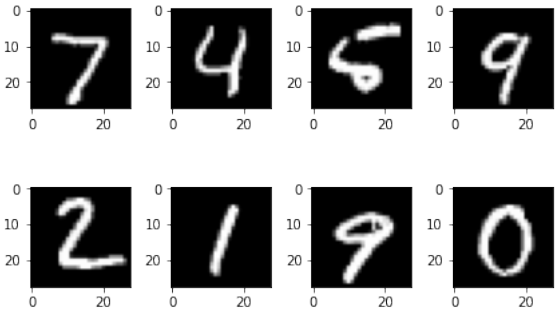

Figure 4.1: Examples of hand-written digits in the MNIST dataset.

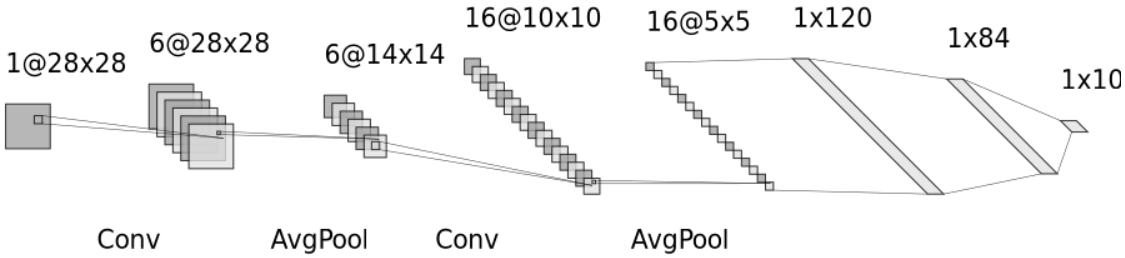

Figure 4.2: LeNet network used for MNIST

Algorithm	Hyperparmeter	Considered values
Adam	η	$\{10^{-5}, 10^{-4}, 10^{-3}, 0.01, 0.1\}$
SGD	η	$\{10^{-5}, 10^{-4}, 10^{-3}, 0.01, 0.1\}$
AdaGrad	η	$\{10^{-4}, 10^{-3}, 0.01, 0.1\}$
SVRG	η	$\{10^{-5}, 10^{-4}, 10^{-3}, 0.01, 0.1\}$
STORM	c	$\{0.1, 1, 10, 100, 1000\}$
	k	$\{10^{-3}, 0.01, 0.1, 1\}$

Table 4.1: Hyperparameters tested for the 4-Fold cross validation for the different algorithms on LeNet for classification of MNIST hand-written digits.

interest. The considered values are presented in Table 4.1. All cross validations were performed for 100 epochs as this appeared to be long enough for reaching convergence for the algorithms and hyperparameters of interest. Note that the hyperparameter w in STORM is not tuned since the original paper already suggests using a value of 0.1 for classifying the MNIST dataset, so we used the proposed value as testing a third hyperparameter would have significantly increased the computational time need to run the cross-validation.

Loss, batch-size & seed. All experiments are done with a fixed seed set to 1. We are using a batch size of 32 samples for all runs. The objective we want to minimize is the cross-entropy loss. Given C classes, this loss can be described as:

$$loss(\hat{y}, label) = -log\left(\frac{exp(\hat{y}[label])}{\sum_{c=1}^{C} exp(\hat{y}[c])}\right), \tag{4.1}$$

where $label$ is the true class of a given datapoint and $\hat{y}$ is a vector with C elements corresponding to the score attributed by the model to the given datapoint for each class.

Clarification on x-axis: gradient computation. To be able to properly compare the results of the different algorithms, we do not compare them in terms of iterations but rather in terms of gradient calculations. This is because SVRG and STORM are doing two gradient calculations per iterations and on top of that SVRG is calculating the whole gradient every m iterations.

4.1.2 Results

Adam cross-validation. For all 5 algorithms, we selected the best hyperparameters based on the best cross-validation error averaged over all 4-folds at the end of the 100 epochs. An example of the results after doing cross-validation for Adam is presented in Figure 4.3. In

