

Stochastic Gradient Descent in Theory and Practice

Panos Achlioptas
Stanford

1 Introduction

Stochastic gradient descent (SGD) is the most widely used optimization method in the machine learning community. Researchers in both academia and industry have put considerable effort to optimize SGD's runtime performance and to develop a theoretical framework for its empirical success. For example, recent advancements in deep neural networks have been largely achieved because, surprisingly, SGD has been found adequate to train them. Here we present three works highlighting desirable properties of SGD. We start with examples of experimental evidence [1] for SGD's efficacy in training deep and recurrent neural networks and the important role of acceleration and initialization. We then turn into theoretical work [2] connecting a model's trainability by SGD to its generalization. And, finally, we discuss a theoretical analysis [3] explaining the dynamics behind the recently introduced versions of asynchronously executed SGD.

1.1 First Order Methods

For an arbitrary scalar function $f : \Omega \rightarrow \mathbb{R}$ consider the general unconstrained optimization problem

$$\arg \min_{w \in \Omega} f(w) . \tag{1}$$

Iterative methods attempt to find a solution by generating a sequence of solutions w_0, w_1, \dots . First order methods are constrained, by definition, to generate this sequence by considering only the function's value and gradient at different points of Ω (possibly other than the iterates). Thus, first order methods are appropriate only when f is at least (sub)-differentiable.

Gradient descent (GD) is arguably the simplest and most intuitive first order method. Since at any point w the direction $-\nabla_w f$ (the antigradient) is the direction of the fastest decrease of f at w , GD starts from a randomly chosen point $w_0 \in \Omega$ and generates each next point by applying the update

$$w_{t+1} = w_t - \alpha_t \nabla_{w_t} f , \tag{2}$$

where $\alpha_t \geq 0$ is an appropriately chosen "step size". Sometimes we will express (2) more compactly as the function (update) $G_{f,\alpha} : \Omega \rightarrow \Omega$

$$G_{f,\alpha}(w) = w - \alpha \nabla_w f .$$

Since GD is a local, greedy algorithm without any further assumptions, at best, it can be expected to converge only to a local minimum of f . Another issue is that the gradient computation itself may be quite demanding computationally. A very common scenario where this is the case is supervised learning, a general framework for which we present below.

1.2 Supervised Learning by Gradient Descent

Consider an *unknown* distribution \mathcal{D} over elements of a space $\mathcal{Z} = \mathcal{X} \times \mathcal{Y}$. Commonly, \mathcal{X} is the space of input *features*, characterizing some aspects of the instances of the learning problem, and \mathcal{Y} is the space of their associated *labels* ("correct answers"). At a high level, the goal is to construct a model, i.e., to select a

model from a family of models, which approximates the conditional distribution on labels given the features. That is, ideally, we would like to find the value of the model parameter, w , minimizing the *population risk*,

$$R(w) = \mathbb{E}_{z \sim \mathcal{D}} \ell(w; z) \text{ ,} \quad (3)$$

where $\ell(w; z)$ denotes the loss of our model on example z when the model parameter is w . Naturally, the choice of loss function is crucial and several alternatives with different properties exist (see [11] for an in-depth treatment), but this is not something that will concern us here. But since the distribution \mathcal{D} is unknown, we assume that we are given a sample $S = (z_1, \dots, z_n)$ of examples drawn i.i.d. from \mathcal{D} and instead of $R(w)$ we use its sample-average proxy, the *empirical risk*,

$$R_S(w) = \frac{1}{n} \sum_{i=1}^n \ell(w; z_i) \text{ .} \quad (4)$$

If we now attempt to “learn”, i.e., to minimize R_S by GD, we find that the gradient of empirical risk depends on all n examples. And since “more data is better” we find ourselves running into a wall.

The key out of this bind is the fundamental observation that empirical risk is a sum. Therefore, its gradient is a vector that results by summing n gradient-vectors, one per example. Moreover, it is reasonable to assume that no matter where we are in parameter space, no particular example exerts too much of an influence on the gradient. As a result, it is reasonable to believe that we can get a good approximation of the gradient at any given point in parameter space by taking a random subset of b examples, adding their gradient vectors, and scaling the result. This stochastic process for estimating the gradient gives rise to *Stochastic Gradient Descent* (SGD).

There are obviously several still-unspecified issues such as what is a good value of b , and whether sampling should be done with replacement or not. We will not be addressing such issues here, other than to say that sampling without replacement is generally better and can be implemented by applying a random permutation to the n examples and then selecting the next b of them for each gradient estimation. Naturally, for $b = n$ we recover GD, and the rest of the paper is organized so that it can be read with this mindset, except when we specifically draw attention to batch formation.

Before delving in further discussion regarding (S)GD in the next section we perform a quick review of relevant mathematics and known facts about the methods. Throughout, our silent assumption is that we are trying to minimize a *differentiable* function $f : \Omega \rightarrow \mathbb{R}$.

2 Some Basic Facts

2.1 Basic Geometric Properties of Functions

Convexity. A function f is convex if $f(u) \geq f(v) + \langle \nabla f(v), u - v \rangle$ for all $u, v \in \Omega$.

In other words, f is convex if its linear approximation is a global underestimator of its value.

Strong Convexity. A function f is γ -strongly convex for $\gamma > 0$, if for all $u, v \in \Omega$,

$$f(u) \geq f(v) + \langle \nabla f(v), u - v \rangle + \frac{\gamma}{2} \|u - v\|^2 \text{ .} \quad (5)$$

Whereas a linear function is convex, a strongly convex function “must bend”. As an example, if f is twice continuously differentiable and its domain is the real line., then convexity is equivalent to $f'' \geq 0$, while strong convexity is equivalent to $f'' \geq \gamma$. We will employ the definitions of Lipschitzness and Smoothness below with respect to the Euclidean norm over Ω . However, they can be defined with respect to any norm.

Lipschitzness. A function f is L -Lipschitz if $\|\nabla f\| \leq L$, implying $|f(u) - f(v)| \leq L\|u - v\|$ for all $u, v \in \Omega$.

