Faster SGD training by minibatch persistency

Matteo Fischetti *
Department of Information Engineering
University of Padova, Italy
matteo.fischetti@unipd.it

Iacopo Mandatelli
Department of Information Engineering
University of Padova, Italy
iacopo.mandatelli@studenti.unipd.it

Domenico Salvagnin
Department of Information Engineering
University of Padova, Italy
domenico.salvagnin@unipd.it

Abstract

It is well known that, for most datasets, the use of large-size minibatches for Stochastic Gradient Descent (SGD) typically leads to slow convergence and poor generalization. On the other hand, large minibatches are of great practical interest as they allow for a better exploitation of modern GPUs. Previous literature on the subject concentrated on how to adjust the main SGD parameters (in particular, the learning rate) when using large minibatches. In this work we introduce an additional feature, that we call *minibatch persistency*, that consists in reusing the same minibatch for K consecutive SGD iterations. The computational conjecture here is that a large minibatch contains a significant sample of the training set, so one can afford to slightly overfitting it without worsening generalization too much. The approach is intended to speedup SGD convergence, and also has the advantage of reducing the overhead related to data loading on the internal GPU memory. We present computational results on CIFAR-10 with an AlexNet architecture, showing that even small persistency values ($K = 2$ or 5) already lead to a significantly faster convergence and to a comparable (or even better) generalization than the standard "disposable minibatch" approach ($K = 1$), in particular when large minibatches are used. The lesson learned is that minibatch persistency can be a simple yet effective way to deal with large minibatches.

1 The idea

Gradient Descent is one of the methods of choice in Machine Learning and, in particular, in Deep Learning [1]. For large training sets, the computation of the true gradient of the (decomposable) loss function is however very expensive, so the *Stochastic Gradient Descent* (SGD) method—in its *on-line* or *minibatch* variant—is highly preferable in practice because of its superior speed of convergence and degree of generalization.

In recent years, the availability of parallel architectures (notably, GPUs) suggested the use of larger and larger minibatches, which however can be problematic because it is known that the larger the minibatch, the poorer the convergence and generalization properties; see, e.g., [13, 8, 5, 9]. Thus, finding a practical way of feeding massively-parallel GPUs with large minibatches is nowadays a hot research topic.

Recent work on the subject pointed out the need of modifying the SGD parameters (in particular, the learning rate) to cope with large minibatches [13, 8, 5, 9], possibly using clever strategies such as

*corresponding author: http://www.dei.unipd.it/~fisch

Preprint. Work in progress.

learning-rate linear scaling and warm-up [2]. All the published works however give for granted that a "disposable minibatch" strategy is adopted, namely: within one epoch, the current minibatch is changed at each SGD iteration.

In this paper we investigate a different strategy that reuses a same minibatch for K (say) consecutive SGD iteration, where parameter K is called *minibatch persistency* ($K = 1$ being the standard rule). Our intuition is that large minibatches contain a lot of information about the training set, that we do not want to waste by dropping them too early. In a sense, we are "slightly overfitting" the current minibatch by investing $K - 1$ additional SGD iterations on it, before replacing it with a different minibatch. The approach also has the practical advantage of reducing the computational overhead related to the operation of loading new data into the GPU memory.

Of course, insisting too much on a same minibatch is a risky policy in terms of overfitting, hence one has to computationally evaluate the viability of the approach by quantifying the pros and cons of the minibatch-persistency idea on some well-established setting.

In the present paper we report the outcome of our computational tests on the well-known CIFAR-10 [6] dataset, using (a reduced version of) the AlexNet [7] architecture, very much in the spirit of the recent work presented in [9]. Our results show that small persistency values $K = 2$ or 5 already produce a significantly improved performance (in terms of training speed and generalization) with minibatches of size 256 or larger. The lesson learned is that the use of large minibatches becomes much more appealing when combined with minibatch persistency, at least in the setting we considered.

2 Experiments

We next report the outcome of some tests we performed to evaluate the practical impact of minibatch persistency on improving training time without affecting generalization in a negative way.

