UNIVERSITÀ COMMERCIALE LUIGI BOCCONI

Department of Computing Sciences

Bachelor degree in Mathematical and Computing Sciences

for Artificial Intelligence

Stochastic Gradient Descent with Barker Dynamics: an empirical study on Neural Networks training

Supervisor: Professor Giacomo Zanella

Dario Filatrlla

Matr. 3144366

Anno Accademico 2022 / 2023

Abstract

Training Neural Networks is a complex task, due to the high non-convexity of the loss landscape that creates many local minima in which algorithms get stuck. Several approaches (first-order methods, Monte Carlo methods,...) have been tried, yielding a wide range of results, but without providing a general-purpose solution. This thesis presents an analysis of the training process with Stochastic Gradient Barker Dynamics (SGDB). SGDB is a Monte Carlo Markov Chain algorithm that simulates a jump kernel moving in the parameter space. Two main modifications have been made in order to make the algorithm suitable for the high dimensionality of the problem: each layer has an adaptive temperature that regulates the likelihood to jump to a new state and each parameter has an adaptive stepsize that depends on its partial derivative. Through experiments on Convolutional networks, it is shown that SGDB outperforms Adam on well-regularized and small networks. The stochastic nature of this algorithm makes it able to visit the space quite well, an issue that is present in more deterministic optimization algorithms. Two variants of SGDB have been analyzed, namely the corrected and the extreme version, yielding very similar results to the standard version. The algorithm limitations suggest that significant improvements should include a better acceptance proposal correction and an adaptive global stepsize, both specifically suited for the dimensionality of the task.

Contents

1 Introduction

The development of Artificial Intelligence (AI) can be traced back to the early days of computing in the 20th century when researchers began exploring the idea of automating complex human tasks. The first prominent contributions were due to Alan Turing, who in the 1950s proposed a test, now known as the Turing test, which served the purpose of determining whether a true Artificial Intelligence system had been developed. The test involves a human judge interacting with a machine and another human. His goal is to understand which one is the computer, and if he thinks they are indistinguishable then the machine has passed the Turing test. This test is now outdated as multiple Large Language Models (LLM) have passed it but they are not regarded as "intelligent".

In 1956, the *Darmouth Summer Research Project on Artificial Intelligence*, hosted by John McCarthy and Marvin Minsky, is considered the first significant AI project. The conference was not as successful as the hosts hoped but it still set a milestone in the field as the first attempt to gather efforts and set a standard. In the same period Allen Newell, Cliff Shaw, and Herbert Simon wrote *Logic Theorist*, which was a program designed to imitate the solving skills of a human and it successfully proved 38 theorems from Russell *Principia Mathematica*. This was one of the first breakthroughs and it ventured extensively into the area of search trees.

From 1957 to 1974, the field of AI flourished, driven by exponential growth in computing power (Moore's law was stated in 1965), increased funding, and government interest. In particular in the US where the DARPA (Defense Advanced Research Project Agency) funded many AI research institutions and the government was particularly interested in translating spoken languages automatically. The expectations were, in hindsight, too high, and, for example, in 1970 mathematician and AI researcher Marvin Minsky stated that *"from three to eight years we will have a machine with the general intelligence of an average human being"*. During this period, two primary approaches were explored in AI research: symbolic AI and connectionist AI. Symbolic AI involved using symbolic logic and rules to represent knowledge and perform reasoning, while connectionist AI drew inspiration from neural networks and the brain's structure and function.

The connectionist approach gave rise to the Perceptron, the first neural network, invented by W. McCulloch and W. Pitts who published it in 1943 but implemented it only in 1958. The Perceptron, a single-layer neural network, captured attention due to its resemblance to the brain and its learning algorithm (analytically it is equivalent to a support vector machine). The way the perceptron is trained is to iterate over all the data points one wants to learn and correct the weights only when a mistake is done. This learning rule seems quite similar to a natural learning process and this similarity created a lot of expectations around it. In 1958 at a conference organized by the Us Navy the New York Times reported that the perceptron was the embryo of a computer capable of walking, talking, and reproducing itself. Despite the high promises, in 1969, a well-known book entitled *Perceptrons* by Marvin Minsky and Seymour Papert showed that this class of functions could not learn a XOR pattern (figure 1). The problem could be solved by using Multi-Layer Percpetrons (which are Neural Networks with only dense layers) but their impracticability led to a slowdown in connectionist AI research.

In the 1970s AI had a setback. The critiques of the perceptron shut down the connectionism field and a series of problems arose:

- Limited computing power
- Exponential complexity of certain problems.
- Lack of Commonsense knowledge in computers. Many important problems required to know information about the world that seem obvious but still need to be taught to the algorithm
- Moravec's paradox. Problems that are easy for humans, such as learning faces, are exceptionally hard for computers, in particular, the field of Natural Language Processing and Computer Vision underestimated this

In the 1980s AI experienced a resurgence as new approaches and technologies emerged. In the industry "Expert systems" were adopted by many corporations and drove a lot of funding toward AI, while the work of John Hopfield and David Rumelhart revitalized the connectionist approach. Hopfield focused on theoretical advancements in Hopfield nets, while Rumelhart popularized a method called *backpropagation* for training neural networks. Those ideas laid down the foundations for current deep neural network training, nowadays gradients are still computed via backpropagation.

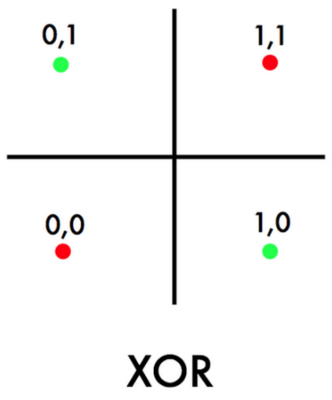

Figure 1: The class of each point is given by the result of $XOR(x, y)$. It is possible to observe that no plane can separate those datapoints and therefore the Perceptron cannot learn them

Another *AI winter* quickly followed and lasted from 1987 to 1993, the main cause was the failure to commercialize and many projects were too ambitious to deliver. For example, Japan's *Fifth Generation Project* goals had not been met and were overly optimistic, such as *"Carry on a casual conversation"*.

The mid-1990s marked yet another boom in the field. The factor was a combination of increased computing power, better models (especially neural networks), better training algorithms, larger datasets available, and commercial interest. Several notable events occurred during this period, such as Deep Blue defeating Kasparov in 1997 (although it was not deep learning but GOFAI, Good Old Fashioned AI) and AlexNet winning an image recognition competition in 2012, marking the first victory for a convolutional neural network. Additionally, attention layers and transformer architectures (such as GPT, BERT, and LamDa) set new standards for machine translation and demonstrated capabilities that passed some forms of Turing tests.

It is worth defining properly what Neural Networks are, given their importance in this thesis. The simplest network, the multi-layer Percpetron is a function $f_{w,b}(x)$ with a fixed number of layers N and an activation function σ in the form:

$$f_{w,b}(x) = W_N x_N \tag{1.1}$$

$$x_{l+1} = \sigma(W_l x_l + b_l) \tag{1.2}$$

$$x_0 = x \tag{1.3}$$

Where W_l and b_l are the weights and biases of the layer l. Equation 1.2 is a recursive definition and it holds for all $l = 0...N$. Often σ is not the same at each layer, and often the last layer has its own activation function but diving into the different types of models would be too complex.

Over the past two decades, neural networks have become widely utilized to address a wide range of problems, revolutionizing various fields such as computer vision with Convolutional Nets (CNN), Natural Language Processing with transformer architectures, image generation with Adversarial and Transformer networks, and many others. Those networks are characterized by high dimensional (overparametrized) parameter spaces and very expressive capabilities which, together, fundamentally differentiate them from classical statistical models. In particular, they can approximate every regular function, given enough parameters. There are some formal results called Universal approximation theorems that, roughly speaking, state that any function $f : \mathbb{R}^N \mapsto \{-1, 1\}$ can be approximated at an arbitrarily small precision ϵ with a Neural network with at least one hidden layer and a non-polynomial activation function. The issue is that there is no easy way to quantify the size of the hidden layer, in the simplest proof it is taken to be exponential, however, this statement does not tell us anything about how to find the right parameter configuration for a Neural Network [HH89].