Smoothness. A function f is β -smooth if its gradient is β -Lipschitz, implying that for all $u, v \in \Omega$,

$$f(u) \leq f(v) + \langle \nabla f(v), u - v \rangle + \frac{\beta}{2} \|u - v\|^2 \text{ .} \quad (6)$$

For a smooth and strongly convex function inequalities (5) and (6) provide both upper and lower bounds on the difference between the function and its first order approximation. Furthermore, if f is twice continuously differentiable in Ω , being β -smooth and γ -strongly convex is equivalent to

$$\beta I \succeq \nabla^2 f(u) \succeq \gamma I \quad , \quad \text{for all } u \in \Omega \quad (7)$$

where ∇^2 is the Hessian of f and I is the identity matrix of the appropriate dimensions (and where $A \succeq B$ denotes that $A - B$ is a positive semidefinite).

2.2 Convergence of Gradient Descent

- If the gradient update is contractive, i.e., there is $c < 1$ such that $\|G_{f,a}(w_1) - G_{f,a}(w_2)\| \leq c\|w_1 - w_2\|$ for any w_1, w_2 , then GD converges to a stationary point w , i.e., $\nabla_w f = 0$.
- If f is convex and β -smooth, and a step size $\alpha \leq 2/\beta$ is used, then the t -th iterate, w_t , of GD satisfies

$$|f(w_t) - f(w_*)| \leq \frac{\|w_0 - w_*\|^2}{2\alpha t} \quad , \quad (8)$$

where w_* is the global minimizer of f . Thus, GD has an $O(1/t)$ rate of convergence.

- If f is γ -strongly convex and β -smooth, and a step size $\alpha \leq \frac{2}{\beta + \gamma}$ is used, then GD converges exponentially fast, with rate $O(c^t)$, for a $0 < c < 1$. The constant c is an increasing function of β/γ .

It is worth noting how dramatically strong convexity helps. Namely, while for a generic convex smooth function GD makes linear progress, when strong convexity is also present it converges *exponentially* faster.

3 Accelerated SGD for Deep Learning

We start with experimental findings demonstrating the effectiveness of SGD even in highly non-convex settings. Concretely, we will see that *accelerated* versions of SGD can greatly improve the speed and the quality of the obtained local minima when training deep and recurrent neural networks. We start with some theoretical observations on the effect of acceleration in standard (non-stochastic) gradient descent (GD). Armed with the intuitions provided by the theory, we then look at experimental results with SGD.

3.1 Polyak’s Classical Momentum

The first method for accelerating GD was introduced in 1964 by Polyak. His method, which we refer to as classical momentum (CM), emphasizes the directions that persistently reduce the objective function across iterations. Concretely, letting α be the learning-step as before, the update equations of CM are

$$w_{t+1} = w_t + v_{t+1} \quad (9)$$

$$v_{t+1} = \mu v_t - \alpha \nabla_{w_t} f \quad . \quad (10)$$

The newly introduced *velocity* vector v_t acts as a memory that accumulates the directions of reduction that were chosen in the previous t steps, while the influence of this memory is controlled by $\mu \in [0, 1]$, which we will call the *momentum coefficient*. Notice that if $\mu = 0$ we recover GD.

A key problem that hinders the success of GD and which motivated the invention of CM is “zig-zagging”. Consider how GD typically behaves in a long shallow ravine leading to the optimum, with steep walls on the sides (it helps to think of the ravine as an oblong quadratic). Since the negative gradient will primarily point across to the opposing wall, GD will tend to oscillate across the narrow ravine, making slow progress towards the minimum (Fig 1, left). By memorizing, through the velocity vector, the direction where the gradient signal has been consistent over the iterations CM helps. Naturally, this is also the direction where

the gradient changes slowly or, equivalently, where the curvature is low. Emphasizing the dimensions of low curvature when minimizing a function is also the basic strategy of second order methods that consult the Hessian to find the next iterate. In general, such methods reweigh the update along each eigendirection of the Hessian by the inverse of the associated eigenvalue. And just as second-order methods enjoy faster convergence to the minimum of a strongly convex function, if we pick α and μ optimally, CM converges in \sqrt{R} -fewer iterations than an optimally tuned GD, where R is the condition number of the Hessian of f .

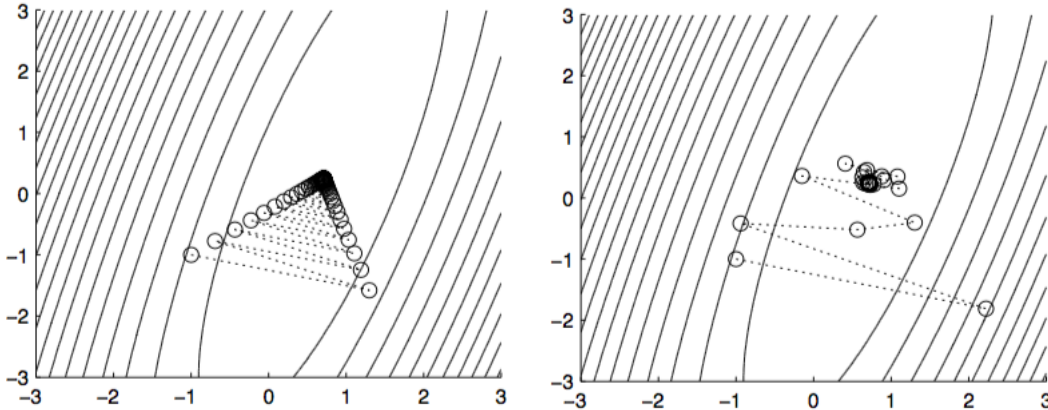


Figure 1: Zig-Zagging problem of GD. **(Left)** Contour lines of an oblong quadratic and the trajectory of GD starting at $(-1,-1)$. **(Right)** Trajectory of CM on same problem. The step size and the momentum coefficient were tuned to achieve optimal theoretical convergence for both methods.

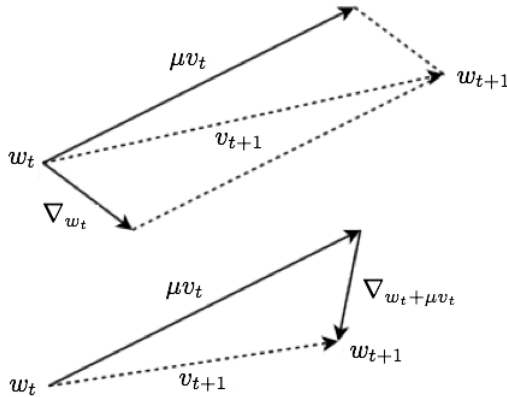


Figure 2: **(Top)** Classical Momentum. **(Bottom)** Nesterov's Accelerated Gradient.