2.1 Setup

As already mentioned, our experimental setting is very similar to the one in [9]. We addressed the CIFAR-10 [6] dataset with a reduced version of the AlexNet [7] architecture involving convolutional layers with stride equal to 1, kernel sizes equal to [11,5,3,3,3], number of channel per layer equal to [64,192,384,256,256], max-pool layers with 2x2 kernels and stride 2, and 256 hidden nodes for the fully-connected layer. As customary, the dataset was shuffled and partitioned into 50,000 examples for the training set, and the remaining 10,000 for the test set.

Our optimization method was SGD with momentum [12] with cross entropy loss—a combination of softmax and negative log-likelihood loss (NLLLoss). The gradient of a minibatch of size m was computed as the sum of the m gradients of the given examples in the minibatch, with respect to the considered loss function. To be more specific, let $L_i(\theta)$ denote the contribution to the loss function of the i-th training example in the minibatch, with respect to the weight vector θ. For every minibatch of size m, the weight update rule at iteration t is as follows:

$$\begin{aligned} \Delta\theta_t &= \sum_{i=1}^{m} \nabla_\theta L_i(\theta_t) \\ v_t &= \gamma v_{t-1} - \mu\, \Delta\theta_t \\ \theta_{t+1} &= \theta_t + v_t. \end{aligned}$$

Training was performed for 100 epochs using PyTorch [10], with learning rate $\mu = 0.001$ and momentum coefficient $\gamma = 0.5$. The initial random seed (affecting, in particular, the random weight initialization) was kept the same for all (deterministic) runs.

All runs have been performed on a PC equipped with a single GPU (Nvidia GTX1080 Ti); reported computing times are in wall-clock seconds.

Our experiments are just aimed at evaluating the impact of different values of the minibatch-persistency parameter K for a given minibatch size m. This is why we decided to use the very basic training algorithm described above. In particular, batch normalization [4], dropout [3] and data augmentation were not used in our experiments. In addition, momentum coefficient and learning rate were fixed a priori (independently of the minibatch size and not viewed as hyper-parameters to tune)

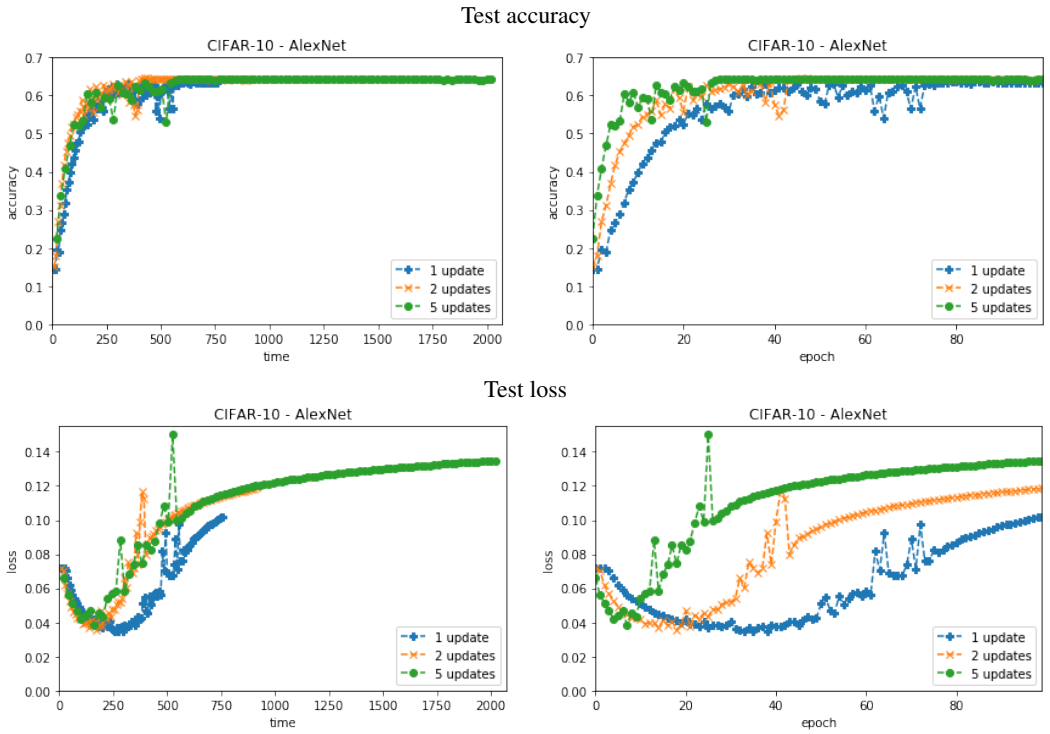

Figure 1: Results with minibatch size $m = 32$ and minibatch persistency K = 1 (standard), 2, and 5.

