Numerical Analysis for PDEs
Advanced Programming for Scientific Computing

PDEs for Optimizing Deep Neural Networks

Stefano D'Alessandro 898939

Faculty of Mathematical Engineering - Computational Science and Engineering
Politecnico di Milano
February 2020

Contents

1 Introduction

The problem analyzed is the minimization of non-convex functions. This is a classic issue that in low dimensions has usually been tackled with methods such as multi-start Gradient Descent, while in high dimensions the standard algorithm is Stochastic Gradient Descent. The purpose of minimization is in this context mainly connected to the training of Neural Networks. Indeed the loss function of neural networks often shows a non-convex dependency to the parameters, hence making the training procedure quite complex. This problem is characterized by two specific aspects:

- the high dimention of the state, that can easily reach the hundreds of thousands
- the kind of function to be minimized, that is essentially the sum of a loss function over the training dataset of dimention D, made of items called ξ_i:

$$f(x) = \frac{1}{D} \sum_{i=1}^{D} g(\xi_i; x) \tag{1}$$

In this context the SGD algorithm is particularly suited as it speeds up the computation of the gradient, which could be a really time-consuming operation. The stochastic algorithm is studied for the optimization of non-convex profiles as it allowes the loss function to grow during the algorithm, in order to find a better "valley" over a "hill". Due to this behaviour, the step setting is often critical for this algorithm. A widely used rule is decreasing the step at each iteration dividing the initial length by the square root of the iteration number (sometimes by the iteration number itself for a faster convergence), other methods may be a step-wise reduction after a fixed number of iterations.

This works presents two new algorithms for the minimization of non-convex functions, and is based on the results of two articles [**entropy**] and [**osher**]. The former gives more detailed motivation to the introduction of a new algorithm for the training of neural networks. Indeed the loss functions that come from the training of neural networks are highly non-convex, so they often come with saddle-points or multiple local minima. Researchers in [**entropy**] state that local minima that lie in "wide valleys" are better in generalization (have a lower generalization error on sequestered data) than sharp isolated minima. Moreover, wide minima show less sensitivity to training data and to initial parameters changes. The new algorithms are then set up to expoloit this empirical observation, and bias the classical algorithms towards this kind of optimal points.

In the section two of this report the theoretical aspects of the algorithms are explained, moving from PDEs concepts. In the third section the algorithms are explained in detail, with a focus on the discretization used and the meaning of the parameters introduced. Wide space is given to the C++ implementation in section four, together with the link with neural network library OpenNN.

Results are shown in the last section, both for the low dimensions cases (with a comparison between the C++ and the Matlab implementation), and the Neural Network case with the application on a very classical dataset for the recognition of hand-written digits (MNIST).

2 Theoretical aspects

As already mentioned, the goal of the algorithms that will be introduced is to bias the classic algorithms towards "wide valleys" instead of "deep sinks". this can be related to the concept of Local Entropy.

Let $f(x)$ be an energy landscape, that will be our loss function in the Neural Network framework. Then the probability distribution

$$P(x;\beta) = Z_{\beta}^{-1} \exp(-\beta f(x)) \tag{2}$$

where Z_{β}^{-1} is a normalization constant, is called Gibbs distribution, and it concentrates at the global minimum of $f(x)$ as $\beta \to \infty$. We can modify this distribution to concentrate on different points by adding another term:

$$P(y;x,\beta,\gamma) = Z_{x,\beta,\gamma}^{-1} \exp\left(-\beta f(y) - \frac{\beta}{2\gamma} \parallel y - x \parallel^2\right). \tag{3}$$

The point where this distribution concentrates changes according to γ: if γ is small the shape is similar to a gaussian concentrated in x, while as γ grows the shape is more similar to the original Gibbs distribution. The parameter γ is used to focus the modified Gibbs distribution upon a local neighborhood of x and is also called scope parameter.

We can recognize the similarity of this function with the convolution product of the Gibbs distribution itself with the Heat equation fundamental solution. We then define the negative Local Entropy in this way:

$$f_{\gamma}(x) = u(x,\gamma) = -\frac{1}{\beta} \log \int_y \left(\frac{\beta}{2\pi\gamma}\right)^{\frac{d}{2}} \exp\left(-\frac{\beta}{2\gamma} \parallel x - y \parallel^2 - \beta f(y)\right) dy \tag{4}$$

It can be proved that the Local Entropy satisfies a Hamilton-Jacobi Viscous equation. The proof can be found in [**osher**], section three, and it is here reported.

Lemma 1: $u(x,\gamma)$ satisfies a Hamilton-Jacobi Viscous equation.

Proof: We can consider γ as a time variable and write:

$$u(x,t) = -\frac{1}{\beta} \log \nu(x,t) \tag{5}$$

so that $\nu(x,t)$ solves the Heat equation with initial condition

$$\nu(x,0) = \exp(-\beta f(x)) \tag{6}$$

that comes from (4) and the fact that the integral is the convolution of the Gibbs distribution with the Heat fundamental solution.