3.2 Nesterov's Accelerated Gradient

The second accelerated method that we will consider is Nesterov's Accelerated Gradient (NAG), introduced in 1983. The update equations of NAG are

$$w_{t+1} = w_t + v_{t+1} \tag{11}$$

$$v_{t+1} = \mu v_t - \alpha \nabla_{w_t + \mu v_t} f . \tag{12}$$

Equations (9), (10) are identical to equations (11), (12) except for a single, seemingly benign, difference: while CM updates the velocity vector by inspecting the gradient at the current iterate w_t , NAG updates it

by inspecting the gradient at $w_t + \mu v_t$ (see Fig 2 for a visual demonstration). To make an analogy, while CM faithfully trusts the gradient at the current iterate, NAG puts less faith into it and looks ahead in the direction suggested by the velocity vector; it then moves in the direction of the gradient at the look ahead point. If $\nabla_{w_t + \mu v_t} f \approx \nabla_{w_t} f$, then the two updates are similar. But if not, this is an indication of curvature and the NAG update has, correctly, put less faith in the gradient. This small difference, compounded over the iterations gives the two methods distinct properties, allowing NAG to adapt faster and in a more stable way than CM in many settings, particularly for higher values of μ . As a result, NAG also enjoys provably faster convergence in certain settings. Concretely, for any convex, smooth function, i.e., not necessarily strongly convex, optimally tuned NAG converges at a $O(1/t^2)$ rate, while GD converges at a rate of $O(1/t)$.

3.3 Relationship between CM and NAG

To examine the above intuitive ideas more closely, [1] analyzed CM and NAG when applied to minimize a positive definite quadratic objective. That is, for any $b \in \mathbb{R}^n$ and $A \succeq 0$, they investigated the performance of the two algorithms on the problem

$$\arg \min_{w \in \mathbb{R}^n} w^T A w + b^T w .$$

By expressing this problem in the eigenvector basis of A , it was shown that the difference of the two methods can be captured in the way they *scale* the velocity vector at every iteration. If we fix the momentum coefficient controlling both methods to be μ and denote by \tilde{v}_t the velocity vector expressed in the eigenbasis of A , then CM scales each dimension of \tilde{v}_t by μ before computing \tilde{v}_{t+1} . NAG, on the other hand, scales the i -th coordinate of \tilde{v}_t by $\mu(1 - \lambda_i \alpha)$, where λ_i is the associated i -th eigenvalue (curvature) of that direction. Thus, for an appropriately small step size α , i.e., so that $\alpha \lambda_i \ll 1$ for every i , the two methods become identical. On the other hand, for large step sizes NAG uses smaller effective momentum for the high-curvature eigen-directions, something that prevents oscillations and allows the use of a larger μ than what would be possible with CM for fixed α .

3.4 Experiments on Autoencoders

The task of a neural network autoencoder is to reconstruct its input, i.e., f is the identity. The catch is that this is to be achieved while at least one of its hidden layers has dimension much smaller than the dimension of the input. As a result, at the end of training, the “bottleneck” layer acts as a low-dimensional code for (is a non-linear embedding of) the input. Autoencoders that have more than six layers are, generally, considered deep neural networks (DNNs) and were considered impossible to train with first order methods before the groundbreaking work [7] that introduced the idea of a greedy layer-wise pre-training. At a high-level, pre-training is the task of training the autoencoder’s layers in sequence using an auxiliary objective, and then fine-tuning by training the entire network using the original objective. For each optimization subtask standard techniques like SGD are used. After [7], Martens [8] introduced a truncated-Newton method, called Hessian-Free (HF) optimization which given an appropriate initialization but without using pre-training was shown to achieve superior results than [7]. Finally, in [1] the authors demonstrated experimentally that one can attain results similar to those of HF, also without pre-training, but with a much simpler algorithm, namely SGD with momentum¹. In summary, the findings of [1] are:

- With good initialization and momentum schedule SGD achieves results similar to [8].
- Without a good initialization, momentum alone cannot drive SGD’s performance to high standards.
- SGD with NAG momentum is more robust than with CM momentum.

¹Here and throughout when we refer to accelerated versions of SGD. These are to be understood as the corresponding accelerated GD method where exact gradient computation has been replaced by a stochastic estimate.

3.4.1 Experimental Setup

To ease comparisons with the HF method, the authors of [1] performed experiments on the same three deep auto-encoders that were presented in [7] and [8]. These networks have 7 – 11 hidden layers and we refer to them as: Curves, MNIST and Faces respectively, according the data set that was used to train them (see [7] for more details). The momentum schedule that was explored in [1] was

$$\mu_t = \min(1 - 2^{-1 - \log_2(\lfloor t/250 \rfloor + 1)}, \mu_{\max}) ,$$

where μ_{\max} was chosen from $\{0.999, 0.995, 0.99, 0.9, 0\}$. For each choice of μ_{\max} they report the learning step that achieved the *best* training error, after a fixed number of iterations. The set of learning steps that they tried was $\{0.05, 0.01, 0.005, 0.001, 0.0005, 0.0001\}$. Finally, they initialized the networks according to the sparse initialization technique (SI) described in [8]. According to SI each hidden unit is connected to 15 randomly chosen units of the previous layers with weights that are drawn from a unit Gaussian and the biases are set to zero. Under this setup, their results are given in Table 1.

task	$0_{(SGD)}$	0.9N	0.99N	0.995N	0.999N	0.9M	0.99M	0.995M	0.999M	SGD_c	HF
Curves	0.48	0.16	0.096	0.091	0.074	0.15	0.10	0.10	0.10	0.16	0.058
Mnist	2.1	1.0	0.73	0.75	0.80	1.0	0.77	0.84	0.90	0.9	0.69
Faces	36.4	14.2	8.5	7.8	7.7	15.3	8.7	8.3	9.3	NA	7.5

Table 1: The table presents the training error for each autoencoder, for each combination of μ_{\max} and a momentum type (NAG, CM). For each value of μ_{\max} , the best-performing learning rate is used. When $\mu_{\max} = 0$, NAG = CM = SGD. The column SGD_c lists the results of Chapelle et al. [9] who used 1.7M SGD steps and tanh networks. Column HF lists the results of HF without L2 regularization.

