NEW APPROACH TO GAN OPTIMIZATION

Egor Gladin
egor.gladin@skoltech.ru

Abduragim Shtanchaev
abduragim.shtanchaev@skoltech.ru

Aleksei Pronkin
aleksei.pronkin@skoltech.ru

Vyacheslav Rezyapkin
vyacheslav.rezyapkin@skoltech.ru

June 7, 2020

ABSTRACT

Adam and Stochastic Gradient Descent are ubiquitous in deep learning. We consider alternative methods for neural networks optimization. Lan's Gradient Sliding, The Ellipsoid method and QuickProp are examined as tools for training neural networks (generative adversarial networks in particular). These methods are not present in PyTorch library – we implement them from scratch. The performance is compared to that of Adam and SGD.

***K*eywords** GAN · Optimization · Adam · SGD · Gradient Sliding · Ellipsoid method · QuickProp

1 Introduction

Deep Learning is essentially optimization of loss function with respect to the parameters (or weights) of a neural network. This problem is usually solved by first-order methods, i.e. algorithms that require computation of gradient of target function at each step. Loss function is generally a sum of a large number terms, each term corresponding to a particular training example. Thus, computation of the exact gradient would be an extremely complex task. Training of neural networks is therefore performed in a stochastic fashion. At each step, an unbiased approximation of the gradient is used which is computed over a mini-batch, i.e. over a relatively small number of terms. Stochastic Gradient Descent (SGD) and Adam [1] are probably the most popular algorithms for training neural networks. However, there is a great number of other first-order methods.

In this project we consider Lan's Gradient Sliding [2], Quickprop optimizer [3] and the Ellipsoid Method (idea was introduced in the paper by Naum Z. Shor in 1972 [4], after that this method was studied for real convex functions [5], [6], [7], [4], for quick reference see e.g. §2.2 [8] or 891 - 899 pages [9]).

Training of Generative Adversarial Networks (GANs) is formulated as two optimization problems:

$$\begin{aligned}
&1)\mathcal{L}_D = -\mathbb{E}_{x\sim p(x)} \log D(x) - \mathbb{E}_{z\sim n(z)} \log\big(1 - D(G(z))\big) \to \min_D,\\
&2)\mathcal{L}_G = -\mathbb{E}_{z\sim n(z)} \log D(G(z)) \to \min_G,\\
&\text{where } \; p(x) \text{ is true distribution of data,}\\
&\qquad\quad n(x) \text{ is noise distribution (we took gaussian),}\\
&\qquad\quad D \text{ is discriminator network,}\\
&\qquad\quad G \text{ is generator network,}
\end{aligned} \tag{1}$$

Optimization process consists of alternating between this two problems. Widely used approaches to this alternating usually have no theoretical guarantees. The idea to use gradient sliding (GS) to solve this problem appeared due to the fact that GS is specifically designed for optimization of a sum of two functions, which potentially make it suitable for training a discriminator.

2 Methods

2.1 Gradient Sliding

Let us first introduce the following notation: $V(x,z)$ denotes Bregman's distance [10]. In our case, $V(x,z) = \frac{1}{2}\|x-z\|^2$. For some function f, denote $l_f(x;y) := f(x) + \langle \nabla f(x), y-x\rangle$.

Consider composite convex optimization problem of the form

$$\Psi^* \equiv \min_{x\in X}\{\Psi(x) := f(x) + h(x)\} \tag{2}$$

Here, $X \subseteq \mathbb{R}^n$ is a closed convex set, $f : X \to \mathbb{R}$ and $h : X \to \mathbb{R}$, respectively, are general smooth and nonsmooth convex functions satisfying

$$f(x) \le f(y) + \langle \nabla f(y), x-y\rangle + \frac{L}{2}\|x-y\|^2, \quad \forall x, y \in X \tag{3}$$

$$h(x) \le h(y) + \langle h'(y), x-y\rangle + M\|x-y\|, \quad \forall x, y \in X \tag{4}$$

for some $L > 0$ and $M > 0$, where subgradient $h'(x) \in \partial h(x)$.

We assume that we can compute the exact gradient $\nabla f(x)$ and a subgradient $h'(x) \in \partial h(x)$ for any $x \in X$. Gradient Sliding (Algorithm 1) requires $\mathcal{O}(\sqrt{\frac{L}{\epsilon}})$ evaluations for ∇f and $\mathcal{O}\left(\sqrt{\frac{L}{\epsilon}} + \frac{M^2}{\epsilon^2}\right)$ evaluations for h' to find an ϵ-solution of (2), i.e. a point $\bar{x} \in X$ s.t. $\Psi(\bar{x}) - \Psi^* \le \epsilon$. The basic idea of this algorithm is to skip the computation of $\nabla f(x)$ from time to time. Specific parameters for algorithm 1 are provided by the following theorem:

Theorem 1. *Assume that $\{p_t\}$ and $\{\theta_t\}$ in the PS procedure are set to*

$$p_t = \frac{t}{2} \quad \text{and} \quad \theta_t = \frac{2(t+1)}{t(t+3)}, \quad \forall t \ge 1$$