Inverting (5) we have:

$$\nu(x,t) = \exp(-\beta u(x,t)) \tag{7}$$

and we can then compute:

$$\begin{aligned} \nu_t &= -\beta\nu u_t \\ \nabla\nu &= -\beta\nu\nabla u \\ \Delta\nu &= -\beta\nu\Delta u + \beta^2 \parallel \nabla u \parallel^2 \nu \end{aligned} \tag{8}$$

Now, ν solves Heat equation as already said:

$$\nu_t = \frac{1}{\beta}\Delta\nu. \tag{9}$$

By substituting (8) in (9) we obtain

$$u_t = - \parallel \nabla u \parallel^2 + \frac{1}{\beta}\Delta u \tag{10}$$

that is a Hamilton-Jacobi Viscous equation. The initial condition is obtained by using (5) and the initial condition on ν:

$$u(x,0) = f(x). \tag{11}$$

□

Another important result, also stated in [**osher**], is about the gradient of the Local Entropy function.

Lemma 2: $\nabla u(x,t)$ can be computed in two equivalent ways:

$$\nabla u(x,t) = \int_y \frac{x-y}{t}\rho_1^\infty(dy;x) = \int_y \nabla f(x-y)\,\rho_2^\infty(dy;x) \tag{12}$$

where ρ_i^∞ for $i = 1,2$ are probability distributions given by:

$$\begin{aligned} \rho_1^\infty(dy;x) &= Z_1^{-1}\exp\left(-\beta f(y) - \frac{\beta}{2t}\parallel x-y\parallel^2\right) \\ \rho_2^\infty(dy;x) &= Z_2^{-1}\exp\left(-\beta f(x-y) - \frac{\beta}{2t}\parallel y\parallel^2\right). \end{aligned} \tag{13}$$

The statement can be proved simply by computing the gradient with respect to the x variable in the two versions of the convolution in the definition of the local entropy function. The gradient can be put inside the sign of integral by linearity of both operations and the fact that x is not the integration variable.

In particular the first form of the gradient in (12) will be used in the implementation of the algorithms.

The minimization process of a loss function $f(x)$ can also be seen as a continuous process described by this continuous-time equation:

$$dx(t) = -\nabla f(x(t))\, dt \tag{14}$$

Our effort in the definition and characterization of the Local Entropy for the loss function was aimed to modify the function to be minimized, in order to obtain a more regular behaviour. Indeed the Local Entropy $u(x;\gamma)$ is smoother and has wider minima thanks to the regularization properties of the Hamilton-Jacoby equation. The smoothing intensity is regulated by the scope parameter γ previously defined. We can then solve a different minimization problem, described by this other continuous equation:

$$dx(t) = -\nabla u(x(t);\gamma)\, dt = -\nabla f_{\gamma}(x(t))\, dt \tag{15}$$

Note that the two equations are related to different problems: (14) is linked to the minimization of the exact loss function $f(x)$ while (15) minimizes another function, obtained by regularizing the former. The discretization of the equation will be discussed in the next paragraph, when the algorithms will be introduced.

Another way of smoothing the loss function, would be to use a Heat equation instead of a non-linear equation such as the Hamilton-Jacobi. The two smoothings are actually different, and lead to different results. The second algorithm, that is discussed in [**osher**], is in fact based on the Heat equation.

Call the function regularized with the Heat equation in this way:

$$f_H(x;\gamma) = v(x,\gamma). \tag{16}$$

And consider γ as the time variable like in the previous case: $v(x,t)$ satisfies the Heat equation in this way:

$$v_t = \frac{1}{\beta}\Delta v \tag{17}$$

and the initial condition:

$$v(x,0) = f(x). \tag{18}$$

We then know the form of $v(x,t)$ is given by the convolution of the initial condition with the fundamental solution of the Heat equation:

$$v(x,t) = \int_y f(x-y)\left(\frac{\beta}{4\pi t}\right)^{\frac{d}{2}} \exp\left(-\frac{\beta \parallel y \parallel^2}{4t}\right) dy. \tag{19}$$

The gradient of this function, that will be used in the third algorithm, can be computed just by passing the gradient with respect to x inside the integral sign:

$$\nabla v(x,t) = \int_y \nabla_x f(x-y)\left(\frac{\beta}{4\pi t}\right)^{\frac{d}{2}} \exp\left(-\frac{\beta \parallel y \parallel^2}{4t}\right) dy. \tag{20}$$

And we can now define a third problem, described by this continuous-time equation, in which the function minimized is the function after the regularization performed by the Heat equation:

$$dx\left(t\right)=-\nabla v\left(x\left(t\right);\gamma\right)dt=-\nabla f_{H}\left(x\left(t\right);\gamma\right)dt \tag{21}$$

We can give a hint on how the smoothings and widening of convex regions are different for the two equations by looking at one particular solution. Suppose the initial condition to be

$$u\left(x,0\right)=v\left(x,0\right)=\frac{cx^{2}}{2} \tag{22}$$

Then the solution of the Hamilton-Jacobi equation would be

$$u\left(x,t\right)=\frac{cx^{2}}{2\left(ct+1\right)}+\frac{1}{2\beta}\log\left(\frac{ct+1}{c}\right). \tag{23}$$

Now imagine c to be big, of the order of 10^{8}, so the initial condition would be in fact convex but the minimum would be in a very deep valley. After only $0.01s$ the convexity constant of the solution would be reduced to nearly 50 and the valley would be a lot wider. So for this kind of equation we can see that the regularization and widening of the profile is very fast.