The high expressibility capacity also makes NNs analytically not tractable, therefore, there is no theoretical way to find the best parameter configuration (i.e., the one that minimizes the empirical risk). In contrast, simpler models, such as Logistic and Linear regressions, exhibit closed-form solutions or algorithms that are guaranteed to converge to a solution. Thus, training NNs becomes a crucial part of the framework. The training can be formulated as an optimization problem similar to the Bayesian statistical setting; thus, many ideas and concepts are common to the two fields and can be transferred from one setting to the other.

The non-trivial nature of the training makes it an interesting subject to study: the same network can achieve a wide variety of performances if trained with different algorithms. This motivates the need to study and test multiple approaches.

The algorithm that will be analyzed in this thesis and adapted to train Neural Networks is called Stochastic Gradient Barker Dynamics (SGDB). It consists of a Markov Chain with a specific transition kernel and comes from the statistical field and the formal derivation of this kernel, together with its properties will be introduced rigorously in the Algorithm section.

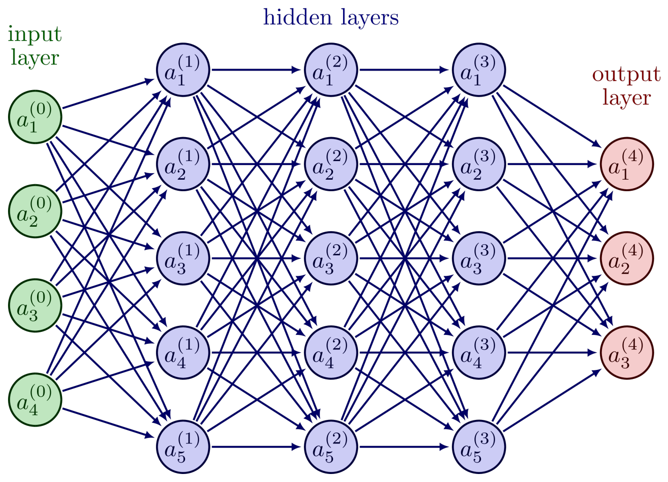

Figure 2: Example of a dense neural network with 5 layers

1.1 The connection between Statistical optimization and Bayesian inference

In the context of Bayesian statistics one usually wants to compute the distribution of the posterior of the parameter θ. Assuming an i.i.d. sample that does not depend on θ, the distribution is implicitly given by

$$f(\theta|X,Y) = \frac{\prod_i f(y_i|x_i,\theta)}{z} p(\theta) \tag{1.4}$$

where $X = \{x_1, x_2, ...\}$ is the data, $p(\theta)$ is the prior and z a normalization constant that only depends on X. With the exception of some simple models (Linear regressions, ANOVA, ...) computing z is often not analytically possible and numerically expensive. However, there are two main workarounds. The first is to only find the most likely value (MLE) of θ, which can even be shown to be an unbiased estimator, [BJV17] and reduces to solving an optimization problem in θ in which z does not need to be computed. The second approach is to try to sample from the unnormalized distribution $p(\theta|X) = f(X|\theta)p(\theta)$ which gives a set of parameters $\theta_1, \theta_2, ...$ that can later be used for predictions. Luckily for a number of models, the posterior distribution $f(\theta|X,Y)$ is convex so often it is possible to approximate the solution with arbitrary precision and guarantee that it is indeed the optimal value. There is a lot of software that implements (STATA, STAN, and similar all exploit MCMC and gradient-based methods) such an approach.

1.2 Training Neural Networks

The usual framework to train a Neural Network is to minimize the loss:

$$L(\theta) = \frac{1}{N} \sum_i \mathcal{L}(f(x_i, \theta), \hat{y}_i) \tag{1.5}$$

where $\mathcal{L}$ is the error function (for example cross-entropy or log loss in the case of classification), N is the size of the dataset, x_i are the datapoints and $\hat{y}$ is the true value.

Training a NN and computing the Bayesian posterior of a model are indeed two similar problems. The differences are the absence of a prior, which is often added in the form of different equivalent regularizations, and the loss not being a probability, even though it can be interpreted in a probabilistic sense as a log-likelihood in many cases (for example the weight decay used by Adam is equivalent to a Gaussian prior). Those problems are usually tackled by optimizing using gradient based methods, as will be discussed in the following sections.

1.3 What is Gradient Descent

Gradient descent is the fundamental building block of almost all gradient based optimization algorithms, the idea is to interpret the gradient as the direction of maximum increase/decrease and to move along such path. The simplest setting in which gradient descent (GD) can be used is to minimize a given function $f : \mathbb{R}^N \mapsto \mathbb{R}$, which is differentiable, with gradient $\nabla f : \mathbb{R}^N \mapsto \mathbb{R}^N$ on the domain $\mathbb{R}^N$.

Algorithm 1 Vanilla Gradient Descent (GD)

```
Input: $x_0$, $\nabla f$, $T$, $\gamma$
  for $t = 1, ..., T$ do
      $x_t = x_{t-1} - \gamma * \nabla f(x_t)$
  end for
```

If f is convex it is possible to show that the algorithm will find a global minimum provided that the stepsize γ is small enough. If one assumes strong convexity or L-smoothness it is possible to show strong upper bounds on the time of convergence. Those results motivate the possibility of applying this algorithm also to non-convex problems [BV04]. GD is very flexible and, with few modifications, it can be extended to cases of constrained optimization (optimization on manifolds, linear programming, ...), online optimization (OCO), stochastic optimization, etc.

1.4 Gradient Descent as a Markov Chain Monte Carlo

Each iteration of GD only depends on the previous x_t therefore it can be interpreted as a (deterministic) Markov chain. The general idea is that one can add some noise to the algorithm to design a MC with a specific stationary distribution, then let the MC converge and sample from the asymptotic behavior. This has been done widely in statistical physics to study the macroscopic properties of a system, one of the most famous algorithms (and foundation block of many methods) being the Metropolis-Hasting. In the specific case of SGDB, the target distribution will not be exact as there is a first-order approximation in the kernel that produces a tampered distribution which will be covered in the section about the formal derivation.

1.5 Neural Networks training with gradient descent

The main algorithms that have been used for Neural Networks training are variations of gradient descent. In this thesis, the performance of the Stochastic Gradient Barker Dynamics Algorithm will be analyzed and compared to ADAM which is the current state of the art.

It is possible to naively train NN using gradient descent, but some (insightful) problems arise:

- Computing the gradient can be expensive as it scales as $O(N)$
- The loss $\mathcal{L}$ is highly non-convex for any non-trivial Neural Network (so at least 1 hidden layer and a non-linear activation function) so GD can get stuck in local

minima

- The scale of the parameters is non-homogeneous so having the same stepsize for each coordinate can cause problems
- The parameter space is high-dimensional so the loss landscape is very sharp and difficult to explore efficiently

To tackle these problems, the most used algorithms are ADAM [KB14] and similar variations which add noise and momentum (such as SGD combined with Nesterov momentum) [Sut+13].

1.6 Previous results for Neural Network training

In the literature, there are many analyses of optimization algorithms from the perspective of Markov Chains and Bayesian inference. For example in [M+17] some theoretical results on how to use SGD to compute Bayesian posteriors are shown. A Bayesian approach for Neural Networks has been applied widely in research but not in industrial solutions due to a variety of factors. In [Wen+20] the authors explain how the exact posterior of a Bayesian NN is not the best predictor and how sharpening the posterior helps the accuracy. In the paper, the authors talk about reducing the "temperature" to sharpen the distribution and a similar concept will be applied also to SGDB but in this context, the temperature needs to be increased to achieve the same result (but it is merely a matter of naming). In [Izm+21] it is shown that MCMC can train Neural Networks better than classical first-order methods, but it requires some attention on the prior, good hyperparameters for the Markov chain and many iterations. Therefore several problems arise in those analyses. The first is that MCMC are properly studied only for relatively low-dimensional tasks, while NNs can reach billions of parameters, and this includes SGDB which will have to be properly modified to be suited for the task. The other issue is in the non-practicality of computing Bayes posterior with HMC or other MCMC. These algorithms are very slow and, from an operative point of view, it would be faster to get a bigger network and train that with Adam or SGD. A reasonable approach is to combine both ideas and perform an optimization task, in the same fashion as SGD but using a MCMC algorithm that will explore the parameter space more extensively. As shown later in this way the running time is not really affected while the quality of the prediction can increase in many cases.