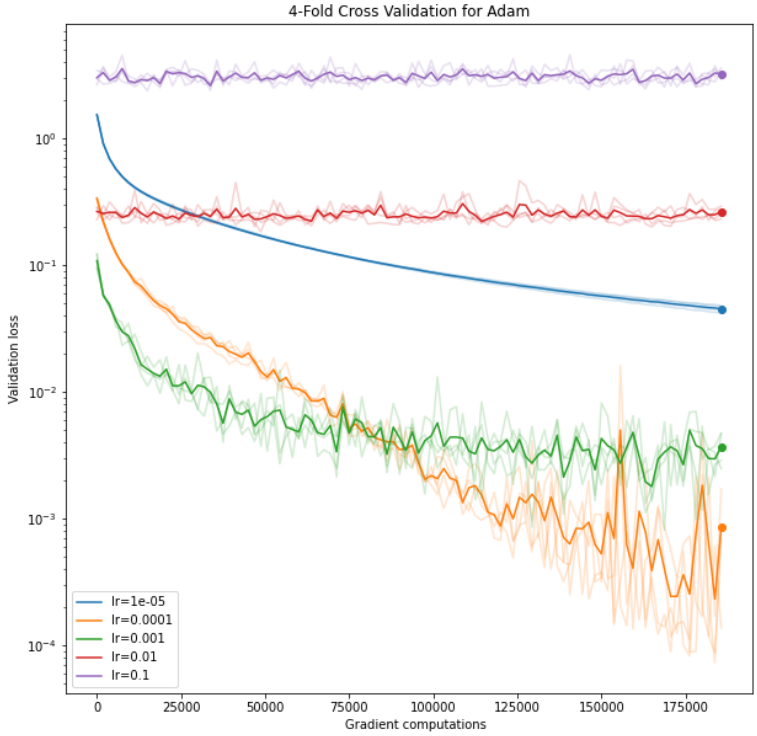

Figure 4.3: Cross-validation error during training of Adam optimizer for classification of MNIST dataset on LeNet architecture with gridsearch on selected values of the learning rate η (see Table 4.1). The solid lines represent the average of the loss across the 4 folds (represented by the faded lines) for each iteration.

this example we see that the best value for the learning rate of Adam for the given task is 10^{-4}. We see that learning rates with big values are stuck with values of the validation loss which are too high. Values of the learning rate which are too small, although they appear to have significantly less variance, because of the fact that each update only updates the weights by a tiny fraction, do not converge to good results in a reasonable time with our computational capacity. Moreover, small values of the learning rate might lead to situations where the algorithm is stuck in a local minimum of the optimization landscape.

Selected hyperparameters. After repeating the same method for all algorithms, we present the selected hyperparameters for each algorithm in Table 4.2.

Algorithm	Hyperparmeter	Selected value
Adam	η	10^{-4}
SGD	η	0.1
AdaGrad	η	0.1
SVRG	η	0.01
STORM	c	100
	k	0.1

Table 4.2: Selected hyperparameters after 4-Fold cross validation for the different algorithms on LeNet for classification of MNIST hand-written digits.

Benchmark. We then compare the 5 algorithms on a test set with the best-tuned hyperparameters to be able to compare them with each other. The results are presented in Figure 4.4 and Table 4.3. We see that SGD, Adam and AdaGrad are better at first but then the variance reduced methods, SVRG and STORM get better results. The fact that non-variance reduced methods are better at first can be explained by the fact that their iterations require only one gradient computation, so they can advance faster in their optimization.

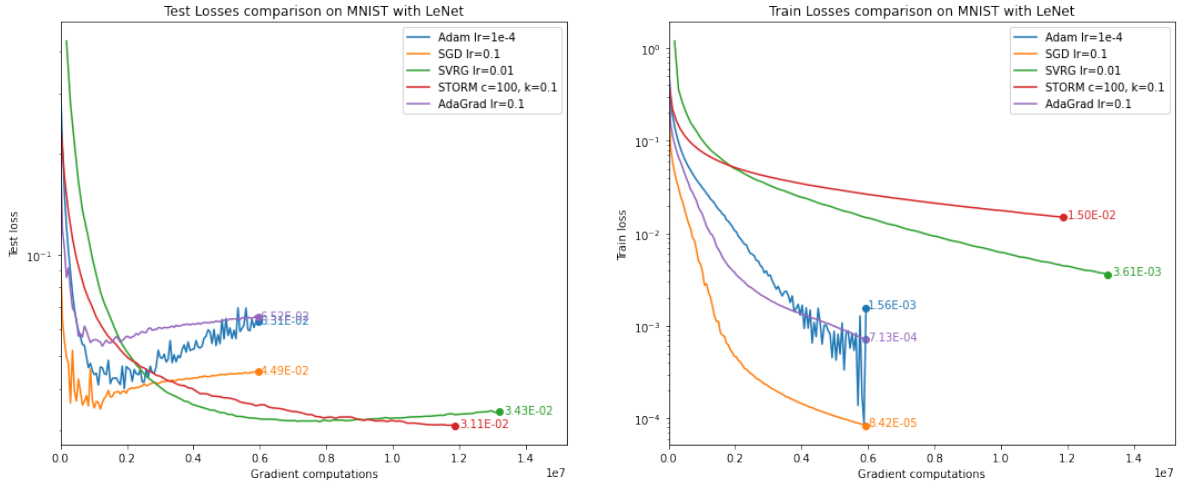

Figure 4.4: Test and train loss during training of Adam, SGD, SVRG, STORM and AdaGrad with hyperparameters selected after doing a 4-fold cross-validation for classification on **MNIST**, using LeNet. All methods are normalized by batch queries, thus the x-axis is gradient computations. The variance reduced methods are performing better than the others.

We thus see that as expected SVRG and STORM outperform the other stochastic methods in this setup.

Algorithm	Test accuracy	Train accuracy
Adam	0.9884	0.9999
SGD	0.9902	1.0
SVRG	0.9891	0.9996
STORM	0.9898	0.9964
AdaGrad	0.9859	0.9999

Table 4.3: Final test and train accuracies after training of Adam, SGD, SVRG, STORM and AdaGrad with hyperparameters selected after doing a 4-fold cross-validation for classification on **MNIST**, using LeNet.