Also assume that there exists an estimate $D_X > 0$ s.t. $V(x,y) \le D_X^2$ $\forall x, y \in X$. If $\{\beta_k\}, \{\gamma_k\}$, and $\{T_k\}$ are set to

$$\beta_k = \frac{9L\left(1 - P_{T_k}\right)}{2(k+1)}, \quad \gamma_k = \frac{3}{k+2}, \quad \text{and} \quad T_k = \left\lceil \frac{M^2(k+1)^3}{\tilde{D}L^2} \right\rceil \tag{5}$$

where $P_t = \frac{2}{(T_k+1)(T_k+2)}$ and $\tilde{D} = 81D_X^2/16$, then

$$\Psi\left(\bar{x}_N\right) - \Psi\left(x^*\right) \le \frac{2L}{N(N+1)}\left[3V(x_0, x^*) + 2\tilde{D}\right], \quad \forall N \ge 1$$

Algorithm 1 The Gradient Sliding (GS) algorithm

```
Input: Initial point $x_0 \in X$, iteration limit $N$, parameters $\{\gamma_k\}$, $\{T_k\}$, $\{\beta_k\}$.
1: $\bar{x}_0 := x_0$
2: for $k = 1, 2, \dots, N$ do
3:     $\underline{x}_k := (1-\gamma_k)\bar{x}_{k-1} + \gamma_k x_{k-1}$,
4:     $g_k(\cdot) := l_f(\underline{x}_k, \cdot)$,
5:     $(x_k, \tilde{x}_k) = \mathrm{PS}(g_k, x_{k-1}, \beta_k, T_k)$  (see procedure 2),
6:     $\bar{x}_k = (1-\gamma_k)\bar{x}_{k-1} + \gamma_k \tilde{x}_k$,
7: end for
Output: $\bar{x}_N$.
```

If PS procedure is called at k-th step of GS algorithm, then on t-th iteration of this procedure u_t is calculated as follows:

$$u_t = \operatorname{argmin}_{u\in X}\left\{g(u) + l_h\left(u_{t-1}, u\right) + \beta V(x,u) + \beta p_t V\left(u_{t-1}, u\right)\right\}. \tag{6}$$

Let us derive the exact solution. Observe that

$$g(u) = l_f(\underline{x}_k, u) := f(\underline{x}_k) + \langle \nabla f(\underline{x}_k), u - \underline{x}_k\rangle = \langle \nabla f(\underline{x}_k), u\rangle + const,$$

$$l_h\left(u_{t-1}, u\right) := h(u_{t-1}) + \langle \nabla h(u_{t-1}), u - u_{t-1}\rangle = \langle \nabla h(u_{t-1}), u\rangle + const,$$

Procedure 2 The Prox-Sliding (PS) procedure

```
Input: $g,\ x,\ \beta,\ T$, parameters $\{p_t\}$, $\{\theta_t\}$.
  $u_0 := x,\ \tilde{u}_0 := x,$
  for $t = 1, 2, \ldots, T$ do
      $u_t = \operatorname{argmin}_{u \in X} \{g(u) + l_h(u_{t-1}, u) + \beta V(x, u) + \beta p_t V(u_{t-1}, u)\}$
      $\tilde{u}_t = (1-\theta_t)\tilde{u}_{t-1} + \theta_t u_t,$
  end for
Output: $(u_T, \tilde{u}_T)$.
```

$$V(x,u) = \frac{1}{2}\|x_{k-1} - u\|_2^2, \quad V(u_{t-1}, u) = \frac{1}{2}\|u_{t-1} - u\|_2^2$$

Thus,

$$u_t = \operatorname{argmin}_{u \in X} \left\{ \langle \nabla f(\underline{x}_k), u\rangle + \langle \nabla h(u_{t-1}), u\rangle + \frac{\beta}{2}\|x_{k-1} - u\|_2^2 + \frac{\beta p_t}{2}\|u_{t-1} - u\|_2^2 \right\},$$

This expression can be minimized analytically

$$\nabla_u \left(\langle \nabla f(\underline{x}_k), u\rangle + \langle \nabla h(u_{t-1}), u\rangle + \frac{\beta}{2}\|x_{k-1} - u\|_2^2 + \frac{\beta p_t}{2}\|u_{t-1} - u\|_2^2 \right)\Bigg|_{u_t} = 0$$

$$\nabla f(\underline{x}_k) + \nabla h(u_{t-1}) + \beta(u_t - x_{k-1}) + \beta p_t(u_t - u_{t-1}) = 0$$
$$\beta u_t(1 + p_t) = \beta(x_{k-1} + p_t u_{t-1}) - \nabla f(\underline{x}_k) - \nabla h(u_{t-1})$$

$$u_t = \frac{\beta(x_{k-1} + p_t u_{t-1}) - \nabla f(\underline{x}_k) - \nabla h(u_{t-1})}{\beta(1 + p_t)}. \tag{7}$$

When applied as a tool for training deep neural networks, gradient sliding faces some difficulties. First, loss as a function of net's parameters is not convex in general. It is well-known that finding a global minimum of non-convex function is an unsolvable problem in general, and local minimum is the best we can hope for, as usual in deep learning. Second, to calculate recommended parameters for the method, we need to know Lipschitz constants L and M. It is usually impossible to find them analytically, and one needs to test different values for a particular problem. However, this two problems are common for optimization methods applied to deep learning tasks.