2 Algorithm

The algorithm taken into analysis is the Stochastic Gradient Barker Dynamics. The idea is to perform a gradient-based MCMC that converges to an approximation of the target distribution using Barker proposal.

Algorithm 2 Barker Proposal in one dimension

```
Input: $\theta_0$, $\sigma$, $\nabla f$
  for $t = 1, ..., T$ do
      Draw $z \sim N(0, \sigma)$
      Define $p = \frac{1}{1 + \exp(-z \nabla f(\theta_{t-1}))}$
      Set $b = 1$ with probability $p$ and $b = 0$ otherwise
      $\theta_t = \theta_{t-1} + z * b$
  end for
```

2.1 Formal derivation

The idea behind MCMC methods is to build a Markov Chain whose invariant distribution coincides with the target distribution one wants to achieve. This is done by finding functions that satisfy the well-known *detailed balance* equations:

$$\frac{\pi(y)q(y,x)}{\pi(x)q(x,y)} = t(x,y) = 1 \tag{2.1}$$

Here $Q(x,A) := \int_A q(x,y)dy$ is the transition kernel of the MC. Usually one has a kernel q that does not satisfy the detailed balance conditions and wants to build a chain on top of it. A simple but analytically intractable way is to define:

$$p(x,y) := g(t(x,y))q(x,y) \tag{2.2}$$

$$g(t) = tg(1/t) \tag{2.3}$$

It is possible to show that this new p satisfies detailed balance, but it is not guaranteed to be a normalized probability distribution, indeed a serious issue. One possible solution is the well-known Metropolis-Hastings acceptance rule defined as follows:

$$g(t) = \min(1, t) \tag{2.4}$$

$$g_h(t(x,y)) = \min\left(1, \frac{\pi(y)q(y,x)}{\pi(x)q(x,y)}\right) \tag{2.5}$$

Another approach is the one used for the Barker proposal: it is possible to build a jump process for which transitions between two points (x, y) happen at a rate $p(x, y)$. This means that if the chain is at a point x it will stay there for a time $t \sim Exp(\lambda(x))$ where

$$\lambda(x) = \int p(x,y)dy \tag{2.6}$$

And then move to another point via the Jump kernel J defined as:

$$J(x, A) := \int_A \frac{p(x,y)}{\lambda(x)}dy \tag{2.7}$$

One can note that this jump kernel J has invariant distribution proportional to $\lambda(x)\pi(x)$ and if λ is constant then J is π-reversible. By approximating $t(x, y)$ it is possible to find a good constant jump rate process for a generic λ.

The way [MZ23] proceed is to first restrict the analysis to the family of transition densities for which $q(x, y) = q(x - y)$ and $q(x, y) = q(y, x)$ hold. Then $t(x, y) = \frac{\pi(y)}{\pi(x)}$ and the following Taylor expansion holds:

$$t(x,y) = \frac{\pi(y)}{\pi(x)} = \exp(\log \pi(y) - \log \pi(x)) \approx \exp((y-x)\nabla\pi(x)) \tag{2.8}$$

Setting $z := y - x$ one defines:

$$t^*_x(z) := e^{z\nabla\pi(x)} = \frac{1}{t^*_x(-z)} \tag{2.9}$$

Using $q(x, y) = q(z)$ it holds that:

$$\lambda^*(x) := \int_{-\infty}^{\infty} g(t^*_x(z))q(z)dz \tag{2.10}$$

First one notices that by 2.3 and $q(z) = q(-z)$ the following is true:

$$g(t_x^*(-z)) = g(1/t_x^*(z)) = g(t_x^*(z))/t_x^*(z) \tag{2.11}$$

By rearranging the terms in the integral one gets (defining $t^* = t_x^*(z)$):

$$\lambda^*(x) = \int_0^\infty \left[1 + \frac{1}{t^*} g(t^*)\right] q(z) dz \tag{2.12}$$

The authors of [MZ23] then suggest that if the expression in the square brackets was constant then $\lambda^*(x)$ would become tractable which leads to the condition:

$$g(t^*) = \frac{c}{1 + 1/t^*} \tag{2.13}$$

Setting $c = 1$ gives the choice $g_B(t^*) = t^*/(1 + t^*)$.

The last step of the derivation puts everything together and formalizes the structure of the Barker Proposal. By using 2.7 with symmetric q, approximated t^* and balancing function g_B gives the kernel:

$$J^*(x, A) := \int_A 2g_b(\exp[(y - x)\nabla \log \pi(x)])q(y - x)dy \tag{2.14}$$

Using $F_L(z) := 1/(1 + e^{-z})$ the cumulative distribution of the logistic distribution, and noting that $g_B(e^z) = F_L(z)$ one gets the transition density:

$$j^*(x, x + z) = 2F_L(\nabla \log \pi(x)z)q(z) \tag{2.15}$$

This density belongs to the family of *skew-symmetric* distributions and therefore one can sample from this distribution using the Barker proposal algorithm.

2.2 Extension to multiple dimensions

The Barker Proposal is designed for the one-dimensional case but it is possible to extend it. A choice is using $p = (1 + \exp -z^T \dot{\nabla} f(\theta_{t-1}))^{-1}$ and allow only movements in $\{-z, z\}$ directions. Alternatively one can compute a different p_i for each component and thus allow 2^D possible movements. The literature demonstrates that the latter better explores the space and is the preferred choice.

If one naively applies the Barker proposal in a high-dimension regime a few problems arise:

- The algorithm becomes very sensitive to the global stepsize σ
- The probabilities p_i are very likely around $1/2$ (σ too small) or around $0, 1$ (σ too large)

Given those premises, a few changes have been made to the algorithm to make the tuning of the hyperparameters automatic and more robust with adaptive stepsizes and a "temperature" inspired by statistical physics algorithms. One thing to note is that most variables are treated separately for each layer of the network to help the algorithm deal with different types of layers that can operate on very different scales of parameters. This makes the notation more cumbersome as it adds a subscript l that depends on the layer to every variable.

Algorithm 3 Barker Proposal for NN

```
Input: $\gamma_0$, $\gamma_r$, $\nabla f$, $\beta$, $\theta_0$
Initialize: $\mu_0 = 0_v$, $\sigma = 0_v$
 1: for $t = 1, ..., K$ do
 2:     for $l$ in layers do
 3:         $\mu_l = (1-\beta)\mu_l + \beta\nabla f_l(\theta_{t-1})$
 4:         $\sigma_l = (1-\beta)\sigma_l + \beta(\nabla f_l(\theta_{t-1}) - \mu)^2$
 5:         Draw $z \sim N(\mu_l, \mu_l/10)$ from a multinormal distribution
 6:         Define $p = \frac{1}{1+\exp(-T_{l,t} z * \nabla f_l(\theta_{t-1}))}$ with $*$ element-wise product
 7:         Set $b_i = 1$ with probability $p_i$ and $b_i = 0$ otherwise
 8:         $\theta_t = \theta_{t-1} + z * b$
 9:         Update the temperatures
10:         $\alpha = \frac{1}{L}\sum_i^L |p_i - 1/2|$
11:         $\log(T_{l,t+1}) = \log(T_{l,t}) - \gamma_t(\alpha - 1/4)$
12:         $\gamma_t = \gamma_0 / t^{\gamma_r}$
13:     end for
14: end for
```

2.3 Adaptive step size with online exponential mean

In order to scale the stepsize for each coordinate both the exponential mean and the variance are computed and used to scale z. The key point is to have z in the right order of magnitude to explore the landscape. A stepsize too large would exacerbate the MCMC oscillations and, when actually running the code, it may also make gradients explode to infinity and result in numerical errors of the "Nan" type. It has been suggested ([Zha+19]) to clip the gradient for several reasons (which mainly have to do with the smoothing introduced by this clipping) and in this case the clipping also helps deal with this issue which is not present in "less stochastic" optimization algorithms. This is not the only source of gradient explosions and more will be covered later.