4.2 Benchmark on deeper neural networks

To check the effect we are hypothesizing about the effect of batch-normalization layers, we first need to have results on DNNs with these layers. We choose a deep neural network with batch-normalization layers and compare Adam, SGD and SVRG. Because of the increased required computational resources we are not able to perform a proper cross-validation procedure, rather we select 5% of the training set and use it as a validation set to select the best hyperparameters for each model and then compare them on the test set. Moreover, for the same reason of increased computational time, we are unable to to get meaningful results for STORM as there are more hyperparameters to tune and we don't have insights on how to properly tune them. However, we think that results found on SVRG could probably also be applied to other variance reduced methods such as STORM.

4.2.1 ResNet18 with batch-normalization layers

Setup

Architecture The deep neural network we choose to perform our experiments on is ResNet18. This is a network for image recognition introduced in 2015 [He et al., 2015]. The key idea of this model is that it is constituted of residual learning building blocks with forward connections as presented in Figure 4.5. These are then put together one after the other as presented in Figure 4.6. ResNet models with batch-normalization layers have a batch-normalization layer after each convolution layer. If not otherwise specified, the ResNet models have batch-normalization layers in all subsequent experiments.

Dataset The task we perform with this model is classifying the image dataset CIFAR-10 [Krizhevsky, 2009]. These images are representing 10 classes: plane, car, bird, cat, deer,

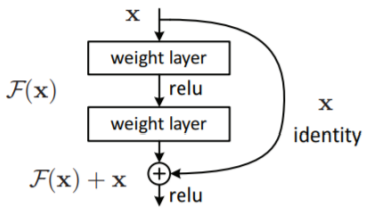

Figure 4.5: Residual learning: a building block.

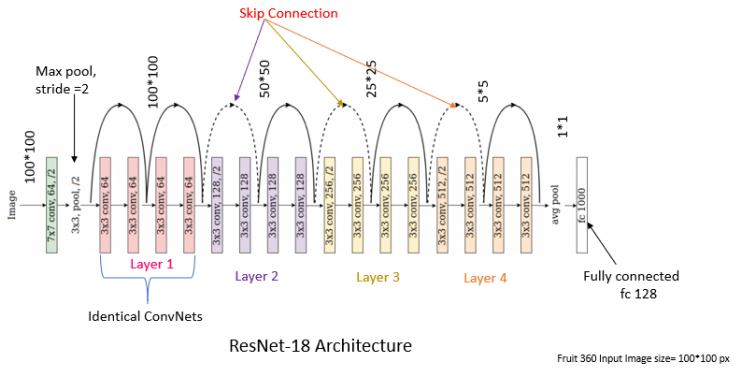

Figure 4.6: ResNet18 architecture with layers and skip connections shown.

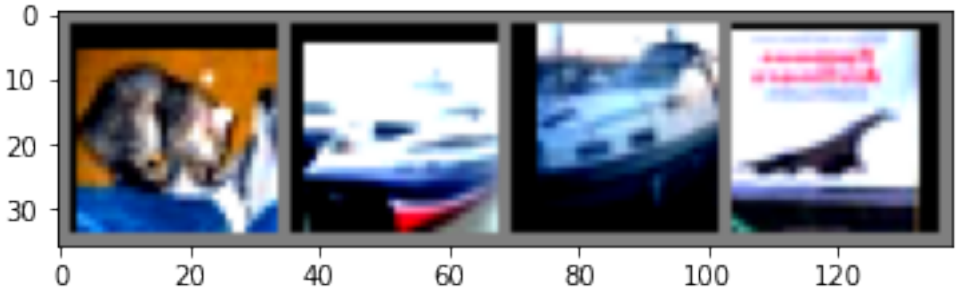

Figure 4.7: 4 samples from the CIFAR10 image dataset containing images of 10 classes: plane, car, bird, cat, deer, dog, frog, horse, ship and truck. The 4 samples here are cat, ship, ship and plane.

Algorithm	Hyperparmeter	Considered values
Adam	η	$\{10^{-5}, 10^{-4}, 10^{-3}\}$
SGD	η	$\{0.005, 0.01, 0.05, 0.1\}$
SVRG	η	$\{0.0005, 0.010, 0.005, 0.01\}$

Table 4.4: Hyperparameters tested for validation for the different algorithms on ResNet18 for classification of CIFAR10 images.

dog, frog, horse, ship and truck. A small sample is given in Figure 4.7.

Input transformations We apply the following transforms to these images: we first pad them with 4 pixels on each side, then perform a random square crop of size 32×32 and then perform a horizontal flip with probability 0.5. We finally normalize each channel (reg, green and blue) with the means and standard deviations as calculated from the training set.

Loss, batch-size & seed We then use the Cross-Entropy loss as an objective as described in Equation 4.1. As before, we fix the random seed to 1 and this time we use batches of size 128. We do the training for 200 epochs for each algorithm except for SVRG where we only use 150 epochs. We notice that the best hyperparameters are usually similar or a bit smaller than the ones found in the case of the shallower neural network above. The choices of tested hyperparameters is presented in Table 4.4.