and used in all the reported experiments; see, e.g., [13] for a discussion about the drawbacks of using a learning rate independent of the minibatch size. As a result, the final test-set accuracies we reached after 100 epochs (about 60%) are definitely not competitive with the state of the art. Of course, better results are expected when using more sophisticated training strategies—some preliminary results in this directions are reported in Section 3.

2.2 Results

Figures 1, 2 and 3 report the results of our experiments for minibatch sizes m = 32, 256 and 512, respectively. Note that, according to [9], the best performance for CIFAR-10 and a (reduced) AlexNet architecture is achieved for $m \leq 8$, while minibatches of size 256 or 512 are considered too large to produce competitive results in the classical setting—as we will see, this is no longer true when minibatch persistency is used.

Each figure plots top-1 accuracy and loss function (both on the test set) for minibatch-persistency parameter $K \in \{1, 2, 5\}$, as a function of the total computing time (subfigures on the left) and of the number of epochs (on the right). Note that, within a single epoch, each training example is evaluated K times, hence one would expect the computing time to perform each epoch be multiplied by K. This is why, to have a fair comparison of different values of K, it is important to report computing times explicitly.

Figure 1 refers to a small minibatch of size $m = 32$. Higher values of K give improved accuracy in terms of epochs (subfigure on the right), however this does not translate into an improvement of computing time (subfigure on the left). Indeed, completing the 100 epochs with $K = 5$ took approximately 3 times more time than case $K = 1$ (about 2000 instead of 700 sec.s), which is in any case significantly less than the factor of 5 one would expect. On the other hand, the run with $K = 5$ reached its best accuracy of 0.64450 after 30 epochs and about 600 sec.s, while the run with $K = 1$ reached its best accuracy of 0.63640 only at epoch 80, after 600 sec.s as well. As to loss, the runs with larger K started overfitting much earlier, both in terms of epochs and computing times. This behavior is not unexpected, as the minibatch is too small to be representative of the whole training set, and reusing it several times is prone to overfitting.