2.2 Ellipsoid Method

In mathematical optimization ellipsoid algorithm is iterative method for finding a minimum of a convex function. Preliminary versions of Ellipsoid method were available before the method was studied by Arkadi Nemirovski and David B. Yudin who improved the method and proved polynomial-time solvability of linear programs. The method iteratively generates a sequence of ellipsoids whose volume decreases uniformly at every iteration ensuring that a minimizer of a convex function is always within a volume of smaller ellipsoid. A concise mathematical description follows as:

Ellipsoid $\mathcal{E}$ is a convex set defined by center c and matrix H: $\mathcal{E} = \{x \in \mathbf{R}^n | (x-c)^T H^{-1}(x-c) \leq 1\}$.

Firstly, we need to choose $\mathcal{E}$ that surely contains our optimal point x^*. Then, from the cutting-plane oracle we obtain a vector w such that $w^T(x^* - c) \leq 0$.

After that we could take "smaller" (in terms of volumes) ellipsoid $\mathcal{E}_0$ that contains our optimal solution (that we don't know).

$$\operatorname{vol}(\mathcal{E}_0) \leq \exp\left(-\frac{1}{2n}\right)\operatorname{vol}(\mathcal{E})$$
$$\mathcal{E}_0 = \{x \in \mathcal{E}_0 \mid (x - c_0)^T H_0^{-1}(x - c_0) \leq 0\}$$

We update H_0 and c_0 with the following formulas:

$$c_0 = c - \frac{1}{n+1}\frac{Hw}{\sqrt{w^T H w}}$$

$$H_0 = \frac{n^2}{n^2 - 1}\left(H - \frac{2}{n+1}\frac{Hww^T H}{w^t H w}\right)$$

Finally, we take $\mathcal{E}_0$ as $\mathcal{E}$ and proceed these steps until convergence.

Algorithm 3 Ellipsoid method for minimization of function $v(y)$

```
Input: Number of iterations $N \geq 1$, sphere $\mathcal{B}_{\mathcal{R}} \supseteq Q_y$, its center $c$ and radius $\mathcal{R}$.
 1: $\mathcal{E}_0 := \mathcal{B}_{\mathcal{R}}, \quad H_0 := \mathcal{R}^2 I_n, \quad c_0 := c.$
 2: for $k = 0, \ldots, N-1$. do
 3:     if $c_k \in Q_y$ then
 4:         $w_k := w \in \partial_\delta v(c_k)$
 5:         if $w_k = 0$ then
 6:             $\tilde{y} := c_k$
 7:             return $\tilde{y}$
 8:         end if
 9:     else
10:         $w_k := w$, where $w \neq 0$ such that $Q_y \subset \{y \in \mathcal{E}_k : w^T(y - c_k) \leq 0\}$
11:     end if
12:     $c_{k+1} := c_k - \frac{1}{n+1} \frac{H_k w_k}{\sqrt{w_k^T H_k w_k}}$
        $H_{k+1} := \frac{n^2}{n^2-1} \left( H_k - \frac{2}{n+1} \frac{H_k w_k w_k^T H_k}{w_k^T H_k w_k} \right)$
        $\mathcal{E}_{k+1} := \{y : (y - c_{k+1})^T H_{k+1}^{-1} (y - c_{k+1}) \leq 1\}$
13: end for
Output: $y^N = \arg \min_{y \in \{c_0, \ldots, c_N\} \cap Q_y} v(y)$
```

We use for every layer of the network new ellipsoid. This approach greatly reduce memory consumption and speed up all computation.

2.3 QuickProp Method

Quickprop is a mathematical optimization technique, invented by Scott E. Fahlman in 1988 [3]. It is an interactive algorithm for optimizing a mathematical function, and it was inspired by Newton's method. Although Newton's approach provides more luxurious information about a function's curvatures by quadratic approximation, its main drawback is a complexity which equals $\mathcal{O}(n^2)$. Due to its complexity, the method is not preferred in large optimization problems such as Neural Networks.

There are two main approaches to finding more precise information about the curvature of a function. The first one is to adjust the learning rate dynamically, either globally or locally, for each weight. That could be done based on some heuristics of computation history. The standard momentum term used in some optimizers is a form of that strategy. Learning rate schedules are also of that kind, though they are based on the programmer's experience rather than the network itself. The second approach is to use second-order approximators of error with respect to weights. Unfortunately, these methods require a vast amount of computation and thus are expensive for large scale optimization problems.

In the light of above mentioned complications Scott E. Fahlman came up with a method that inherits from both approaches mentioned above – Quickprop. It is a second-order method that does not require explicit computation of the second-order derivate, thus it is more heuristic than formal. The only difference from the vanilla SGD is that a copy of error $\frac{\partial E}{\partial w^{t-1}}$ from a previous step is kept and used in every iteration. Also, a difference between current and previous values of weights are kept. The update step is straightforward:

$$\Delta w_{ij}^t = \Delta w_{ij}^{t-1} \frac{\Delta L^t}{\Delta L^{t-1} - \Delta L^t},$$
$$w_{ij}^t = w_{ij}^t + \alpha \Delta w_{ij}^t,$$

where ΔL^t is a current gradient and α is a learning rate

3 Implementation

GitHub repository: gan-optimization

For implementation and experiments we chose Pytorch deep learning library. Our choice is particularly supported by the features that make Pytorch library able to do autogradient computations with backward mechanism and an interface for optimization algorithms. In pytorch, optimization algorithms have to be inherent as torch.optim.Optimizer children to provide compatibility with torch neural network class. Two main methods that were implemented are *.__init__()* and *.step()*.