Algorithm	Hyperparmeter	Selected value
Adam	η	10^{-4}
SGD	η	0.05
SVRG	η	0.01

Table 4.5: Best hyperparameters after validation for the different algorithms on ResNet18 for classification of CIFAR10 images.

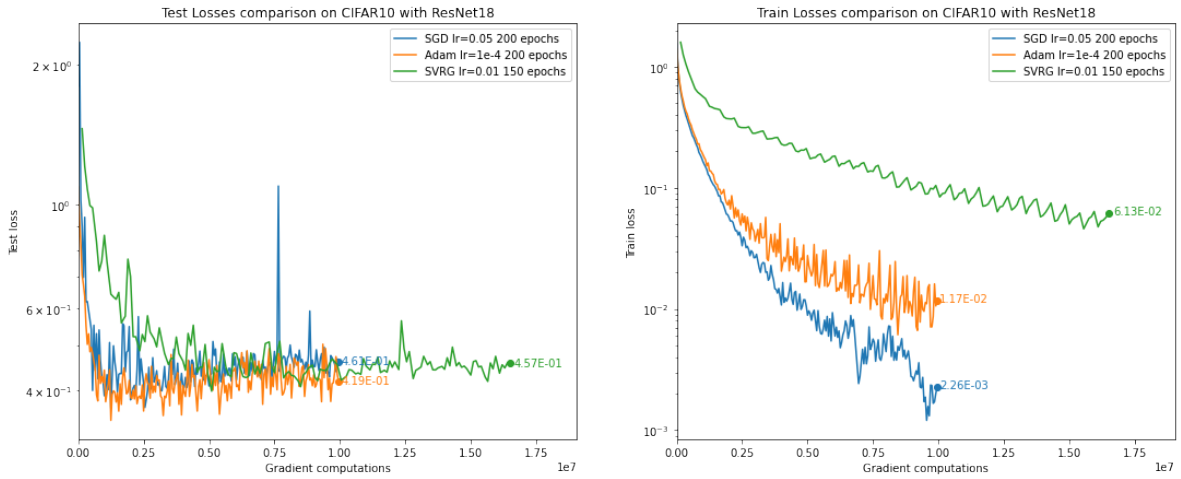

Figure 4.8: Benchmark of SGD, Adam and SVRG using test loss on **CIFAR10**, using **ResNet18**, where the x-axis is gradient computations. Train losses are also presented on the right plot. The hyperparameters are selected on the validation set. We observe that SVRG does outperform the rest of the methods in this setting.

Results

Selected hyperparameters After performing validation, we choose the best hyperparameters based on the validation loss at the end of training. The best hyperparameters are presented in Table 4.5.

Results on test set We compare the 3 algorithms with the best-tuned hyperparameters on the test set. In Figure 4.8 we show the test and train loss during training and in Table 4.6 we present the final accuracies. We see that in this setting the variance reduced method, SVRG is not outperforming the other methods as in the case of the shallow neural network.

Algorithm	Test accuracy	Train accuracy
SGD	0.9221	0.9996
ADAM	0.9184	0.9976
SVRG	0.886	0.97664

Table 4.6: Final test and train accuracies after training of SGD, ADAM and SVRG with hyperparameters selected after validation for classification on **CIFAR10**, using ResNet18.

4.2.2 ResNet18 without batch-normalization layers

Setup

Architecture To test our hypothesis we do the same analysis on the same network, ResNet18, except that this time we remove the batch-normalization layers.

Validation of hyperparameters We do the same validation process with the same values (see Table 4.4). One thing that we notice during this phase is that finding the correct hyperparameters becomes even more crucial in this scenario, as there are some choices for which the values become irrelevant. For example, we start getting NaN results for high learning rates of SGD. This might be due to problems of exploding gradients but we do not explore the exact reasons further as other choices of hyperparameters yield good results. However, it is important to note that removing batch-normalization layers can make some choices of hyperparameters completely irrelevant, although they were still getting acceptable results, although not the best, in a setting with a network with batch-normalization layers.

Results

Selected hyperparameters The selected hyperparameters for each algorithm are presented in Table 4.7 We notice that the choices of the best hyperparameters are always a little bit smaller than in the setting with a network with batch-normalization layers when we compare with the choices in Table 4.5.

Results on test set The results on the test set are presented in Figure 4.9 and in Table 4.8. In this setting we see that SVRG seems to get to better results that the other methods which are affected a lot by the removal of the batch-normalization layers. SVRG seems to be less negatively affected by the removal of the batch-normalization layers. We notice that the best

Algorithm	Hyperparmeter	Selected value
Adam	η	10^{-5}
SGD	η	0.01
SVRG	η	0.005

Table 4.7: Best hyperparameters after validation for the different algorithms on ResNet18 with BN layers for classification of CIFAR10 images.