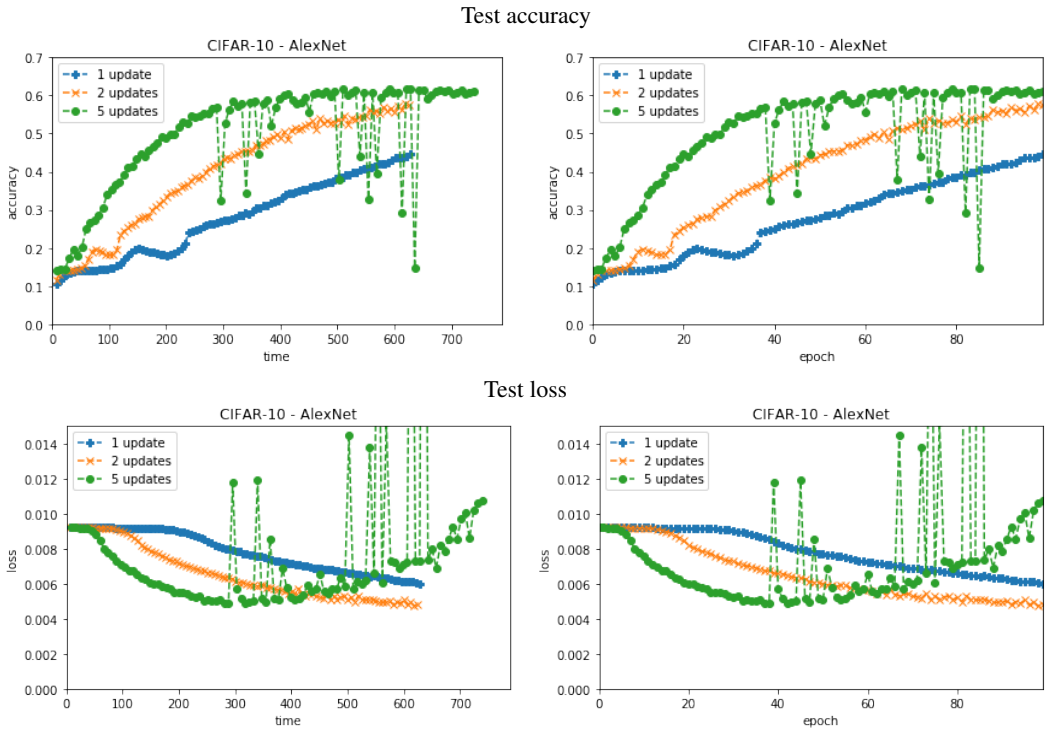

Figure 2: Results with minibatch size $m = 256$ and minibatch persistency K = 1 (standard), 2, and 5.

The overall picture radically changes when a medium-size minibatch with $m = 256$ examples is considered (Figure 2). Here, overfitting was less pronounced, and the computing time to complete the 100 epochs did not increase significantly with the value of K—thus confirming that minibatch persistency has a positive effect in terms of GPU exploitation when the minibatch is not too small.

The positive effect of minibatch persistency is even more evident when larger minibatches come into play (case $m = 512$ of Figure 3). Here, the run with $K = 5$ is the clear winner, both in terms of accuracy and loss.

The above results are rather encouraging, and show that the use of large minibatches becomes much more appealing when combined with minibatch persistency.

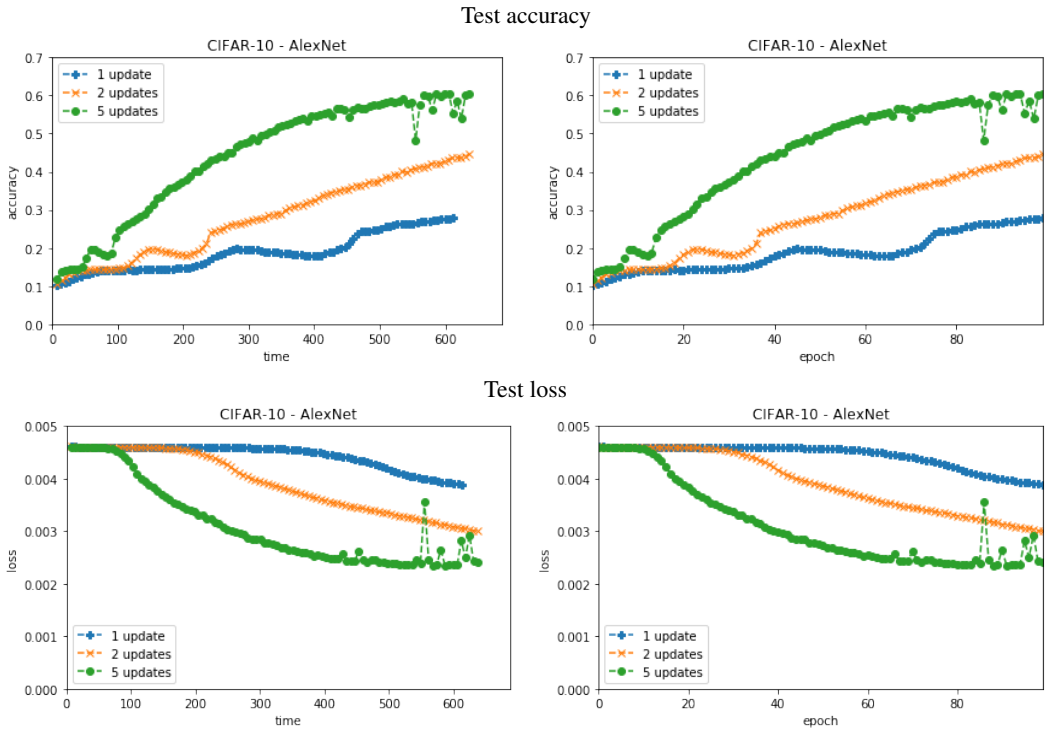

Figure 3: Results with minibatch size $m = 512$ and minibatch persistency $K = 1$ (standard), 2, and 5.