3.1 Gradient Sliding (`optim/grad_sliding.py`)

__init__ takes as input model parameters (weights in case of neural networks) with respect to which optimization is to be done, and algorithm parameters: L, M and $\widetilde{D}$ (see (3), (4) and theorem 1). *step* is the method that does all necessary calculations for weight updating.

Our implementation of *step* method consists of two stages: main stage and proximal sliding (PS). In the first stage, first-order approximation of f is calculated and in the second it is used to make optimization procedure. According to procedure 2, PS is a loop with T iterations. It calculates derivatives of h with respect to u. Since algorithm has two stages and gradients are calculated at different points during these stages, model's active parameters depend on current stage: $\underline{x}$ at main stage and u at PS stage. However, output of the algorithm (i.e. trained parameters) is point $\bar{x}$. To switch model's parameters between different points, we have implemented *to_train* and *to_eval* methods which are supposed to be called in training loop where it is desired to change function estimation behaviour – call *to_eval* before evaluation. To control the current stage of algorithm we have counter variables k and t.

3.2 Ellipsoid Method (`optim/ellipsoid_sgd.py`)

Ellipsoid method based on Pytorch SGD class and inherits all the features of SGD (momenta, Nesterov momentum, dampening, weight decay)

3.3 Quickprop (`optim/vanilla_quickprop_sgd.py`)

In the implementation of Quickprop algorithm two main functions of the base class torch.optim.Optimizer have been overridden. These are *__init__()* and *step()* fuctions. *.__init__()* function takes in four parameters. First parameter is iterable of a model weights, second is learning rate, third is weight decay and the last is epsilon – value used for stability in fractions. *step()* function contains the update rule and keeps previous steps' values that are needed for computing the update.

4 Experiments

4.1 Gradient sliding

4.1.1 Linear Regression (`grad_sliding_regression.ipynb`)

Consider linear model without bias: $g(x) = Ax$, where $A \in \mathbb{R}^{m\times n}$ is a feature matrix, $x \in \mathbb{R}^n$ is a vector of model's parameters. We use root-mean-square error (RMSE) and L_2-regularization:

$$h(x) = \sqrt{\frac{1}{m}\|g(x) - b\|^2} = \frac{1}{\sqrt{m}}\|Ax - b\|, \quad f(x) = \frac{\lambda}{n}\|x\|^2.$$

For such setup, constants L and M can be found analytically (see derivation in the notebook):

$$M = \frac{2}{\sqrt{m}}\|A\|, \quad L = \frac{2\lambda}{n}$$

Derivation uses Lemma 2 from [11], which states the following: if convex function h is Lipschitz continuous with parameter M_h (i.e. $|h(x) - h(y)| \le M_h\|x - y\|$ for any $x, y \in X$), then condition (4) holds for $M = 2M_h$. The reason that we chose RMSE instead of MSE is that MSE is not Lipschitz continuous. As we can see from figure 1, GS works better than SGD at first few iterations and as good as SGD at the subsequent iterations.

Next, we consider multilayer perceptron (MLP) instead of linear model. We use 2 hidden layers and ReLU activations. It is difficult to calculate L and M for such a network. We tried to use the same parameters as for linear model, which lead to instable convergence. After we multiplied both parameters by 1.5, the got good results (see figure 2).

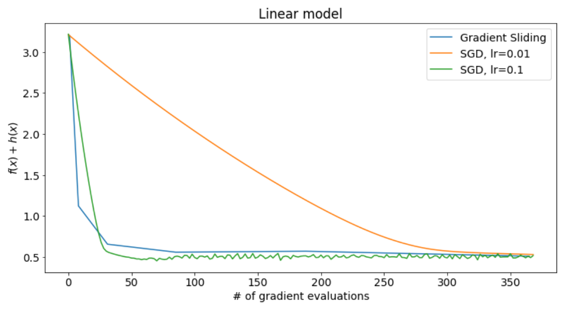

Figure 1: Convergence of GS and SGD for linear model.

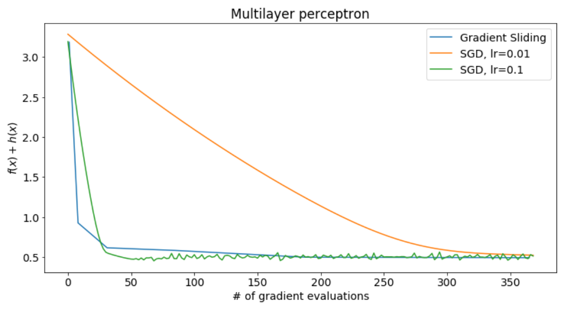

Figure 2: Convergence of GS and SGD for MLP.