2.4 Stochastic gradient

Given the size of datasets, it is often expensive to compute the loss over the whole dataset $L(\theta) = \sum_i^N \mathcal{L}(\hat{y}_i, f(x_i, \theta))$. Usually, it is more convenient to extract a smaller sample S of size K from the dataset and to compute only $\sum_i^K \mathcal{L}(\hat{y}_{S(i)}, f(x_{S(i)}, \theta)$. The difference in the two gradients vanishes in $O(1/\sqrt{K})$, for the central limit theorem, if N is large enough. Therefore, in practical application, it is standard to use this (good) approximation implicitly. In PyTorch it is possible to abstract from this when writing the code for the optimization step and control K when defining the model and the training loop. It is relevant to note that using those batches can make the training faster and more robust: more updates to the weights will be made in the same amount of computing time and some "good" noise is added which helps generalization. The way the batch size influences the training is obviously far more complicated as the noise and the frequency of the updates work in opposite directions. There is a significant amount of theory and applications behind tuning the batch size [DC18], but for the scope of the algorithm it has been considered superfluous or even dangerous to set it as a variable hyper-parameter (dangerous from a multiple hypothesis testing perspective).

2.5 Temperature

As mentioned above, a temperature T_l has been added in how the acceptance probability is computed (line 6 of algorithm 3):

$$p = \frac{1}{1 + \exp\left(-T_{l,t} z * \nabla f_l(\theta_{t-1})\right)} \tag{2.16}$$

In this way, T acts as a stepsize and helps the MC to have reasonable accepting probabilities. For instance, the desired behavior is neither a measure concentrated around 50% nor close to 100%. The temperature is adaptive (lines 10-12) and works as follows:

$$\alpha = \frac{1}{L} \sum_{i}^{L} |p_i - 1/2| \tag{2.17}$$

is the average distance from 1/2 and one wants this not to be close to 0 (random steps) or 1/2 every proposal is accepted. Here it is proposed to use $\alpha^* = 1/4$ as standard, which makes the distribution flat but it does not always work. For instance, for larger models, it can lead to unstable behavior as shown in figure 3. A good heuristic is to lower α when the number of parameters of the model increases.

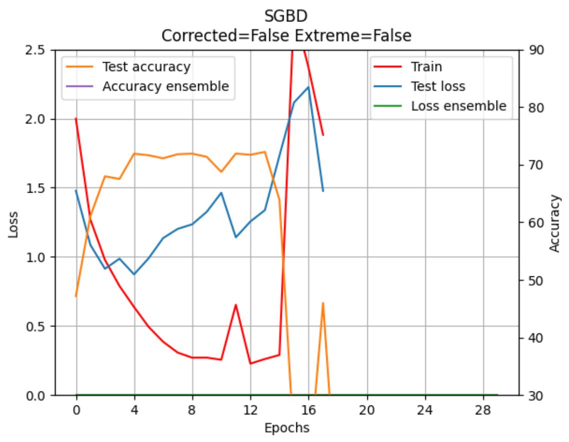

Figure 3: Training of a large model (12M parameters, 10 layers) with a parameter $\alpha* = 1/4$ which leads to chaotic behavior and a gradient explosion

To correct T, a step similar to algorithm 4 of Andrieu and Thoms [AT08] is employed:

$$\log(T_t) = \log(T_{t-1}) - \gamma_t(\alpha(p(T_{t-1})) - \alpha^*) \tag{2.18}$$

$$\gamma_t = \gamma_0 t^{-\gamma_r} \tag{2.19}$$

The stepsize is decreasing, as keeping it fixed would result in non-Markovian behavior. In this way, after a certain time, the temperature oscillations become negligible. The decrease rate γ_r of the stepsize and of the initial value γ_0 only control how fast the temperature can change at the beginning. For every sensible choice of parameters the convergence is quite quick towards the same equilibrium values shown in figure 4, so they are not important parameters to tune nor are they difficult to pick up.

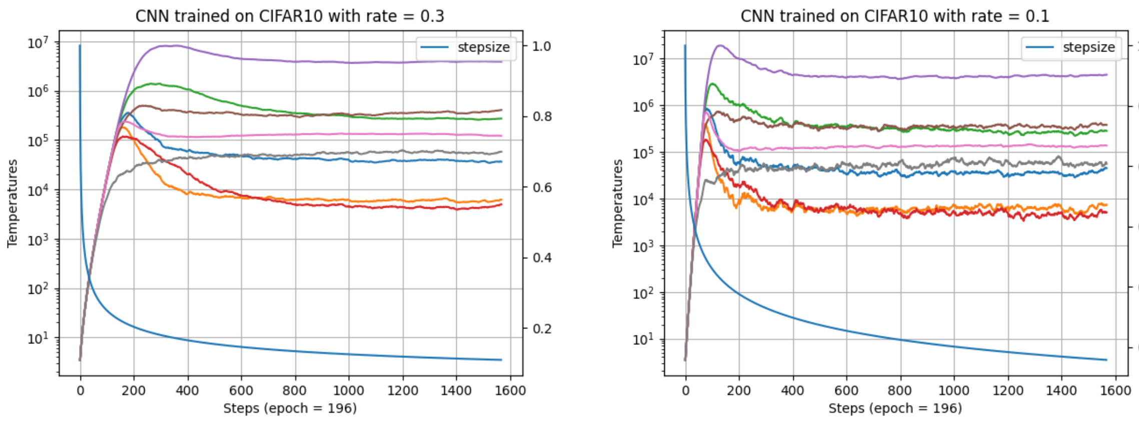

Figure 4: Temperature evolution in time: The network has 204186 parameters and 6 hidden layers (see CnnMedium in the code)

2.6 Curse of dimensionality

Very often in MCMC methods, the temperature is included in the form of stepsize, namely $z \sim N(T\mu, T\mu/10)$. This choice makes sense from the physical perspective where a system at high temperature will tend to move more. This solution sadly seems to break down due to a scaling problem caused by the high-dimensionality. First of all, it is possible to get the following derivation for the order of magnitude of T. Independently of whether the temperature is part of z or it is multiplied only when computing p, one gets:

$$p_i = \frac{1}{1 + \exp(T(\partial_i f)^2)} \tag{2.20}$$

$$p_i \sim 1/4 \tag{2.21}$$

$$\exp(T(\partial_i f)^2) \sim 3 \tag{2.22}$$

$$T(\partial_i f)^2 \sim 1 \tag{2.23}$$

Empirically one can notice that in general $\|\nabla f\| \sim 1$ for neural networks, which implies that $\mathbb{E}[(\partial_i f)^2] \sim 1/N$, where N is the number of parameters of the network. In turn, this implies that $T \sim N$. Considering that N is in the order of millions generally (even a simple logistic regression on CIFAR10 has 30'000 parameters) incorporating this temperature in the stepsize would not change the acceptance probabilities but it would just make the network diverge after very few steps. In the section about fine-tuning, a different way of setting a stepsize will by tested, confirming that an online estimate of the gradient is a sensible choice that is both robust and capable of adapting to the size of the network.

2.7 Ensemble

Given the MCMC nature of present the algorithm, once it stabilizes on the stationary distribution it is arbitrary to pick the last parameters configuration θ_T. A possible solution is to sample from all the visited points. The way it was carried out is that at every step t the model is saved with a fixed probability p and the last M models are saved. This stochasticity is applied to implement an ergodic behavior: to make predictions, the input is fed to all the M saved models and the average of the outputs is used.