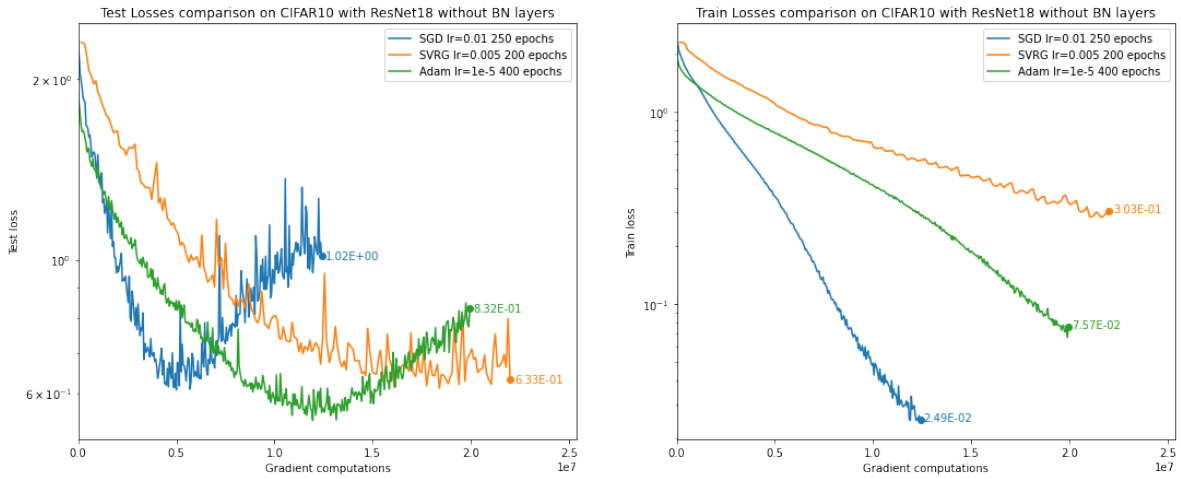

Figure 4.9: Test loss during training of SGD, Adam and SVRG with hyperparameters selected on the validation set for classification of the CIFAR10 dataset on a ResNet18 architecture without batch-normalization layers. The x-axis is in terms of gradient computations. Train losses are presented on the right plot.

test loss in this setting is not as good as in the setting when we do have batch-normalization layers.

4.2.3 ResNet101 with batch-normalization layers

Setup

Architecture To test our hypothesis even further we do a similar approach on a deeper neural network. To do so we choose the ResNet101 architecture. This network is similar to ResNet18 except that is has more layers stacked one after the other. A figure from the original paper presenting the ResNet architecture [He et al., 2015] summarizes the differences and is presented in Figure 4.10.

Algorithm	Test accuracy	Train accuracy
SGD	0.8382	0.9924
ADAM	0.8268	0.9712
SVRG	0.8044	0.8990

Table 4.8: Final test and train accuracies after training of SGD, ADAM and SVRG with hyper-parameters selected after validation for classification on **CIFAR10**, using ResNet18 without batch-normalization layers.

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

Figure 4.10: Different variants of the ResNet architecture presented in the original paper. Building blocks are shown in brackets, with the numbers of blocks stacked. Downsampling is performed by conv3_1, conv4_1, and conv5_1 with a stride of 2.

Dataset For this new architecture we perform a similar classification task but on a bigger dataset with more classes, CIFAR100 [Krizhevsky, 2009]. Some images from this dataset are presented in Figure 4.11.

Input transformation For this task, we perform the same transformations and preprocessing as with CIFAR10. We first pad the images with 4 pixels on each side, then perform a random square crop of size 32×32 and then perform a horizontal flip with probability 0.5 and finally we normalize each channel (reg, green and blue) with the means and standard deviations as calculated from the training set.

Validation of hyperparameters Then we repeat the same steps as we did for ResNet18. We first use a validation set of 5% to select the best hyperparameters and then compare on the test set the cross entropy loss with the best tuned hyperparameter for Adam, SGD and SVRG. The hyperparameters we checked in the validation phase are presented in Table 4.9.

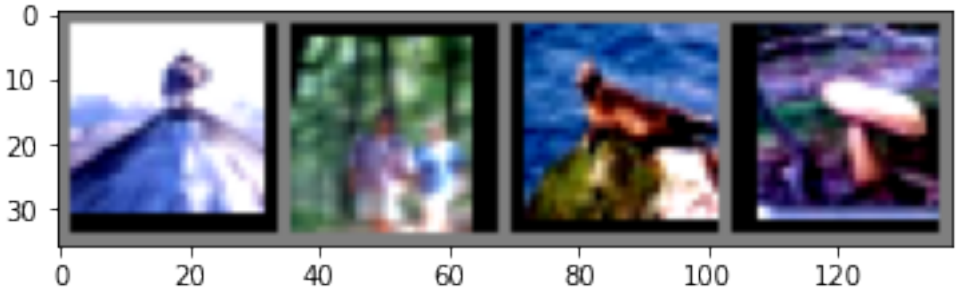

Figure 4.11: 4 samples from the CIFAR100 image dataset containing images of 100 classes. The 4 samples here are mountain, forest, seal and mushroom.

Algorithm	Hyperparmeter	Considered values
Adam	η	$\{10^{-6}, 10^{-5}, 10^{-4}\}$
SGD	η	$\{0.005, 0.01, 0.05\}$
SVRG	η	$\{0.0005, 0.010, 0.005\}$

Table 4.9: Hyperparameters tested for validation for the different algorithms on ResNet101 for classification of CIFAR100 images.