4.1.2 Generative Adversarial Networks (`1d_gan.ipynb`)

Our next experiment was related to GAN optimization problem (1). In our experiment setting we used Adam optimizer for training generator and tried different optimizers for training discriminator: Adam and GS with different choice of f and h. We didn't observe convergence of GS algorithm in any setting. We suppose that the reason could be in mentioned above problem of estimating constants L, M for neural networks. We tried out different values bearing in mind the following considerations: according to (5), the larger the ratio L/M is, the less iterations of PS procedure will be done on each iteration of main loop. Therefore, the algorithm won't concentrate solely on minimization of h and will update gradients of f more frequently. Moreover, large ratio L/M implies large β, which means that solution of sub-task (6) won't result in a large step. This makes a lot of sense: if f is not very smooth, we shouldn't trust its linear approximation too much. We mostly tried to keep L larger than M to make sure both terms in the loss get enough attention. However, this didn't lead to convergence. Another possible reason for negative result is the fact that GAN training is min-max problem (in general, not convex-concave) which implies additional difficulties. But we have some progress in this task: if we set $f = h = \frac{1}{2}\mathcal{L}_D$ then GS shows convergence and works well. In this setting we do not use GS benefits in composite acceleration but we see that the algorithm have perspectives for further research.

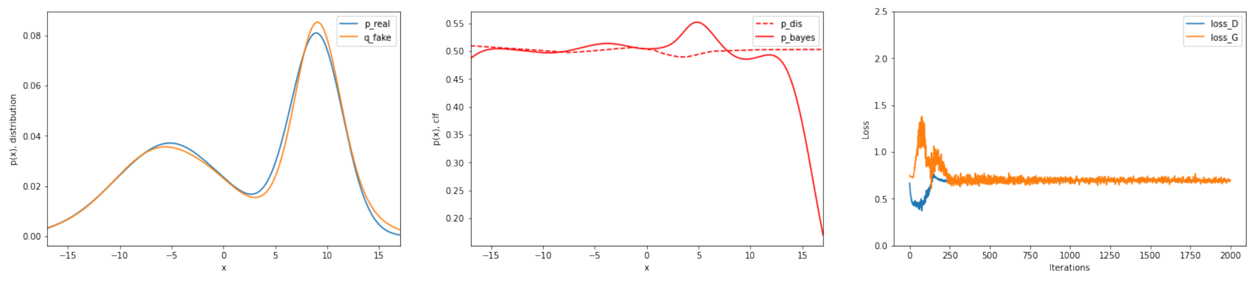

Figure 3: Adam

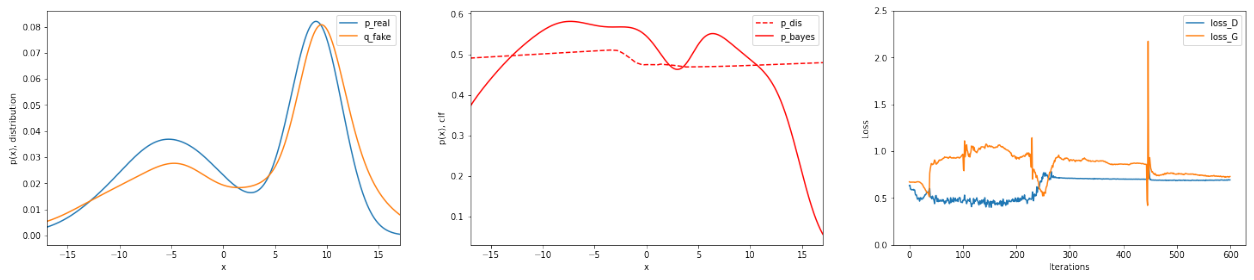

Figure 4: GS with $f = h$

4.2 Ellipsoid method

Results of ellipsoid algorithm are poor. The reasons could be: mistakes in implementation, our trick that drops multi-level connections.