Model	M	p	ESS bulk (Min, Med)	ESS tail (Min, Med)
LogisticReg	50	5.0%	1.5, 3.8	12.1, 18.4
LogisticReg	50	100%	1.5, 5.0	12.1, 18.4
LogisticReg	50	1.0%	1.5, 8.4	12.1, 18.4
LargeModel	20	100%	1.8, 3.7	1.7, 20.4
LargeModel	20	5.0%	2.0, 7.7	2.7, 25.6
CnnMedium	50	100%	1.5, 4.8	12.1, 18.4
CnnMedium	50	5.0%	1.5, 5.8	12.1, 18.4
CnnMedium	50	2.5%	1.5, 4.7	9.0, 18.4

Table 1: Minimum and Median Effective Sample size for different models and probabilities combinations

As one can see from 2.7, the ESS does not depend on the frequency at which models were picked. This is a known issue: in high-dimensional regimes, the mixing of a MCMC is very slow and it is quite inevitable to experience this phenomenon. In other words, all the points visited by the MC are highly correlated independently of the frequency.

2.8 Extreme version

As suggested in [MZ23] a simple and natural variation of the algorithm is to remove the stochasticity in choosing b and just set $b = 1$ if $p < 1/2$ and $b = -1$ if $p > 1/2$. In this way, the algorithm will simply follow the gradient using a random stepsize. This seems to work well only in low dimensional regimes as shown in figure 5. On the top, there are 2 quite large models for which the training failed. After epochs 4 and 5 respectively the parameters of the network became Nan. The loss also was already showing unstable behavior. On the bottom, instead, the training of a logistic regression with the extreme version of SGBD and without is shown. In both cases, the training is quite similar.

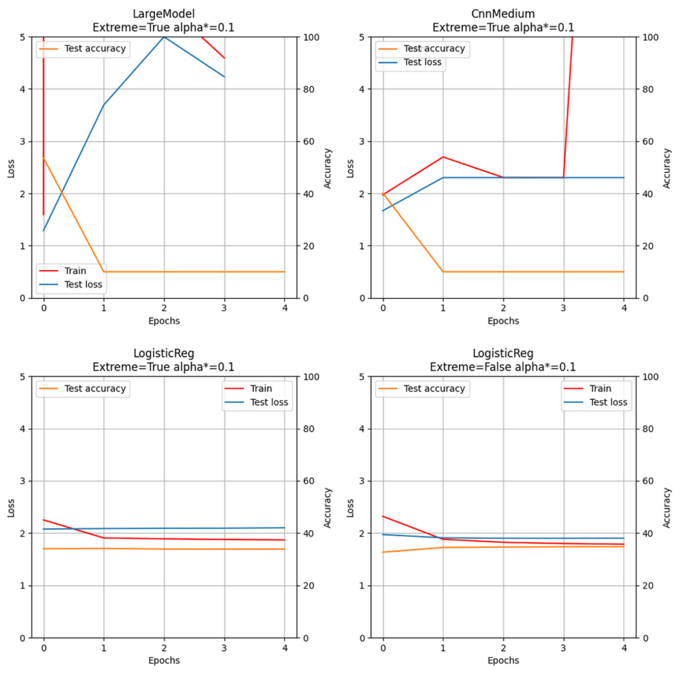

Figure 5: Comparison of training of 3 models with the extreme version

2.9 Corrected version

As one can notice the formal derivation assumes access to the full gradient. As already discussed in practical applications a noisy observation of the gradient is employed. Under the following regularity condition:

$$\hat{\nabla} g(\theta) \sim \nabla g(\theta) + \eta(\theta)\ N(0, \tau(\theta)) \tag{2.24}$$

Where $\hat{\nabla} g$ is the stochastic gradient and $\tau(\theta)$ is the standard deviation of the stochastic noise at value θ. It is worth noting that those conditions are very commonly assumed in the literature and are not very strong. Under condition 2.24 it is possible to recover a bound on the bias of p (acceptance probability)

$$\left|\mathbb{E}[p(\hat{\nabla}g(\theta), z)] - p\left(\frac{1.702}{\sqrt{1.702^2 + z^2\tau^2(\theta)}}\nabla g(\theta), z\right)\right| < 0.019 \tag{2.25}$$

The proof can be found in [MZ23]. This motivates the definition of a new estimator $\tilde{p}$,

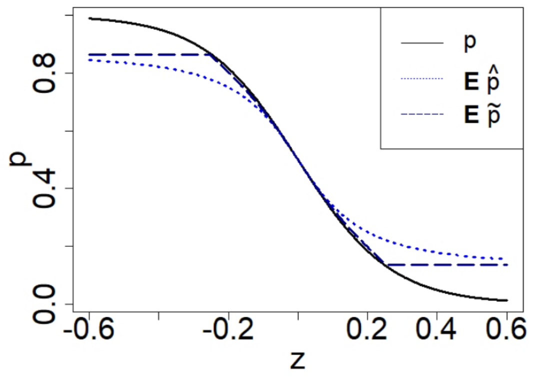

Figure 6: Difference between estimation using p (true gradient), $\tilde{p}$ (Barker proposal with noisy gradient) and $\hat{p}$ (corrected version with noisy gradient) [MZ23]. On the y-axis there is the acceptance probability of the three methods and on the x-axis there is the proposed z.

that takes into account this factor as follows:

$$\hat{\alpha} = \frac{1.702}{\sqrt{1.702^2 - z^2\tau^2(\theta)}} \tag{2.26}$$

$$\tilde{p}_\alpha(\hat{\nabla}g(\theta), z) = \frac{1}{1 + \exp(-z\alpha\hat{\nabla}g(\theta))} \tag{2.27}$$

The new definition can only be applied whenever $|z| < 1.702/\tau(\theta)$ and, as shown in figure 6, it has a positive impact on a significant range of values of z.

3 Results

In this section the results of the experiments will be presented, together with a detailed description of the networks, the datasets and the parameter tested.

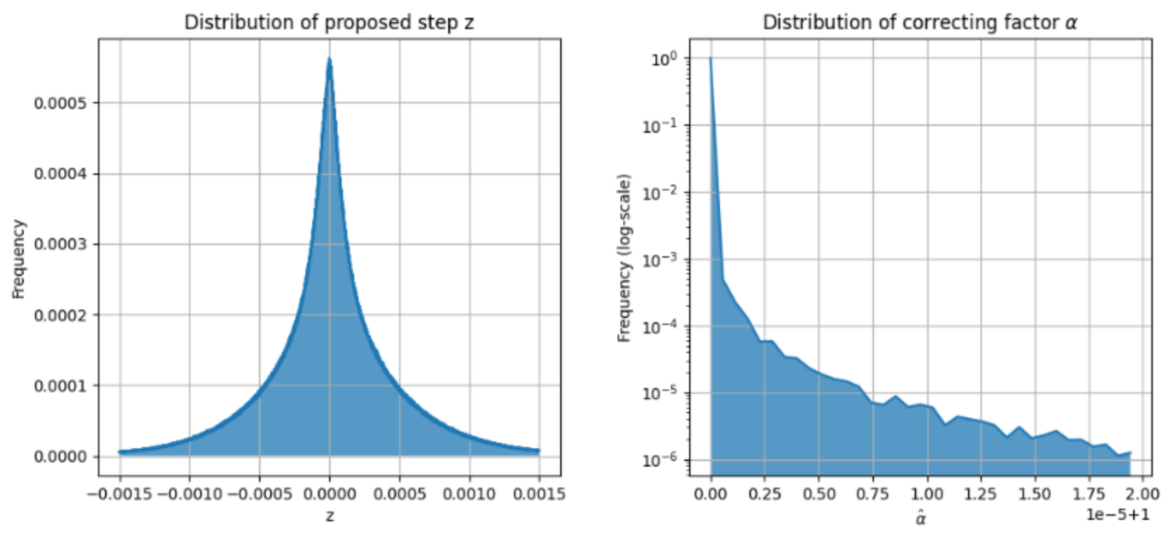

Figure 7: Empirical data of the distribution of the proposed steps z between epochs 4 and 6 (so once the training has stabilized and is quite stationary). On the y-axis there is the frequency for each bin (the number of bins is 10000 and 35 respectively). The distribution is very concentrated around 0 as already pointed out in the analysis of the order of magnitude and this affects also the correcting factor α. It is clear that the distribution of α is very concentrated around 1, therefore the correction ($z_c = \alpha z$) does not influence the acceptance probability.