First we see that both CM and NAG do significantly better than simple SGD. In fact, the best results among these methods are achieved by NAG and are very close to those achieved by HF. Second, we see that larger values of μ_{\max} tend to achieve better performance and that NAG outperforms CM, especially when μ_{\max} is 0.995 or 0.999. To demonstrate the influence the random initialization has for the training, [1] presented results showing how changing the standard deviation of the weights used in SI, alters the final achieved error. As we see in Table 2, using Gaussians with variance 0.25, 0.5, or 4 result in very large training error relative to using variance 1 or 2, demonstrating an extremely fragile relationship between the initialization weights and the final model.

SI scale multiplier	0.25	0.5	1	2	4
error	16	16	0.074	0.083	0.35

Table 2: Training error of "Curves" as a function of the variance of the initial weights.

3.5 Experiments on Recurrent Neural Networks

A recurrent neural network (RNN) is the temporal analogue of a DNN. Specifically, it belongs in a class of artificial neural networks where connections between units form a directed cycle. This creates an internal state for the network which allows it to model complex temporal behavior. Unlike feedforward deep neural networks, RNNs can use their internal memory to process arbitrary input *sequences*. In fact, RNNs can be viewed as very deep networks that have a "layer" for each element of the input sequence. With long-range dependencies between the sequence's time steps, RNNs were considered even harder to train with first order methods than DNNs. For instance, research [10] that dates back to 1994, pinpoints the fundamental problem of the exploding/vanishing gradient and suggests a spectral condition on the weight matrices of the neurons, as a necessary (but not sufficient) condition when training RNNs with SGD. Besides the experiments on DNNs reported in the previous section, the authors of [1] conducted experiments investigating the efficacy of momentum for training RNNs with long range dependencies. In summary, their results are:

- With a good momentum schedule both CM SGD and NAG SGD achieved levels of training error that were previously attainable only with HF.
- RNNs were found to be more sensitive on the initialization of the weights than DNNs.
- Higher values of momentum lead to better accuracy for NAG compared to CM, potentially due to the higher tolerance of NAG in μ , as described in 3.3.

Table 3.5 summarizes the training error of NAG and CM on the various RNN problems in [1].

problem	biases	SGD	0.9N	0.98N	0.995N	0.9M	0.98M	0.995M
add T=80	0.82	0.39	0.02	0.21	0.00025	0.43	0.62	0.036
mul T=80	0.84	0.48	0.36	0.22	0.0013	0.029	0.025	0.37
mem-5 T=200	2.5	1.27	1.02	0.96	0.63	1.12	1.09	0.92
mem-20 T=80	8.0	5.37	2.77	0.0144	0.00005	1.75	0.0017	0.053

Table 3: Each column reports the training error on different problems for some $\mu \in \{0.9, 0.98, 0.995\}$ and momentum type (NAG, CM) combination, averaged over 4 different random seeds. The biases column lists the error of an RNN that failed to establish communication between its inputs and targets. For each , the above results stem from using the learning rate that gave the best performance (see [1] for details).

4 Generalization of Iterative Methods

The *generalization error* of a model parameterized by w is the difference between its empirical and population risk, i.e., $R_S(w) - R(w)$. When the parameter $w = A(S)$ is chosen as a function of the data by a potentially randomized algorithm A it makes sense to consider the expected generalization error, i.e.,

$$\epsilon_{gen} = \mathbb{E}_{S,A}[R_S[A(S)] - R[A(S)]] , \quad (13)$$

where the expectation is over the randomness of A and the sample $S = (z_1, \dots, z_n)$ of the training examples.

In order to bound the generalization error of an algorithm, [2] adapted the original definition of stability [4], making it applicable for randomized algorithms, as follows.

Definition 1. A randomized algorithm A is ϵ -uniformly stable if for all data sets S, S' that differ in at most one example,

$$\sup_z \mathbb{E}_A[f(A(S); z) - f(A(S'); z)] \leq \epsilon. \quad (14)$$

We will denote by $\epsilon_{stab}(A, n)$ the infimum over all ϵ for which (14), omitting (A, n) when clear from context.

Remark 2. Note that in (14), the expectation is only over the internal randomness of A .

Now, we state a fundamental theorem linking the stability of an algorithm to its generalization error.

Theorem 3 (Generalization in expectation). *If algorithm A is ϵ -uniformly stable, then*

$$|\mathbb{E}_{S,A}[R_S(A(S)) - R(A(S))]| \leq \epsilon. \quad (15)$$

The proof of the theorem is a clever exchange and averaging argument, presented in Appendix A. To apply it for an iterative method we thus need to consider its stability. For this we will start from general considerations concerning updates that map a point $w \in \Omega$ in the parameter space to another point $G(w)$. In particular, the following two definitions are key for the analysis.

Definition 4. An update rule is σ -bounded if $\sup_{u \in \Omega} \|u - G(u)\| \leq \sigma$.

Definition 5. An update rule is ν -expansive if

$$\sup_{u,v \in \Omega} \frac{\|G(u) - G(v)\|}{\|u - v\|} \leq \nu .$$

In terms of σ -boundedness and ν -expansiveness the following hold for the GD update.

Lemma 6. If f is L -Lipschitz, then $G_{f,\alpha}$ is (αL) -bounded.

Lemma 7. If f is β -smooth, then $G_{f,\alpha}$ is $(1 + \alpha\beta)$ -expansive. If f is also convex, then $G_{f,\alpha}$ is 1-expansive for $\alpha \leq 2/\beta$. If f is also γ -strongly convex, then $G_{f,\alpha}$ is $(1 - \frac{2\alpha\beta\gamma}{\beta+\gamma})$ -expansive for $\alpha \leq \frac{2}{\beta+\gamma}$, i.e., $G_{f,\alpha}$ is contractive.

At the same time, the following lemma bounds the divergence of two arbitrary sequence of updates that start at the same point, in terms of the σ -boundedness and ν -expansiveness of the updates.