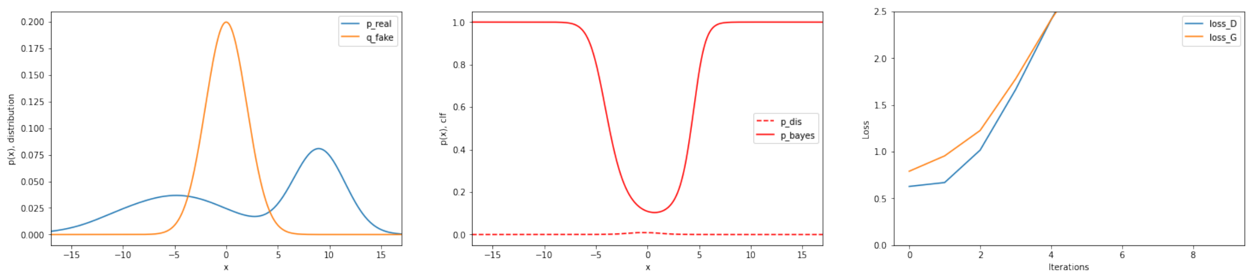

Figure 5: GS

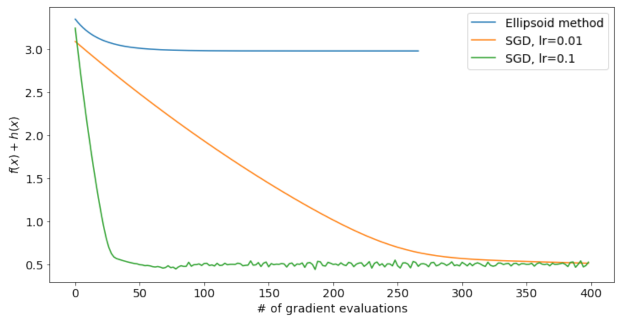

Figure 6: Ellipsoid method for linear regression

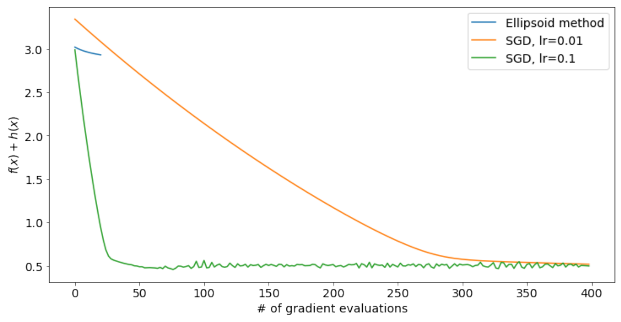

Figure 7: Ellipsoid method for MLP

4.3 Future work

- Ellipsoid algorithm
  - Try kronecker-factored approximate curvature for ellipsoid to reduce memory consumption [12]
  - Sparse linear algebra
  - Adapt gradient checkpointing for large batch sizes (solve problem of the high stochasticity of the gradient)
- Gradient Sliding
  - Develop algorithms to calculate lipschitz constants for arbitrary function and loss gradients.
- Try particle swarm optimization [13], Resilient Propagation (RProp) [14]
- Add natural gradient / second order methods
- Add NAS (neural architecture search) like optimizers. (That compute which parameters to optimize based on machine learning)
- Add loss surface visualisation [15] [16]

4.4 Quickprop

As experiment set for the Quickprop algorithm were have chosen a Multiplayer Perception and simple GAN architecture. MLP had two layer with hidden size being 100 units. ReLU was used as a non-linearity between layers. A criterion for loss is chosen MSE. For GAN we had generator and discriminator with 4 layer. Highest number of neurons in the hidden layer is 32. ReLU non-linearity was employed in between the layer and a sigmoid at the end of the discriminator. The job of the GAN was to reconstruct a particular Gaussian like distribution. The criterion was following:

$$\mathcal{L}_D = \mathbb{E}_{x\sim p(x)} \max\big(0, 1 - D(x)\big) + \mathbb{E}_{z\sim p(z)} \max\big(0, 1 + D(G(z))\big)$$
$$\mathcal{L}_G = -\mathbb{E}_{z\sim p(z)}\, D(G(z))$$

4.4.1 Multiplayer Perceptron (`QuckpropMLP.ipynb`)

Figure 8 illustrates the loss of vanilla version of quickprop algorithm for MLP, with a learning rate of 1e-6. The comparison is made with Adam with default parameters and the learning rate of 1e-3 and SGD with the learning rate of

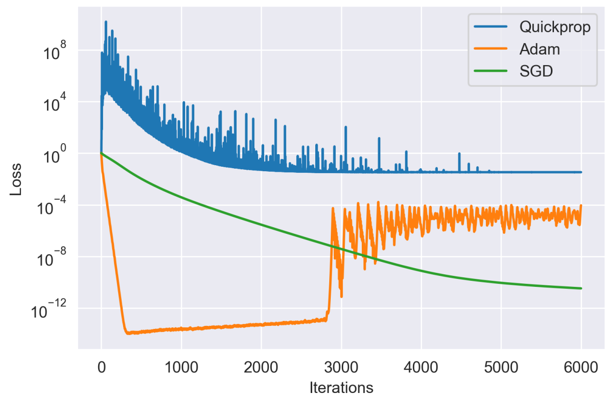

Figure 8: Quickprop vanilla

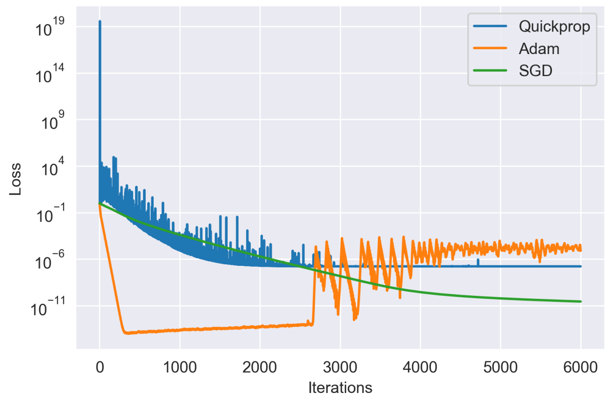

Figure 9: Improved Quckprop

1-2e. Although the quick prop algorithm has heuristic second-order approximation, it's clear that the vanilla quickprop algorithm performs quite poorly, even compared to SGD. Increasing the learning rate for the quickprop algorithm has shown to give even worse results. The reason for such poor performance is due to the initial conditions of the algorithm. At the zeroth iteration, there is no information available about the previous step; thus, synthetic information about the last step had to be used. To mitigate that, we used the "warming-up" scheme. For the first couple of interactions, we used a simple vanilla SGD update rule and kept saving information about every last gradient as well as a difference between the previous and current weight. Then, we switch to quickprop update rule as valid information about previous steps is available. Figure 9 illustrates the improvement of the results provided by the "warm-up" scheme.