Results

Selected hyperparameters We notice that we usually need similar learning rates than the smaller ResNet18 network, as shown in Table 4.10.

Results on test set The losses on the test and train sets are presented in Figure 4.12 and final accuracies are shown in Table 4.13. It appears that SVRG with best-tuned hyperparameters has results that are consistently worse than both Adam and SGD at any point during training. However, we can note that the difference is not extreme between the variance reduced method and the others.

Algorithm	Hyperparmeter	Selected value
Adam	η	10^{-4}
SGD	η	0.05
SVRG	η	0.005

Table 4.10: Best hyperparameters after validation for the different algorithms on ResNet101 for classification of CIFAR100 images.

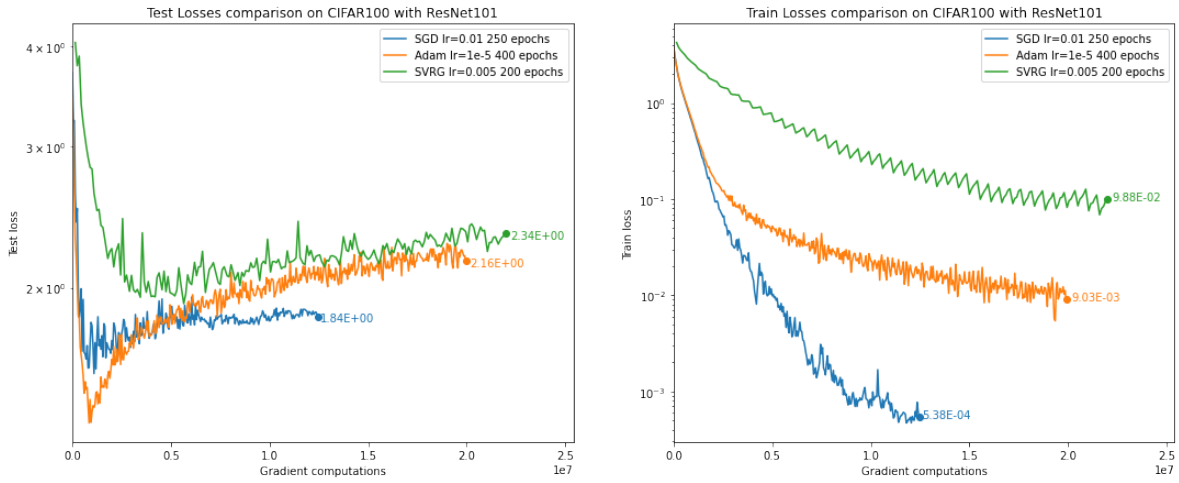

Figure 4.12: Test loss during training of SGD, Adam and SVRG with hyperparameters selected on the validation set for classification of the **CIFAR100** dataset on a **ResNet101** architecture with batch-normalization layers. Train losses are presented on the right plot. The x-axis is gradient computations.

Algorithm	Test accuracy	Train accuracy
SGD	0.7239	0.9999
ADAM	0.7073	0.9984
SVRG	0.5947	0.9734

Table 4.11: Final test and train accuracies after training of SGD, ADAM and SVRG with hyperparameters selected after validation for classification on **CIFAR100**, using ResNet101.

4.2.4 ResNet101 without batch-normalization layers

Setup

We perform the exact same setup as what we did above with ResNet101 and CIFAR100, except that this time we remove the batch-normalization layers to test once more the hypothesis we formulated at the beginning. We check the same hyperparameters on the validation set (see Table 4.9).

Results

Selected hyperparameters The best learning rate for each algorithm is presented in Table 4.12.

Algorithm	Hyperparmeter	Selected value
Adam	η	10^{-4}
SGD	η	0.005
SVRG	η	0.005

Table 4.12: Hyperparameters tested for validation for the different algorithms on ResNet101 without batch-normalization layers for classification of CIFAR100 images.