Lemma 8 (Growth recursion). Fix arbitrary sequences of updates G_1, \dots, G_t and G'_1, \dots, G'_t . Let $w_0 = w'_0$ and define $\delta_t = \|w'_t - w_t\|$ where w_t, w'_t are defined recursively via $w_{t+1} = G_t(w_t)$ and $w'_{t+1} = G'_t(w'_t)$. Then,

$$\begin{aligned} \delta_0 &= 0 \\ \delta_{t+1} &\leq \begin{cases} \eta\delta_t & \text{if } G_t = G'_t \text{ is } \eta\text{-expansive} \\ \min(\eta, 1)\delta_t + 2\sigma_t & \text{if } G_t \text{ and } G'_t \text{ are } \sigma\text{-bounded and } G_t \text{ is } \eta\text{-expansive} \end{cases} \end{aligned} \quad (16)$$

4.1 Stability of Stochastic Gradient Descent

Given n labeled examples $S = (z_1, \dots, z_n)$ where $z_i \in \mathcal{Z}$, consider the objective function

$$f(w) = \frac{1}{n} \sum_{i=1}^n \ell(w; z_i) ,$$

where $\ell(w; z_i)$ denotes the loss of w on the example z_i . If we try to minimize f with SGD with batch size of 1, in the t -th update we are picking a single example i_t uniformly. For simplicity of exposition we will assume that this choice is done *with* replacement, so that the chosen examples are i.i.d., even though the results of [2] hold even when the sampling is done without replacement. Denoting the learning rate at time t as $\alpha_t > 0$, we see that the stochastic gradient update is

$$w_{t+1} = w_t - \alpha_t \nabla_{w_t} \ell(w_t; z_{i_t}) . \quad (17)$$

We are now ready to state the results of [2] regarding the stability of SGD and give a sense of the proofs.

4.1.1 Convex Optimization

We begin with a simple stability bound for convex loss minimization via SGD.

Theorem 9. Assume that the loss function $\ell(\cdot; z) \in [0, 1]$ is β -smooth, convex and L -Lipschitz for all z . Then SGD with step sizes $\alpha_t \leq 2/\beta$ for t steps is ϵ -uniformly stable for

$$\epsilon_{stab} \leq \frac{2L^2}{n} \sum_{t=1}^t \alpha_t . \quad (18)$$

We present the proof of Theorem 9 in Appendix B. The idea behind the proof is simply to consider the two sequences of updates that correspond to running SGD from the same initial conditions on two sample sets S, S' that differ on a single point. In particular, in the typical case where the chosen example is the same we apply the case $G_t = G'_t$ of Lemma 8, whereas when they differ we apply the other case. Linearity of expectation then bounds the expected growth of $\|w_t - w'_t\|$.

4.1.2 Strongly Convex Optimization

In the strongly convex case we can bound stability with no dependence on the number of steps at all. The proof is very similar to the one for the smooth case.

Theorem 10. *Assume that the loss function $l(\cdot; z) \in [0, 1]$ is β -smooth, L -Lipschitz and γ -strongly convex for all z . Then SGD with constant step size $\alpha \leq \min(\frac{1}{\beta}, \frac{2}{\beta+\gamma})$ for t steps is ϵ -uniformly stable, where*

$$\epsilon_{stab} \leq \frac{2L^2}{\gamma n}.$$

4.1.3 Non-convex Optimization

Theorem 11. *Assume that the loss function $l(\cdot; z) \in [0, 1]$ is L -Lipschitz and β -smooth for all z . SGD with monotonically non-increasing step sizes $\alpha_t \leq c/t$ for t steps for some constant c is ϵ -uniformly stable, where*

$$\epsilon_{stab} \lesssim \frac{t^{1-1/(\beta c+1)}}{n},$$

where we have omitted constant factors dependent on β , L , and c .

The the proof for the non-convex case exploits the fact that SGD typically makes several steps before it encounters the one example on which two data sets in the stability analysis differ. Specifically, for any m , the probability that SGD first encounters that example in the m -th step is at most m/n . This allows us to argue that SGD has a long “burn-in period” during which $w_t = w'_t$. Thus, by the time $\delta_t = \|w_t - w'_t\|$ begins to grow, the step size has already decayed allowing us to obtain a non-trivial bound.

5 Asynchronous SGD

5.1 Preliminaries

Having established bounds on the generalization error of SGD, we now turn our attention in the analysis of the algorithm when it is executed in an asynchronous, parallel fashion. Conceptually, the computational benefit of applying SGD in a parallel manner is easily understood, but when this happens in an asynchronous environment, i.e., with a lock-free schedule, the analysis of the final results can be difficult. Unlike the sequential case where there is a single copy of the iterate w , in the asynchronous case each of the many cores used, has a separate copy of w in its own cache. Writes from one core may take some time to be propagated to another core’s copy of w , which results in race conditions where stale data is used to compute the gradient updates. In [3] the authors provide a general framework for producing convergence bounds for SGD in the asynchronous setting. Broadly, they analyze iterative algorithms that update the current iterate w by an update step of the form

$$w_{t+1} = w_t - \tilde{G}_t(w_t), \tag{19}$$

for some i.i.d. update function \tilde{G} . Here, to make our exposition simpler and more coherent, we will restrict our attention at the SGD update with batch size $b = 1$, i.e., $\tilde{G}(w_t) = \alpha \nabla_{w_t} l(w_t; z_i)$ as in Section 4.1. Also, we will assume that $\Omega = \mathbb{R}^n$.

In what follows, using the terminology of [3], we will declare an application of SGD *successful* if it produces an iterate in some success region O , e.g., a ball centered at the optimum w^* . Particularly, after running SGD for T timesteps, we say that the algorithm has succeeded if $w_t \in O$ for some $t \leq T$; otherwise, we say that SGD has failed, and we denote this failure event as F_T . We are now ready for the first definition.

Definition 12. *A non-negative process $S_t : \mathbb{R}^{n \times t} \rightarrow \mathbb{R}$ is a rate supermartingale with horizon B if (i) it is a supermartingale, i.e., for any sequence w_t, \dots, w_0 and any $t \leq B$,*

$$\mathbb{E}[S_{t+1}(w_{t+1}, w_t, \dots, w_0)] \leq S_t(w_t, \dots, w_0), \tag{20}$$

and (ii) for all $t \leq B$ and for any sequence w_t, \dots, w_0 that never entered O , it must be that

$$S_t(w_t, \dots, w_0) \geq t. \quad (21)$$

In a martingale-based proof of convergence for the sequential version of SGD, one must construct a supermartingale S_t that is a function of time and the current and past iterates. Informally, S_t will represent how unhappy we are with the current state of the iterate. The first part of the above definition, states that in expectation the process must be non-increasing over time, which when used in the context of SGD convergence, requires that SGD makes some progress towards reaching w^* as time passes. The second part mandates *linear* (in t) lower bound for S_t as long as the algorithm has not succeeded. These two properties of a rate supermartingale provide us with a simple and efficient convergence bound for sequential SGD. The exact bound is given in the following theorem.