Looking at the figures, one might have noticed the oscillations, and in particular, an out-shoot of the loss at the beginning of the training process. To mitigate these oscillations, the original paper suggests introducing a new parameter μ - "maximum growth factor". No weight step is allowed to be greater in magnitude than μ($\mu = 1.75$ in the original paper) times the previous step for that weight; if the step computed by the quickprop formula would be too large, infinite, or uphill on the current slope, we instead use μ times the previous step as the size of the new step. The proposed scheme did not give any improvements in our case.

4.4.2 Generative Adversarial Networks (`Gaussian_GAN_QuickProp.ipynb`)

For the experiments with Generative Adversarial Networks, we have considered three different cases. Firstly, we analyze Adam optimizer, then vanilla SGD, and lastly, we analyze and compare the Quickprop algorithm. All hyperparameters for optimizes are as in the section of Multiplayer Perceptron. The architecture of generator and discriminator are described in the section 4.4

Figure 10 illustrates three plots. The first plot show the generated distribution and actual distribution. The second plot stands for KDE of real distribution and generated one. The last plot shows values loss function verses number of iterations. As we can see the reconstruction alomostly perfectly fits the original data distribution and loss value converge after some iterations. For our experiments this is considered as a good reconstruction.

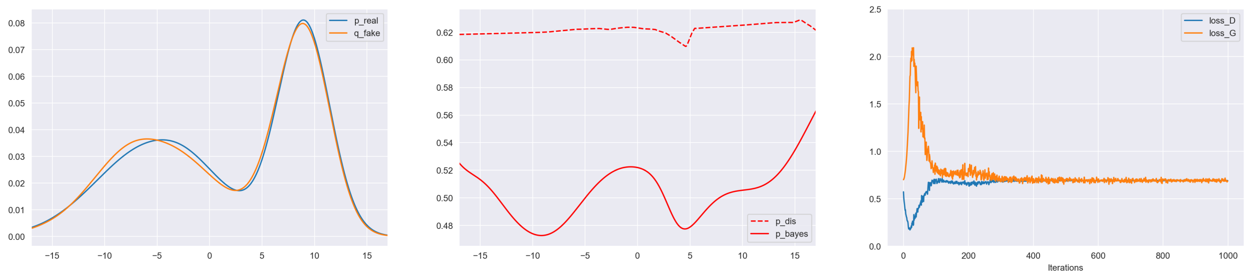

Figure 10: Adam optimizer for Gaussian distribution generation

Figure 11 illustrates the same plots but for vanilla SGD optimizer. Its obvious SGD needs much more time for the reconstruction of the distribution as the learning curve tried to converge.

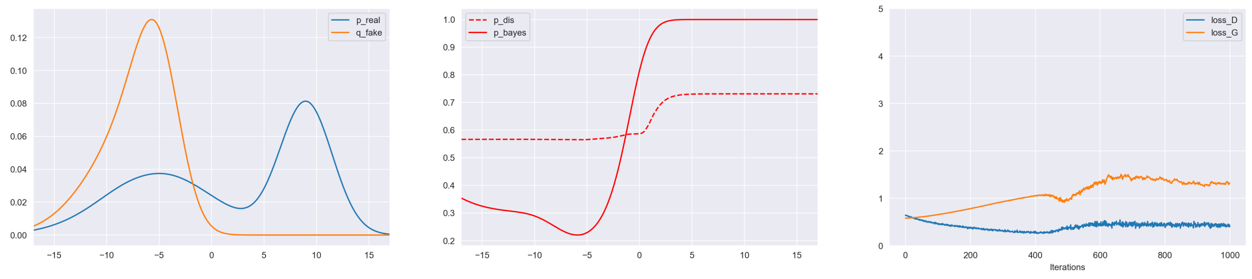

Figure 11: Vanilla SGD optimizer for Gaussian distribution generation

Figure 12 again illustrates the same kind of plots but this time for Quickprop algorithm. One can see from the first plot that reconstruction is still in the progress. And if one looks at the third plot it clear - circle of iterations has not been traversed. If we look the curves we can see that the curves are diverging at the end. During the experimentation we could make the algorithm to converge. Divergence point comes about at different iteration on every trial. This must mean that the divergence depends of the curvature of the loss function. We suspect that divergence issue originates for the oscillations that we mentioned in the previous section.