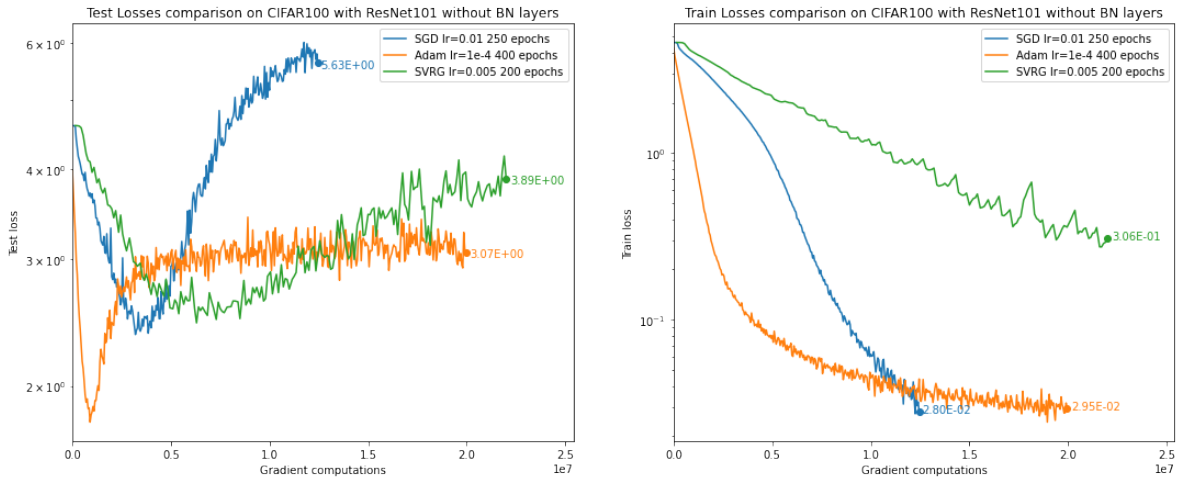

Figure 4.13: Test loss during training of SGD, Adam and SVRG with hyperparameters selected on the validation set, on **CIFAR100** using the **ResNet101** architecture, without batch-normalization layers. Train losses are presented on the right. X-axis is gradient computations.

Results on test set The results on the test and train set are presented in Figure 4.13 as well as in Table 4.13. We see that in this setup SVRG seems to have similar and sometimes better results than the other algorithms, which was not the case with ResNet101 with batch-normalization layers.

4.3 Effects of MetaInit

4.3.1 Setup

Initialization in previous experiments In the previous experiments we see that it seems that SVRG gets comparable and even sometimes better results than Adam and SGD in the setup without batch-normalization. However, we note that all results of all the 3 algorithms worsen a little bit relative to their respective results in the case when we use batch-normalization layers. This might be due to effects of the batch-normalization layers, which help reach better

Algorithm	Test accuracy	Train accuracy
SGD	0.4771	0.9925
ADAM	0.6055	0.9934
SVRG	0.4441	0.8915

Table 4.13: Final test and train accuracies after training of SGD, ADAM and SVRG with hyperparameters selected after validation for classification on **CIFAR100**, using ResNet101 without batch-normalization layers.

convergence, that are lost when removing these layers. To counter this we think that using better initialization methods than the ones used by default in Pytorch could get better results in the setup without batch-normalization layers. For now, all experiments we did used the default initialization implemented by Pytorch as of January 2021, some examples are given below. Equation 4.2 for Linear layers with $in\_feature$ input features of the layer and Equation 4.3 for 2 dimensional convolutional layers with C_in input channels and $kernel\_size$ is a tuple containing the size of the kernel in the 2 dimensions and $*$ is the valid 2D cross-correlation operator.

$$\mathcal{U}(-\sqrt{k}, \sqrt{k}), \text{ where } k = \frac{1}{in\_features} \tag{4.2}$$

$$\mathcal{U}(-\sqrt{k}, \sqrt{k}), \text{ where } k = \frac{1}{C\_in * \prod_{i=0}^{1} kernel\_size[i]} \tag{4.3}$$

Architecture, Dataset and hyperparameters To test the effects of weight initialization we use the previously described algorithm MetaInit. We run an experiment with SVRG on ResNet101 without batch-normalization layers with a task of classification of the CIFAR100 dataset.We try different learning rates between 0.005 and 0.1 and run the training of SVRG for 200 epochs. We run the Gradient descent part of the MetaInit algorithm for 200 epochs with a learning rate of 0.1, momentum of 0.9 and $\epsilon = 10^{-5}$.

4.3.2 Results

Results on test set The results on the test set between a run using MetaInit and one not using it are compared in Figure 4.14. The best experiment we have in this setting without MetaInit applied is presented in red and has a learning rate of 0.005. We see that the runs with MetaInit seem to allow for bigger learning rates that can converge faster to a good result, which even outperforms the baseline without MetaInit. However, we can note two things. First,