Theorem 13. *Assume that we run a sequential version of SGD, for which S is a rate supermartingale. For any $t \leq B$, the probability that the algorithm has not succeeded by time t is*

$$P(F_t) \leq \frac{\mathbb{E}[S_0(w_0)]}{t}. \quad (22)$$

5.2 Modeling Asynchronicity

Before we proceed with asynchronous SGD, we state here the basic assumptions made by [3]. They are the same as those used in previous related work (e.g., [5]).

In summary:

1. There are multiple threads running iterations of (19), each with its own cache. At any point in time these caches may hold different values for the variable w , and they communicate via some cache coherency protocol. Furthermore, to compute the gradient each thread can in principle use a different training example.
2. There exists a central store Q (typically RAM) at which all writes are serialized. This provides a consistent value for the state of the system at any point in real time.
3. If a thread performs a read R of a previously written value W , and then writes another value W' (dependent on R), then the write that produced W will be committed to Q before the write that produced W' .
4. Each write of the iteration of (19) is to only a single entry of w and is done using an atomic read-add-write instruction. That is, there are no write-after-write races.

Notice that, if we let w_t denote the value of the vector w in the central store Q after t writes have occurred, then since the writes are atomic, the value of w_{t+1} is solely dependent on the single thread that produces the write that is serialized next in Q . If we let \tilde{G}_t denote the update function sample that is used by that thread for that write, and \tilde{v}_t denote the cached value of w used by that write, then

$$w_{t+1} = w_t - \tilde{G}_t(\tilde{v}_t).$$

This hardware model further constrains the value of \tilde{v}_t : all the read elements of \tilde{v}_t must have been written to Q at some time before t . Therefore, for some nonnegative variable $\tilde{\tau}_{i,t}$,

$$e_i^T \tilde{v}_t = e_i^T w_{t-\tilde{\tau}_{i,t}}$$

where e_i is the i th standard basis vector. We can think of $\tilde{\tau}_{i,t}$, as the delay in the i th coordinate caused by the parallel updates. Lastly, to model the delays under this model we will assume that the delays are bounded from above by some random variable $\tilde{\tau}$. Specifically, if \mathcal{F}_t , the filtration, denotes all random events that occurred before time step t , then for any i, t , and k

$$P(\tilde{\tau}_{i,t} \geq k | \mathcal{F}_t) \leq P(\tilde{\tau} \geq k).$$

We let $\tau = \mathbb{E}[\tilde{\tau}]$, and call τ the *worst-case expected delay*.

5.3 Convergence Rates

Equipped with a concrete hardware model, we are now ready to present the convergence results for the asynchronous SGD. To do so, we start by extending the previous definition regarding a rate-supermartingale process by incorporating some necessary conditions of continuity and boundness.

Definition 14. *An algorithm with rate supermartingale S is (H, β, L) -bounded if:*

1. S must be Lipschitz continuous in the current iterate with parameter H ; that is, for any t, u, v , and sequence w_t, \dots, w_0 ,

$$\|S_t(u, w_{t-1}, \dots, w_0) - S_t(v, w_{t-1}, \dots, w_0)\| \leq H\|u - v\|. \quad (23)$$

2. The update function must be norm-1 Lipschitz continuous in expectation with parameter β ; for SGD, that implies that for any $u, v \in \Omega$ and any training examples z, z' ,

$$\mathbb{E}[\|\nabla_u \ell(u; z) - \nabla_v \ell(v; z')\|] \leq \frac{\beta}{\alpha} \|u - v\|_1 \quad (24)$$

i.e., it suffices that the loss ℓ to be β/α -smooth.

3. The expected magnitude of the update must be bounded by L . That is, for any $u \in \Omega$ and training example z ,

$$\mathbb{E}[\|\nabla_u \ell(u; z)\|] \leq \frac{L}{\alpha}. \quad (25)$$

i.e., it suffices for ℓ to be L/α -Lipschitz.

It is worth pointing out that in Definition 14 the smoothness of ℓ is measured under the 1-norm. This is the only case in this manuscript where we are not using the 2-norm and, as it becomes apparent in the related asynchronous SGD proofs, this stems from the restriction of a thread being able to update only a single entry of w . It is also worth pointing out the relationship between the boundedness prerequisites and the learning step α . Namely, for a fixed β (or L), the smaller the learning step is, the laxer the bounding restrictions on ℓ become. This nicely ties with the dependence we presented between the learning step and the generalization bounds of SGD as those bounds similarly tend to become tighter with smaller α .

The fundamental convergence result of [3] is the following.

Theorem 15. *Assume that we run asynchronous SGD with the above hardware model, for which S is a (H, β, L) -bounded rate supermartingale with horizon B . Further assume that $H\beta L\tau < 1$. For any $t \leq B$, the probability that the algorithm has not succeeded by time t is*

$$P(F_t) \leq \frac{\mathbb{E}[S(0, w_0)]}{(1 - H\beta L\tau)t}. \quad (26)$$

Notice that the rate of convergence depends only on the worst-case expected delay τ and not on any other properties of the hardware model. Secondly, compared to the result of the sequential case (Theorem 13), the probability of failure has only increased by a factor of $1/(1 - H\beta L\tau)$. In most practical cases, $H\beta L\tau \ll 1$, so this increase in probability is negligible. To give some intuition linking the supermartingale process of an asynchronous SGD and the above theorem, we will present a sketch for its proof.

First, notice that the process S_t , which was a supermartingale in the sequential case, is not in the asynchronous case because of the delayed updates (i.e., the difference between w_t and \tilde{v}_t). To compensate for this, in [3] the authors used S_t to produce a new process \bar{S}_t that is a supermartingale in the asynchronous case. Namely, for any t if $w_u \notin O$ for all $u < t$, they define

$$\bar{S}_t(w_t, \dots, w_0) = S_t(w_t, \dots, w_0) - H\beta L\tau t + H\beta \sum_{k=1}^{\infty} \|w_{t-k+1} - w_{t-k}\| \sum_{m=k}^{\infty} P(\tilde{\tau} \geq m).$$

Compared with S , \bar{S} has two additional terms. The first negative term is canceling out some of the unhappiness that was ascribed to (21) for running for many iterations. Since the asynchrony makes the problem harder, we can interpret this cancellation as allowing SGD to run more iterations in the asynchronous case compared to the sequential version. The second term measures the distance between consecutive iterates and emphasizes the more recent ones; intuitively we would be unhappy if this becomes large because then the noise from the delayed updates would also be large. By exploiting the boundness conditions imposed in the definition of S , it was then straightforward to show that \bar{S}_t is a supermartingale for the asynchronous SGD. Once this was proved, by a similar argument to the one used to prove convergence in the sequential case, namely Theorem 13, they proved Theorem 15.