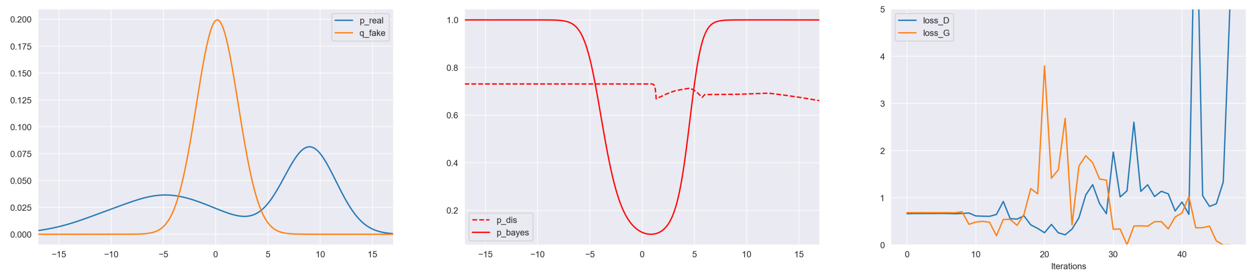

Figure 12: Improved QuickProp optimizer for Gaussian distribution generation

5 Conclusion

We observed that gradient sliding performs very well while training simple neural nets for regression. However, it has a limited area of application, and its performance depends heavily on how well algorithm's parameters are defined. Training GANs with GS lead to negative results, but the hope still lingers on.

Ellipsoid algorithm don't converge in all three scenarios: linear regression, MLP and simple GAN over Gaussian distributions.

Overall the Quickprop algorithm showed some promising when it was applied to a shallow MLP as it performed almost as good as SGD but Adam is superior. Not getting to speculations, it would a great research step to find the root of oscillations and see how the algorithm would behave if the problem with oscillations was solved

6 Team members' contribution

1. Abduragim Shtanchaev

   Initially worked on Ellipsoid method along with Alexey, but due to insufficient background in optimization theory took a simpler method - Quickprop. Studied and implemented Quickprop algorithm. Conducted a number of experiments on the algorithm with MLP and GANs. Wrote everything related to Quickprop algorithm and related work.

2. Aleksei Pronkin
   - Studying scientific literature
   - Generating ideas
   - Implementation of Ellipsoid method
   - Experiment with linear regression, MLP
   - Experiment with GANs
   - Report:
     - Methods
     - Implementations
     - Experiments
     - Future Work

3. Egor Gladin
   - Project topic proposal
   - Studying scientific literature
   - Implementation of GS algorithm
   - Derivation of Lipschitz constants for linear regression
   - Experiment with linear model
   - Report:
     - Abstract
     - Gradient Sliding (theory, implementation)
     - Experiments

4. Vyacheslav Rezyapkin
   - Studying scientific literature
   - Implementation of GS algorithm
   - Experiment with MLP
   - Experiment with GANs
   - Report:
     - Introduction
     - Experiments

References

[1] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[2] Guanghui Lan. Gradient sliding for composite optimization. *Mathematical Programming*, 159(1-2):201–235, 2016.

[3] Scott Fahlman. An empirical study of learning speed in back-propagation networks. 01 1999.

[4] Naum Z Shor. Cut-off method with space extension in convex programming problems. *Cybernetics*, 13(1):94–96, 1977.

[5] DB Yudin and Arkadii S Nemirovskii. Informational complexity and efficient methods for the solution of convex extremal problems. *Ekonomika i Matematicheskie Metody, Matekon Transl. Russian and East European Math*, 13(2):3–25, 1976.

[6] DB Judin and AS Nemirovskii. Evaluation of the informational complexity of mathematical programming problems. *Ekonomika i Matematicheskie Metody, Matekon Transl. Russian and East European Math*, 13(2):3–24, 1977.

[7] Arkadii S Nemirovskii and DB Yudin. Optimization methods adapting to problem of significant dimension. *Automation and Remote Control*, 38(4):513–524, 1977.

[8] Sébastien Bubeck. Convex optimization: Algorithms and complexity. *arXiv preprint arXiv:1405.4980*, 2014.

[9] Christodoulos A Floudas and Panos M Pardalos. *Encyclopedia of optimization*. Springer Science & Business Media, 2008.

[10] Lev M Bregman. The relaxation method of finding the common point of convex sets and its application to the solution of problems in convex programming. *USSR computational mathematics and mathematical physics*, 7(3):200–217, 1967.

[11] Guanghui Lan. An optimal method for stochastic composite optimization. *Mathematical Programming*, 133(1-2):365–397, 2012.

[12] James Martens and Roger B. Grosse. Optimizing neural networks with kronecker-factored approximate curvature. *ArXiv*, abs/1503.05671, 2015.

[13] Russell Eberhart and James Kennedy. Particle swarm optimization. In *Proceedings of the IEEE international conference on neural networks*, volume 4, pages 1942–1948. Citeseer, 1995.

[14] Martin Riedmiller and Heinrich Braun. A direct adaptive method for faster backpropagation learning: The rprop algorithm. In *IEEE international conference on neural networks*, pages 586–591. IEEE, 1993.

[15] Diego Granziol, Xingchen Wan, Timur Garipov, Dmitry Vetrov, and Stephen Roberts. Mlrg deep curvature. *arXiv preprint arXiv:1912.09656*, 2019.

[16] Hao Li, Zheng Xu, Gavin Taylor, Christoph Studer, and Tom Goldstein. Visualizing the loss landscape of neural nets. In *Neural Information Processing Systems*, 2018.