3.1 Dataset

The models have been tested on MNIST and CIFAR10, two image classification datasets. MNIST consists of 60,000 28x28 gray-scale images of handwritten digits. The dataset has been very extensively studied and almost "solved", in the sense that very good classifiers have been written which achieve almost perfect accuracy, fairly easily above 99%. It has been used as a benchmark to establish if the algorithm was converging, for it has quite small images (28*28) and can be iterated over quickly. The other dataset, called CIFAR10, is a subset of CIFAR and consists of 60,000 32x32 colored pictures of various objects. CIFAR10 only contains the following 10 classes: [plane, car, bird, cat, deer, dog, frog, horse, ship, truck]. This dataset is significantly harder and scoring above 50% cannot be achieved via classic algorithms such as SVM or logistic regressions.

3.2 Methodology

The algorithm has been tested against Adam. The philosophy was to test as few hyperparameters as possible for both methods. Therefore, only a grid search on the learning rate and on weight decay has been performed. The best learning rate is always 0.001 which is also the default value and it is consistently better on all the models tested. The weight decay instead has been set to 0.01, and this is the only form of regularization applied.

3.3 Training of CNN

The first model that has been tested is a small convolutional neural network with architecture illustrated in figure 8. The total number of parameters is 204,186 when trained on CIFAR10.

It is observed that on 9 SGDB performs better than ADAM on this model. The accuracy of the ensemble is better than just using the last iteration of the epoch but it suffers from higher loss. Depending on the problem the two models can both be useful: often the network confidence in its answer is discarded as it has no practical use, hence, it is better to have higher accuracy.

When training larger models a different pattern is observed, as shown in figure 10. SGDB is capable of finding a better training accuracy, probably due to its ability to explore the parameter space, but the training loss starts to grow after very few epochs. This is a classic example of overfitting but nevertheless, the test accuracy is still higher. A possible explanation is that the model is more polarizing so the loss is going up but the accuracy is not catching this phenomenon.

One general consideration is that ADAM has been specifically designed to train neural networks, so it is expected to be able to find wider minima (i.e. better generalization error). On the other hand, SGDB can more easily run into overfitting issues as it is only trying

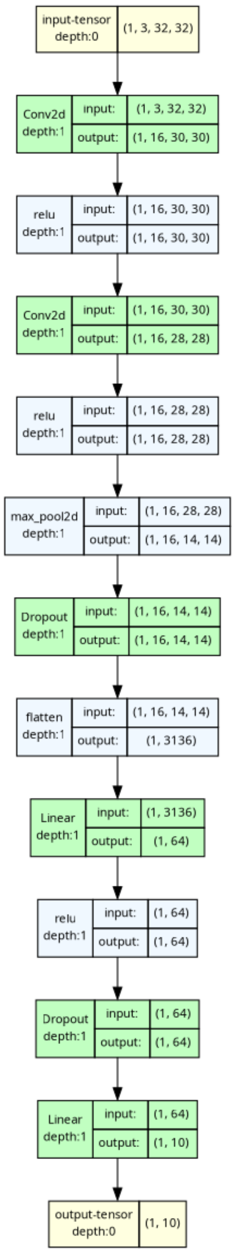
input-tensor depth:0 (1, 3, 32, 32)
Conv2d depth:1 input: (1, 3, 32, 32) output: (1, 16, 30, 30)
relu depth:1 input: (1, 16, 30, 30) output: (1, 16, 30, 30)
Conv2d depth:1 input: (1, 16, 30, 30) output: (1, 16, 28, 28)
relu depth:1 input: (1, 16, 28, 28) output: (1, 16, 28, 28)
max_pool2d depth:1 input: (1, 16, 28, 28) output: (1, 16, 14, 14)
Dropout depth:1 input: (1, 16, 14, 14) output: (1, 16, 14, 14)
flatten depth:1 input: (1, 16, 14, 14) output: (1, 3136)
Linear depth:1 input: (1, 3136) output: (1, 64)
relu depth:1 input: (1, 64) output: (1, 64)
Dropout depth:1 input: (1, 64) output: (1, 64)
Linear depth:1 input: (1, 64) output: (1, 10)
output-tensor depth:0 (1, 10)

Figure 8: Medium CNN architecture. The large model is similar, just with more layers and parameters

to minimize the training error by visiting as much of the parameter space as possible. This effect is more visible in the training of the bigger convolutional model in 10 and in the training of ResNet18 which required some special parameters to be set and will be discussed later. In this case, the model has multiple dropout layers as shown in figure 8, which help regularization and therefore SGDB is performing better.

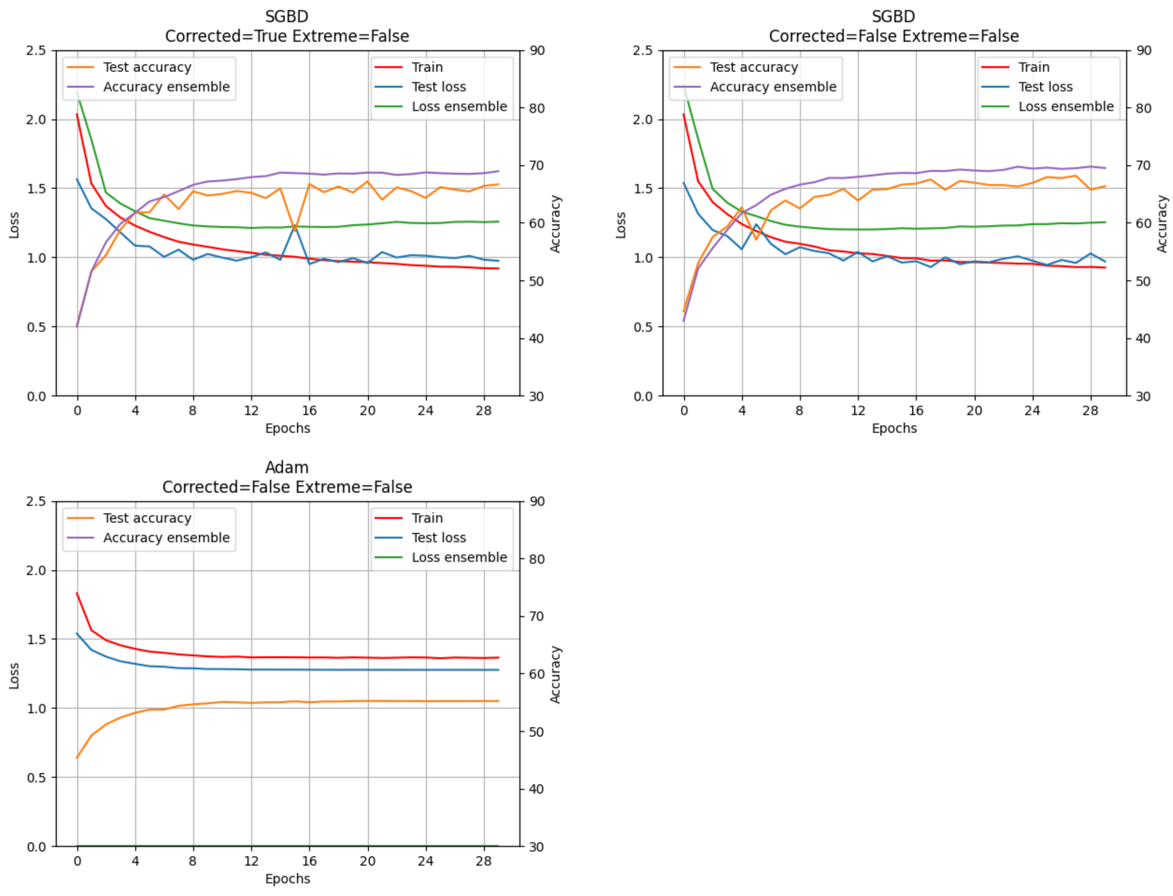

Figure 9: Training iterations of Cnn network on CIFAR10. Training parameters are all default for SGDB, for Adam lr=10^{-3}, weight_decay=10^{-2}

3.4 Running time

The overall running time of one epoch of SGDB is similar to Adam and other optimization algorithms. They both require the gradient to be computed, so the forward-pass and the backward-pass are executed in both cases and they take most of the time. One thing to note is that SGDB has some calculations that depend on the layers, so the number of parameters is not the only factor for speed as the depth/width ratio also matters. The computations of the corrected version suffer from less parallelization, therefore the ratios between GPU/CPU time are different.