3 Conclusions and future work

We have investigated the idea of reusing, for K consecutive times, a same minibatch during SGD training. Computational experiments indicate that the idea is promising, at least in the specific setting we considered (AlexNet on CIFAR-10). More extensive tests should be performed on different datasets and neural networks, as one may expect different (possibly worse) behaviors for different settings.

In our tests, for the sake of comparison we fixed a same learning rate $\mu = 0.001$ for all values of K. However, we also did some preliminary experiments with a different policy that increases the learning rate when reusing the same minibatch, i.e., a learning rate equal to $k \cdot \mu$ is applied for the k-th use of a same minibatch ($k = 1, \ldots, K$). The rational is that, each time the minibatch is reused, the computed update direction becomes more reliable (at least, for the purpose of optimizing the loss function within the current minibatch). This *adaptive* learning-rate strategy is reminiscent of the Cyclical Learning Rate training policy introduced in [11], but it is tailored for our setting. According to the preliminary results reported in Figures 4 and 5, the approach seems rather effective—a similar behavior has been observed for minibatch size $m = 256$ (not reported). Future work should therefore be devoted to better investigate this (or similar) adaptive strategies.

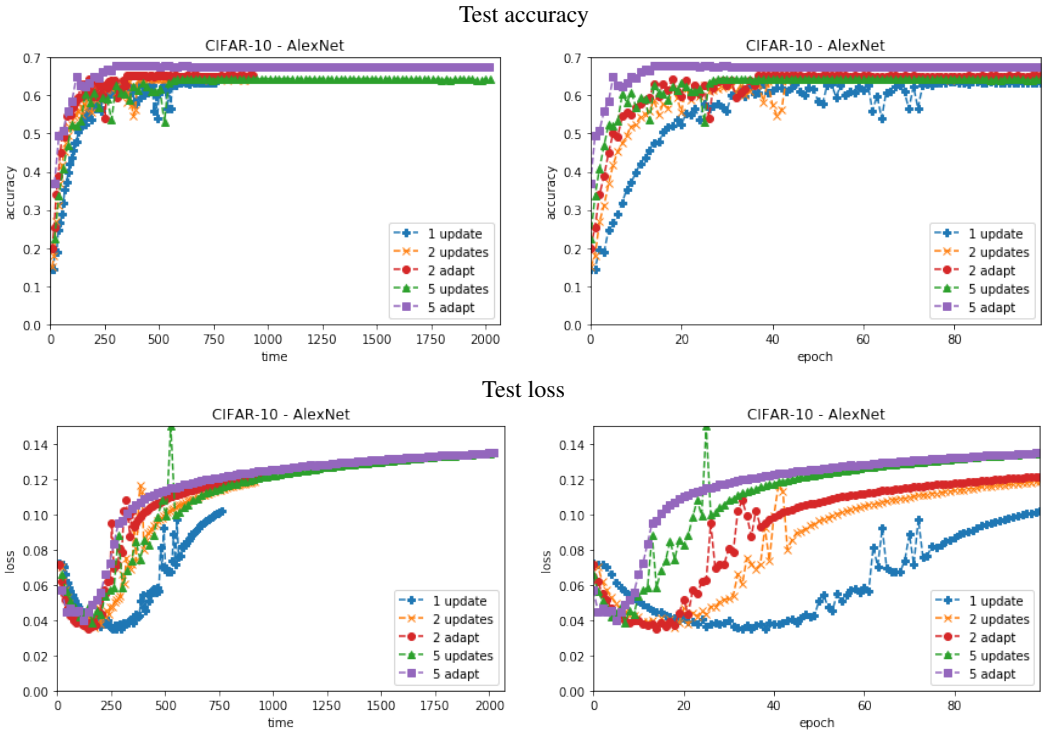

Figure 4: Results with minibatch size $m = 32$, with and without adaptive learning rate.