5.4 Applications

Here we present the rate-martingales and their corresponding convergence bounds for asynchronous SGD for two different applications. The first application concerns the minimization of a strongly-convex loss function via SGD applied under high precision fixed-point arithmetic. The second one uses the same setup but the arithmetic is done with low precision. Applying SGD with low-precision is a popular way for further speeding up its computation and as we will show rate-martingales can be used to address this form of noise as well.

5.4.1 High-precision arithmetic

To begin, assume a loss ℓ that is γ -strongly convex, it is norm-1 smooth with Lipschitz constant β , and has second moment bounded gradient, i.e., $\mathbb{E}[\|\nabla\ell\|^2] \leq M^2$. Furthermore, for some $\epsilon > 0$, let the success region be

$$O = \{w \mid \|w - w^*\| \leq \epsilon\} .$$

Under these assumptions, we define the following process that takes place for as long as asynchronous SGD hasn't succeeded. Concretely, if SGD has not succeeded by time step t , let

$$S_t(w_t, \dots, w_0) = \frac{\epsilon}{2\alpha\gamma\epsilon - \alpha^2 M^2} \log \left(e \|w_t - w^*\|^2 \epsilon^{-1} \right) + t .$$

As the authors of [3] prove, S_t is an $(H, \alpha\beta, \alpha M)$ -bounded supermartingale for SGD with horizon $B = \infty$ and $H = 2\sqrt{\epsilon}(2\alpha\gamma\epsilon - \alpha^2 M^2)^{-1}$. Using this fact and Theorem 15 we can derive an exact bound on the failure event for SGD.

Corollary 16. *Assume that we run asynchronous SGD, where for some constant $\vartheta \in (0, 1)$ we choose the learning step*

$$\alpha = \frac{\gamma\epsilon\vartheta}{M^2 + 2\beta M\tau\sqrt{\epsilon}} .$$

Then for any t , the probability that the algorithm has not succeeded by time t is

$$P(F_t) \leq \frac{M^2 + 2\beta M\tau\sqrt{\epsilon}}{\gamma^2\epsilon\vartheta t} \log \left(e \|w_0 - w^*\|^2 \epsilon^{-1} \right) .$$

5.4.2 Low-precision arithmetic

One of the ways of speeding up the performance of asynchronous SGD is by using low-precision fixed-point arithmetic. This practice introduces additional noise to the system in the form of round-off error. To model this noise, assume that the round-off error can be governed by an unbiased rounding function operating on the gradient updates. That is, for some chosen precision factor κ , there is a random quantization function \tilde{Q} such that, for any $w \in \Omega$, it holds that $\mathbf{E}[\tilde{Q}(w)] = w$, and the round-off error is bounded by $|\tilde{Q}(w) - w| < \alpha\kappa M$. Using this function, we can write a low-precision asynchronous update rule for convex SGD as

$$w_{t+1} = w_t - \tilde{Q}_t(\alpha \nabla_{\tilde{v}_t} \ell(\tilde{v}_t)), \tag{27}$$

where \tilde{Q}_t operates only on a single entry of $\nabla\tilde{\ell}(\tilde{v}_t)$.

Similarly to what was done for the high-precision case, one can exploit the above formulation and the assumptions made to construct a rate supermartingale for the low-precision version of the convex SGD. Once this is done, we can rely again to Theorem 15 to bound the failure rate of this case.

Corollary 17. *Assume that we run asynchronous low-precision convex SGD, and for some $\vartheta \in (0,1)$ we choose step size*

$$\alpha = \frac{c\epsilon\vartheta}{M^2(1 + \kappa^2) + LM\tau(2 + \kappa^2)\sqrt{\epsilon}},$$

then for any t , the probability that the algorithm has not succeeded by time t is

$$P(F_t) \leq \frac{M^2(1 + \kappa^2) + \beta M\tau(2 + \kappa^2)\sqrt{\epsilon}}{\gamma^2\epsilon\vartheta t} \log\left(e\|w_0 - w^*\|^2\epsilon^{-1}\right).$$

In typical systems, the precision is chosen so that that $\kappa \ll 1$; in this case, the increased error compared to the result of Corollary 16 will be negligible and we will converge in a number of steps that is very similar to the high-precision, sequential case. Since each low precision update runs in less time than an equivalent high precision update, this result means that an execution of low-precision SGD will produce same-quality output in less wall-clock time compared to a high-precision SGD. To demonstrate this latter point along with the computational gains of asynchronous vs. sequential SGD, we conclude this paper with some experimental findings.

5.4.3 Experiments with Asynchronous SGD

The first experiment of [3] revealed that asynchronous SGD (ASGD) can train a convex function in a robust way with respect to the arithmetic precision used. Concretely, as we see in Table 4 in four datasets ASGD achieved very similar training error for a variety of arithmetic precisions. The underlying loss used in this experiments was the logistic regression and the step size was fixed to be $\alpha = 0.0001$. According to the authors the results were similar for several choices of step sizes.

Dataset	32-bit float	16-bit int	8-bit int
Reuters	0.5700	0.5700	0.5709
Forest	0.6463	0.6463	0.6447
RCV1	0.1888	0.1888	0.1879
Music	0.8785	0.8785	0.8781

Table 4: Training error of ASGD as a function of arithmetic precision for logistic regression. Each row reports the training error for a particular dataset.

Their second experiment, which was done under the same settings, revealed that by using low-precision arithmetic one can get a speed-up of up to $2.3\times$ compared to using high-precision. It also demonstrated that both versions of ASGD can gain significant speedups compared to their sequential counterpart (see Fig. 4 and [3] for more details).

Performance of **low precision** ASGD for logistic regression

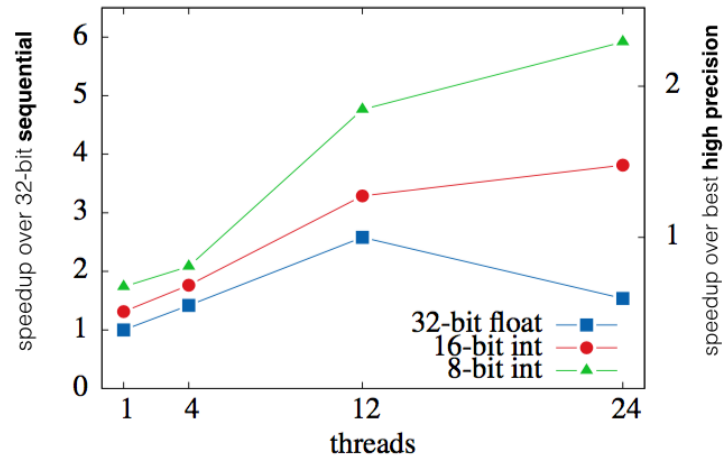


Figure 4: Runtime performance comparison of low ASGD vs. high ASGD vs. SGD on the RCV1 dataset.