Model	Algorithm	CPU	GPU
CnnMedium	Adam	13.2s	2.8s
CnnMedium	SGDB	10.2s	3.8s
CnnMedium	SGDB corrected	10.3s	5.5s
LargeCNN	SGDB	158.0s	26.0s

Table 2: Running time of Adam vs SGDB
CPU and GPU time are not comparable: the training was done on two different devices and operative systems, the CPU is an i5-1135G7 and the model was compiled with torch.compile while the GPU is a GTX-970M running on Windows. This benchmark does not include the time it takes to load data into memory. Furthermore, the time is quite stable and with a standard deviation of virtually 0

3.5 Convergence speed

The convergence speed of SGDB has been compared to the one of Adam with the following metric: after setting a threshold on the train accuracy (in this case 66.6%) the training has been repeated and the number of epochs required to reach it has been recorded. The results are that SGDB is very stable and on the LargeCnn it took 5 epochs in all the experiments ($N = 10$) while it took Adam 9 ± 0.63 ($N = 10$). This pattern is also reflected in other models, in general, it seems that SGBD can lower the train loss/accuracy faster and more will be commented on this when analyzing some specific results (especially on finetuning).

3.6 Training of Resnet

A model in computer vision that has been very popular is ResNet [He+15] which in 2015 reached State-of-the-Art performance on various datasets (including CIFAR10). Those results won them 1st place on the ILSVRC 2015 classification task. ResNet stands for

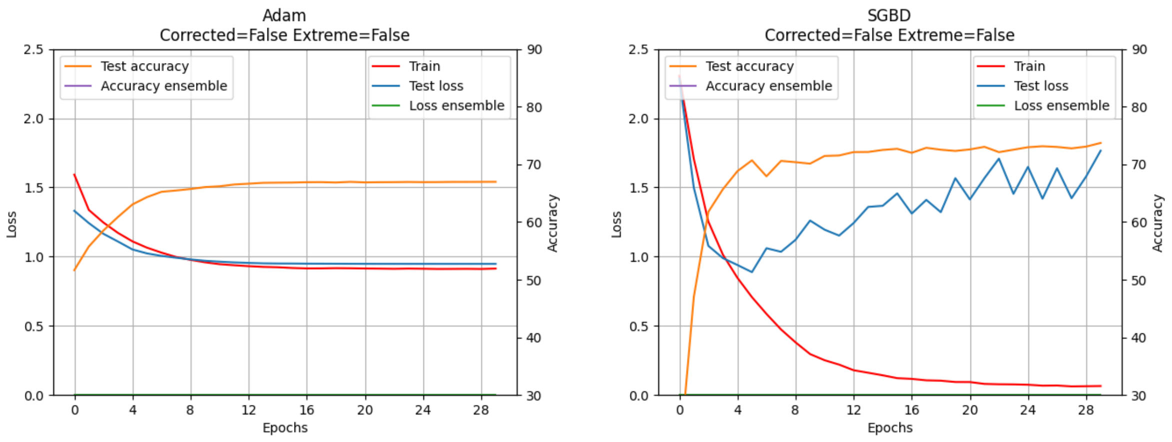

Figure 10: Training iterations of a large Cnn (12M parameters, 10 layers) network on CIFAR10. Training parameters are all default except a for SGDB, for Adam lr=10^{-3}, weight_decay=10^{-2}

Residual Networks and indicates the presence of skip connections and a very deep network. This depth can induce some issues and in fact, during the training of ResNet18 with SGDB a few problems arose, in particular, the temperature was diverging on certain layers (figure 11). The gradients on those last few layers have gradients that are too small and to compensate the temperature starts to go up until it reaches a threshold ($\sim 10^40$) at which the acceptance probability cannot be computed and just return Nan. To fix this issue a simple upper limit to the temperature has been added, imposing $\log(T) < 30$.

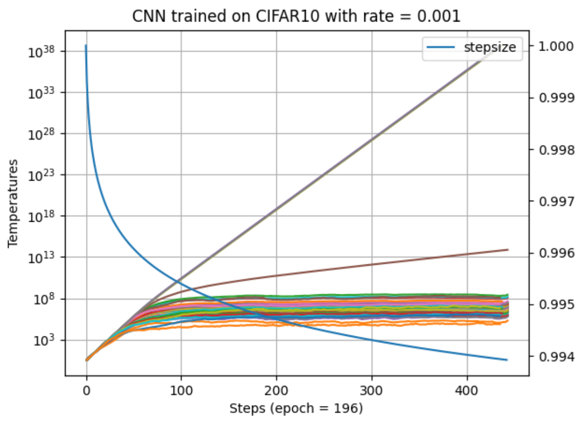

Figure 11: Temperature in ResNet18. It is clear that one of the temperatures keeps increasing without showing any sign of damping

Overall the training of ResNet seems slightly inferior to SGDB compared to Adam and it proposed a challenge due to the depth of the network. ResNet seemed to be too unstable, usually when one sees noise like in figure 12 at stepsize=1 the solution is to reduce the learning rate. In the way the SGDB algorithm is formulated there is no learning rate (or it is implicitly 1) so a modified version has been used and instead of updating $\theta_t = \theta_{t-1} + z * b$ it uses:

$$\theta_t = \theta_{t-1} + \sigma z * b \tag{3.1}$$

So that all the temperatures and accepting probabilities still work. Another possibility is to scale z by the learning rate σ but empirically it seems less stable.

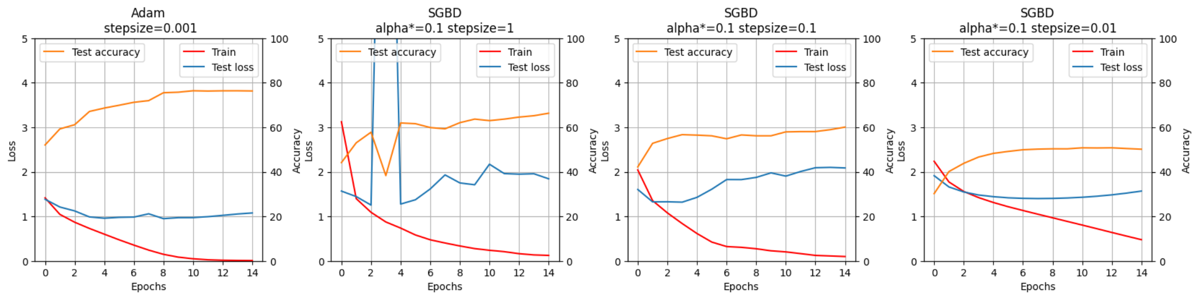

Figure 12: Different training of ResNet18. On the left there is Adam, used as a baseline comparison, the other 3 are SGDB with different stepsizes. The one that performed better was the default stepsize=1 but it seems very unstable. Smaller stepsizes do not perform as well. Using values lower than 0.01 could not make the network converge in a reasonable amount of epochs.