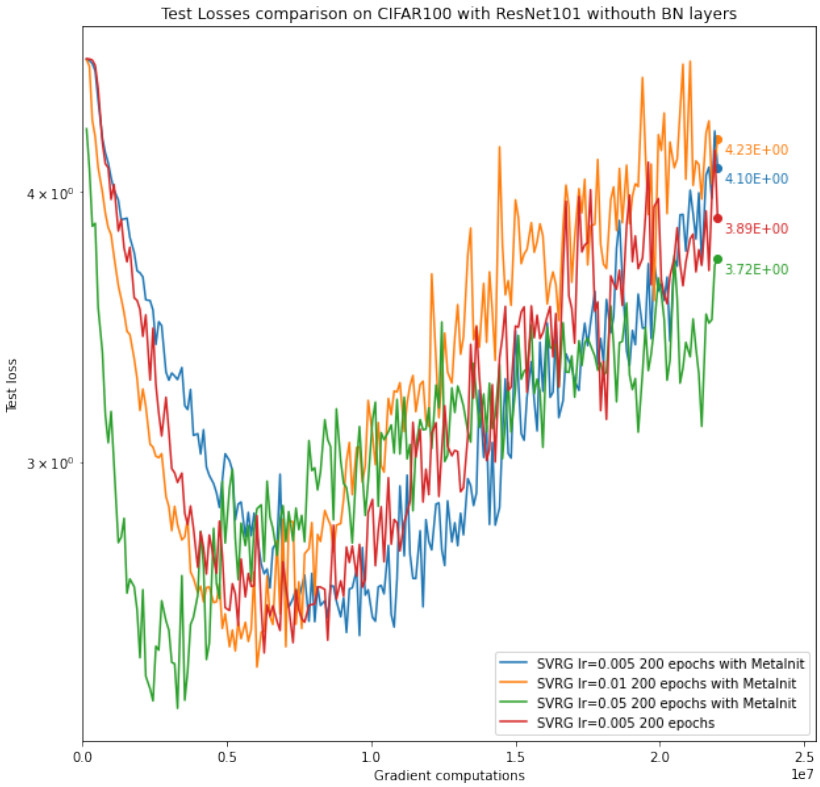

Figure 4.14: SVRG without batch-normalization layers on **CIFAR100**, using **ResNet101**. Comparison of experiments with MetaInit applied and in red the best run with SVRG without MetaInit in the same setting.

we are still limited in the selection of learning rates as learning rates of 0.1 and above do not yield significant results. After a few epochs of training we get NaN values, probably because of exploding or vanishing variants. Second, we can note that experiments with MetaInit do not extremely outperform the baseline without MetaInit. This probably suggests that the default weight initialization used by Pytorch is already performing well on such an architecture and that MetaInit helps mainly for getting faster results in the beginning of training.

Chapter 5

Conclusion

After the various experiments, we see that stochastic variance reduced gradient methods are performing better than Adam and SGD on smaller architectures such as LeNet for classification of MNIST and we see that when applied on common Deep Neural Networks with batch-normalization layers the variance reduced method SVRG is outperformed by Adam and SGD. Because the batch-normalization layers are breaking the assumption of unbiasedness of the updates of SVRG, we test the same setups with architectures where batch-normalization layers have been removed and note that in this case SVRG gets comparable and sometimes better results than Adam and SGD.

However, when doing this we note two things. First, searching for good hyperparameters is harder when batch-normalization layers are removed as we get scenarios with DNNs that get irrelevant results in some hyperparameter setups. Second, we note that the best results reached with all three algorithms (SGD, Adam and SVRG) on the network are not as good as the best results on the same network with batch-normalization layers.

To try to tackle this problem we try using a weight initialization method, MetaInit. This seems to mainly help get good results faster in the beginning of training, by allowing bigger learning rates to be used.

In summary, we seem to observe that SVRG is not disadvantaged as much as the other algorithms when removing batch-normalization layers from common DNNs but there are other unexplained phenomena that make the results somehow worse for all optimization algorithms.

We think that future work can be done in order to perform better hyperparameter selection with ResNet18 and ResNet101 by performing a proper cross-validation procedure. Moreover, we think that doing this cross-validation procedure together with MetaInit can help, as starting training with better choices of initial weights can allow for more favorable learning rates

that are not possible to use when the batch-normalization layers are removed and default initialization is used. Finally we think that experiments should be tested with more DNNs, data-sets and random seeds in order to get more conclusive results.

Acknowledgments

I would like to thank Tatjana who helped me with advice, ideas and suggestions for this project throughout the whole semester. This was especially helpful for me in these special times marked by the COVID pandemic because of which we have not had the chance to meet even once in person. Despite these unfortunate circumstances, Tatjana helped to make this project extremely interesting and enriching for me.

I would like to also thank Pr. Jaggi who took me in his lab and helped me find an interesting project idea. I would also like to thank him for giving me access to different computational resources that were needed for this project.

I would also like to thank Mathias Payer for his available thesis template that I am using here.

Lausanne, February 1, 2021 Alexander Apostolov

Bibliography

A. Cutkosky and F. Orabona. Momentum-based variance reduction in non-convex SGD. *CoRR*, abs/1905.10018, 2019. URL http://arxiv.org/abs/1905.10018.

Y. N. Dauphin and S. Schoenholz. Metainit: Initializing learning by learning to initialize. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32, pages 12645–12657. Curran Associates, Inc., 2019. URL https://proceedings.neurips.cc/paper/2019/file/876e8108f87eb61877c6263228b67256-Paper.pdf.

A. Defazio and L. Bottou. On the ineffectiveness of variance reduced optimization for deep learning. In *NeurIPS*, 2019.

J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(61):2121–2159, 2011. URL http://jmlr.org/papers/v12/duchi11a.html.

K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition, 2015.

S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift, 2015.

R. Johnson and T. Zhang. Accelerating stochastic gradient descent using predictive variance reduction. In C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 26, pages 315–323. Curran Associates, Inc., 2013. URL https://proceedings.neurips.cc/paper/2013/file/ac1dd209cbcc5e5d1c6e28598e8cbbe8-Paper.pdf.

D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. In *ICLR*, 2015.

A. Krizhevsky. Learning multiple layers of features from tiny images. Technical report, 2009.

Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1: 541–551, 1989.

Y. LeCun, C. Cortes, and C. Burges. Mnist handwritten digit database. *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist*, 2, 2010.

M. D. Zeiler. ADADELTA: an adaptive learning rate method. *arXiv:1212.5701*, 2012.