A Proof of Theorem 3

Let $S = (z_1, \dots, z_n)$ and $S' = (z'_1, \dots, z'_n)$ be two independent random samples and denote the sample that is identical to S except in the i -th example where we replace z_i with z'_i by $S^{(i)} = (z_1, \dots, z_{i-1}, z'_i, z_{i+1}, \dots, z_n)$. With this notation, equations (28) and (31) are trivial, i.e., hold by definition. Equation (30) is also trivial, by the definition of δ . The heart of the matter, and the clever part of the proof, is equation (29). To see why (29) holds it is important to note that the running index i also changes the set $S^{(i)}$ and to make very strong use of linearity of expectation, i.e., that for any fixed i we have $\mathbb{E}_S \mathbb{E}_{S'} \mathbb{E}_A f(A(S); z_i) = \mathbb{E}_S \mathbb{E}_{S'} \mathbb{E}_A f(A(S^{(i)}); z'_i)$.

$$\mathbb{E}_S \mathbb{E}_A [R_S(A(S))] = \mathbb{E}_S \mathbb{E}_A \left[\frac{1}{n} \sum_{i=1}^n f(A(S); z_i) \right] \quad (28)$$

$$= \mathbb{E}_S \mathbb{E}_{S'} \mathbb{E}_A \left[\frac{1}{n} \sum_{i=1}^n f(A(S^{(i)}); z'_i) \right] \quad (29)$$

$$= \mathbb{E}_S \mathbb{E}_{S'} \mathbb{E}_A \left[\frac{1}{n} \sum_{i=1}^n f(A(S); z'_i) \right] + \delta \quad (30)$$

$$= \mathbb{E}_S \mathbb{E}_A [R(A(S))] + \delta, \quad (31)$$

where

$$\delta = \mathbb{E}_S \mathbb{E}_{S'} \mathbb{E}_A \left[\frac{1}{n} \sum_{i=1}^n f(A(S^{(i)}); z'_i) - \frac{1}{n} \sum_{i=1}^n f(A(S); z'_i) \right].$$

Taking the supremum over any two data sets S, S' differing in only one sample we see that our assumption on the uniform stability of A implies

$$|\delta| \leq \sup_{S, S', z} \mathbb{E}_A [f(A(S); z) - f(A(S'); z)] \leq \epsilon .$$

B Proof of Theorem 9

Let S and S' be two samples differing in only a single example. Consider the gradient updates G_1, \dots, G_T and G'_1, \dots, G'_T induced by running SGD on S and S' , respectively. Let w_T and w'_T denote the corresponding outputs. For any example $z \in \mathcal{Z}$, the Lipschitzness of $f(\cdot; z)$ implies

$$\mathbb{E}|f(w_T; z) - f(w'_T; z)| \leq L \mathbb{E}[\delta_T] , \quad (32)$$

where $\delta_T = \|w_T - w'_T\|$.

Observe that at step t , with probability $1 - 1/n$, the example selected by SGD is the same in both S and S' . In this case $G_t = G'_t$ and we can use the 1-expansivity of the update rule G_t which follows from Lemma 7 using that the objective function is convex and that $\alpha_t \leq 2/\beta$. With probability $1/n$ the selected example is different in which case we use that both G_t and G'_t are α_t -bounded as a consequence of Lemma 6. Hence, we can apply the growth recursion lemma and linearity of expectation to conclude that for every t ,

$$\mathbb{E}[\delta_{t+1}] \leq \left(1 - \frac{1}{n}\right) \mathbb{E}[\delta_t] + \frac{1}{n} \mathbb{E}[\delta_t] + \frac{2\alpha_t L}{n} = \mathbb{E}[\delta_t] + \frac{2L\alpha_t}{n} .$$

Unraveling the recursion gives

$$\mathbb{E}[\delta_T] \leq \frac{2L}{n} \sum_{i=1}^T \alpha_i .$$

Plugging this back into equation (32) we obtain

$$\mathbb{E}|f(w_T; z) - f(w'_T; z)| \leq \frac{2L^2}{n} \sum_{t=1}^T \alpha_t .$$

Since this bounds holds for all S, S' and z , we obtain the desired bound on the uniform stability.

References

- [1] Ilya Sutskever, James Martens, George Dahl, Geoffrey Hinton *On the importance of initialization and momentum in deep learning*. In ICML, 2013.
- [2] Moritz Hardt, Benjamin Recht, Yoram Singer *Train faster, generalize better: Stability of stochastic gradient descent*. In JMLR, 2015.
- [3] Christopher De Sa, Ce Zhang, Kunle Olukotun, Christopher Re *Taming the wild: a unified analysis of HOGWILD!-style algorithms*. In NIPS, 2015.
- [4] Bousquet Olivier and Elisseeff Andre. *Stability and generalization*. In JMLR, 2002.
- [5] Feng Niu, Benjamin Recht, Christopher Re, Stephen Wright *Hogwild: A lock-free approach to parallelizing stochastic gradient descent*. In NIPS, 2011.
- [6] Yurii Nesterov. *Introductory lectures on convex optimization. A basic course*. Volume 87 of Applied Optimization. Kluwer Academic Publishers, Boston, MA, 2004.
- [7] Geoffrey Hinton, and Ruslan Salakhutdinov *Reducing the dimensionality of data with neural networks*. In Science, 313:504507, 2006.
- [8] James Martens *Deep learning via Hessian-free optimization*. In ICML, 2010.
- [9] Olivier Chapelle and D. Erhan *Improved Preconditioner for Hessian Free Optimization*. In NIPS, 2011.
- [10] Yoshua Bengio, Simard P., and Frasconi P. *Learning long-term dependencies with gradient descent is difficult*. In IEEE Transactions on Neural Networks, 1994.
- [11] Trevor Hastie, Jerome H. Friedman, Robert Tibshirani *The Elements of Statistical Learning*. Springer series in Statistics.