Despite the different behaviors at different learning rates, the overall pattern is that rates close to 1 seem to work better and below 0.01 it just is not able to learn. In contrast to Adam where the standard learning rate is quite lower and the algorithm is more susceptible (other algorithms such as SGD are even more delicate to tune). It is worth noting that for the other CNN changing the learning rate did not yield any significant result (better accuracy was achieved slightly below 1) and the better results are probably just due to noise and multiple hypothesis testing effects.

3.7 Finetuning on Resnet

A training pattern that has recently become very popular is finetuning [cite]. The idea consists of taking an already existing model which has been trained on a large dataset and then re-training (fine-tuning) it for a few epochs on a more specific problem of interest. The advantages are not only the lower computational cost due to most of the training already completed by someone else but also the higher performance of those networks: given the larger amount of patterns they have seen, they tend to generalize better. Another key factor is the number of open-source models available which is steadily increasing. The model that has been tested with SGDB is Resnet18 with weights pre-trained on ImageNet1K [Den+09]. Due to the weights already being in a good position in the parameter space one does not want to change them too much so it is usually advised to lower the learning rate of gradient descent.

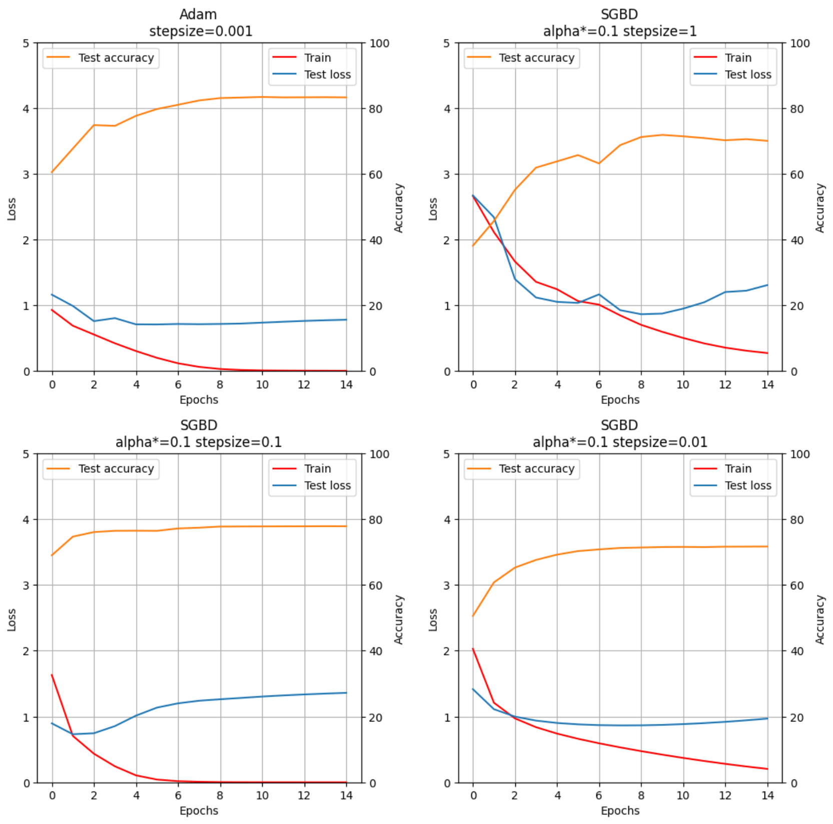

Figure 13: Different training on ResNet18 starting from the model pretrained on ImageNet1k.

Looking at figure 3.7 there are a few things to notice: the optimal stepsize is 0.1 and the training error goes down faster than Adam but the generalization is worse. The first effect supports the idea that keeping the stepsize at 1 as a default value is sensible and that the order of magnitude of the step is $O(1)$. The second effect has multiple possible explanations and probably it is a result of a combination of different factors. The main point is that both algorithms find a parameter configuration such that every image in the training set is correctly classified and with good confidence (cross-entropy loss < 0.01) and SGBD reaches those values faster but the generalization error (test loss) has an opposite pattern. This suggests that SGDB is more overfit-prone compared to Adam. ResNet does not have any dropout layer while on the other custom CNN architecture there were multiple with significant dropout rates (0.25 and 0.5).

4 Conclusions

The Stochastic Gradient Barker Dynamics Algorithm has been tested in an optimization setting in which multiple Neural Networks were trained. It showed excellent training potential and some technical challenges. Results indicate that the performance of SGBD can overall be better/on par with Adam. The ability of the algorithm to visit the parameter space makes it perform consistently better on the train loss/accuracy therefore it is more suited to finding minima. On the contrary, it incurred into higher generalization errors in Networks without enough regularization, which gives some insight into which networks can benefit from training with SGDB.

It has also been demonstrated that it is possible to remove certain hyperparameters (stepsize and temperature) and automatically tune them, thus suggesting that it might be possible to further automatize the hyperparameter tuning of the algorithm.

Given the good performance on well-regularized newtorks a possible continuation, path would be to introduce regularization in the algorithm itself. Up until now, only a gradient clip has been applied but introducing a weight decay can be possible.

Appendix

The code can be found in the public GitHub repository at the following link `https://github.com/DarioFi/Thesis`. The repository consists of a series of utility functions to run tests, collect data, and plot results. The actual algorithm is in the file SGBD/optimizer.py. All the files are commented and the experiments are all reproducible with this source code.

References

[HH89] Stinchcombe M. Hornik K. and White H. "Multilayer feedforward networks are universal approximators. Neural Networks". In: (1989). URL: `https://doi.org/10.1016/0893-6080(89)90020-8`.

[CG95] Siddartha Chib and Edward Greenberg. *Understanding the Metropolis Hasting algorithm.* 1995. URL: `https://eml.berkeley.edu/reprints/misc/understanding.pdf`.

[BV04] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization.* 2004. URL: `https://stanford.edu/~boyd/cvxbook/`.

[AT08] Christophe Andrieu and Johannes Thoms. "A tutorial on adaptive MCMC". In: (2008).

[Den+09] Jia Deng et al. "Imagenet: A large-scale hierarchical image database". In: *2009 IEEE conference on computer vision and pattern recognition.* Ieee. 2009, pp. 248–255.

[Sut+13] Ilya Sutskever et al. "On the importance of initialization and momentum in deep learning". In: *Proceedings of the 30th International Conference on Machine Learning.* Ed. by Sanjoy Dasgupta and David McAllester. Vol. 28. 2013, pp. 1139–1147. URL: `https://proceedings.mlr.press/v28/sutskever13.html`.

[KB14] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization.* 2014. eprint: `arXiv:1412.6980`.

[He+15] K. He et al. "Deep Residual Learning for Image Recognition." In: (2015). eprint: `ArXiv./abs/1512.03385`.

[BJV17] Fetsje Bijma, Marianne Jonker, and Aad van der Vaart. *An Introduction to Mathematical Statistics.* 2017.

[M+17] Stephan M et al. "Stochastic Gradient Descent as Approximate Bayesian Inference". In: *Journal of Machine Learning Research* 18.134 (2017), pp. 1–35. URL: `http://jmlr.org/papers/v18/17-214.html`.

[DC18] Masters D. and Luschi C. "Revisiting Small Batch Training for Deep Neural Networks." In: (2018). eprint: `arXiv:1804.07612`.

[Zha+19] Jingzhao Zhang et al. *Why gradient clipping accelerates training: A theoretical justification for adaptivity.* 2019. eprint: `arXiv:1905.11881`.

[Wen+20] F. Wenzel et al. "How Good is the Bayes Posterior in Deep Neural Networks Really?" In: (2020). eprint: `ArXiv./abs/2002.02405`.

[Izm+21] P. Izmailov et al. "What Are Bayesian Neural Network Posteriors Really Like?" In: (2021). eprint: `ArXiv./abs/2104.14421`.

[MZ23] Lorenzo Mauri and Giacomo Zanella. "Stochastic Gradient Barker Dynamics. In preparation." In: (2023+).