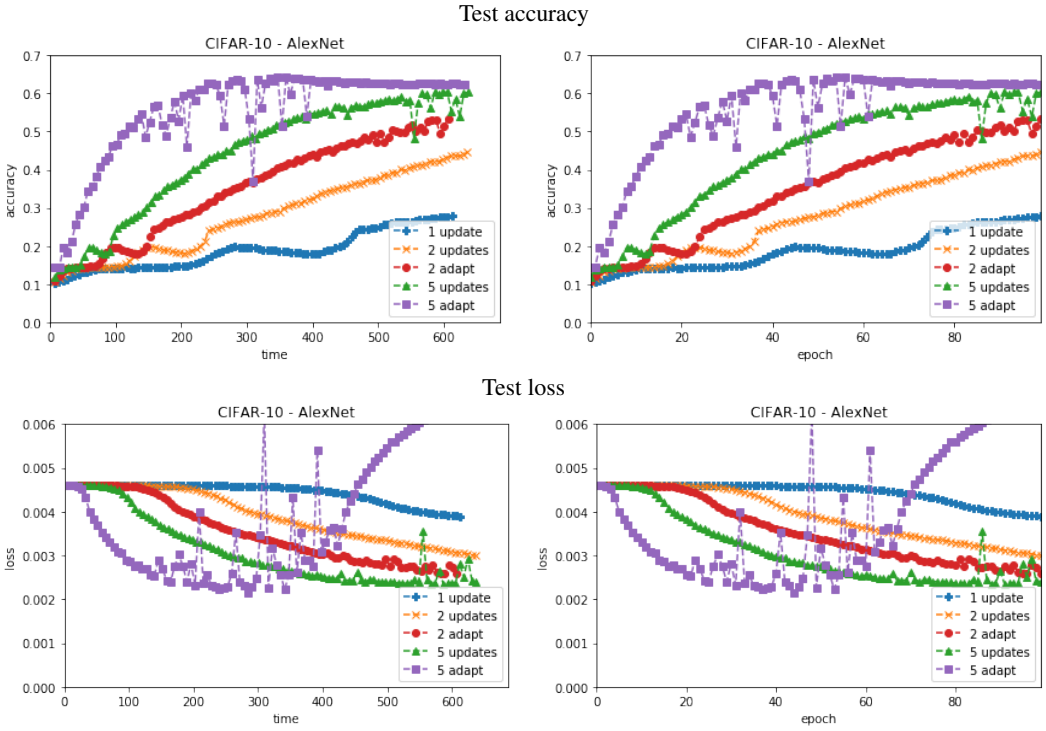

Figure 5: Results with minibatch size $m = 512$, with and without adaptive learning rate.

References

[1] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[2] Priya Goyal, Piotr Dollár, Ross B. Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch SGD: training Imagenet in 1 hour. *arXiv preprint, CoRR*, abs/1706.02677, 2017.

[3] Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint, CoRR*, abs/1207.0580, 2012.

[4] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint, CoRR*, abs/1502.03167, 2015.

[5] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On large-batch training for deep learning: Generalization gap and sharp minima. *arXiv preprint, CoRR*, abs/1609.04836, 2016.

[6] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. CIFAR-10 (Canadian Institute for Advanced Research). *http://www.cs.toronto.edu/~kriz/cifar.html*.

[7] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[8] Yann A. LeCun, Léon Bottou, Genevieve B. Orr, and Klaus Robert Müller. *Efficient backprop*, volume 7700 of *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, pages 9–48. 2012.

[9] D. Masters and C. Luschi. Revisiting Small Batch Training for Deep Neural Networks. *arXiv preprint, CoRR*, abs/1804.07612, April 2018.

[10] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in PyTorch. *31st Conference on Neural Information Processing Systems (NIPS 2017), Long Beach, CA, USA, 2017*.

[11] Leslie N. Smith. No more pesky learning rate guessing games. *arXiv preprint, CoRR*, abs/1506.01186, 2015.

[12] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, ICML'13, pages III–1139–III–1147. JMLR.org, 2013.

[13] D. Randall Wilson and Tony R. Martinez. The general inefficiency of batch training for gradient descent learning. *Neural Networks*, 16(10):1429 – 1451, 2003.