For the Heat equation instead the process may be a lot slower. Indeed consider the first eigenfunction of the Heat equation $\nu_{1}\left(x\right)$ corresponding to the eigenvalue λ_{1}. The first eigenvalue is usually of order 1. Then $\nu\left(x,t\right)=\exp\left(-\lambda_{1}t\right)\nu_{1}\left(x\right)$. The complete solution would be constructed with the series of the eigenfunctions, with their corresponding coefficients set according to the initial condition. The convexity constant would decrease with an exponential behaviour given by

$$C\left(t\right)=c_{0}\exp\left(-\lambda_{1}t\right)\approx c_{0}\left(1-\lambda_{1}t\right) \tag{24}$$

for small t, where c_{0} is the coefficient of the first eiegenfunction of the series. This behaviour is slower than the previous one.

3 Algorithms

The theoretical considerations done in the previous section, brought to the identification of two continuous-time equations, namely (15) and (21), that describe a minimization process. The discretization of these equation will be the starting point for the definition of the new algorithms that will be called **EntropySGD** and **Heat**. The former comes from the local entropy that has been introduced before, and is based on the Hamilton-Jacobi equation, while the latter makes use of the Heat equation for the regularization of the loss function.

3.1 EntropySGD

The algorithm starts from equation (15) that is here reported:

$$dx\left(t\right)=-\nabla u\left(x\left(t\right);\gamma\right)dt=-\nabla f_{\gamma}\left(x\left(t\right)\right)dt.$$

The variation of x will be discretized in a forward-Euler fashion, so it will just be written as the difference of two subsequent iterations $dx\left(t\right)\approx x_{k+1}-x_{k}$. The term dt will play the role of the length of the step used for the computation of the next value x_{k+1}. The gradient will be computed explicitly, using the previous value of the parameter x_k. The computation of the gradient, though, is quite complicated. We will make use of **Lemma 2** and in particular of the first equivalence in (12).

$$\nabla f_{\gamma}\left(x\right)=\int_{y}\frac{x-y}{\gamma}\rho_{1}^{\infty}\left(dy;x\right). \tag{25}$$

In particular we can see that the gradient is in this context given by the average of a function with respect to a certain distribution ρ_1^{∞}. We may write

$$\nabla f_{\gamma}\left(x\right)=\frac{x-\left\langle y\right\rangle}{\gamma} \tag{26}$$

where with $\langle y\rangle$ we indicate the average of y with respect to the distribution ρ_1^{∞} which is parametrized by x. This average will be computed recursively using a particular Markov chain Monte-Carlo technique (MCMC) called Stochastic Gradient Langevin Dynamics (SGLD). This technique is explained in [**entropy**], appendix A.

Consider the distribution over which the average has to be performed:

$$\rho_{1}^{\infty}\left(dy;x\right)=Z_{1}^{-1}\exp\left(-\beta f\left(y\right)-\frac{\beta}{2\gamma}\parallel x-y\parallel^{2}\right). \tag{27}$$

The Langevin Dynamics compute the average recursively, using as upgrade the gradient of the logarithm of the distribution, and injecting Gaussian noise. Starting from a value of $y_0=x_k$ and $\langle y\rangle_0=x_k$ both equal to the current iterate of x, the rules for the development of the average is given by:

$$\begin{aligned}y_{i+1}&=y_{i}+\eta_{y}\left(\nabla f\left(y_{i}\right)+\frac{y_{i}-x_{k}}{\gamma}\right)+\sqrt{\frac{\eta_{y}}{\beta}}\mathcal{N}\left(0,1\right)\\\left\langle y\right\rangle_{i+1}&=\left(1-\alpha\right)\left\langle y\right\rangle_{i}+\alpha y_{i+1}\end{aligned} \tag{28}$$

where η_y is the sizestep, β is a parameter that regulates the noise and should be proportional to the variance of the gradient of the loss function through the parameter space, α is a parameter that regulates the update of the average, usually is set near 1 so the average is more sensitive to the latest updates of y. This procedure is repeated for a fixed amount of times for each iteration of x: the number of "inner" iterations L is set differently in different cases. The

gradient will be computed stochastically, by picking a random mini-batch of size mb from the dataset and summing up the gradient terms relative to those dataset points.

Here we can summarize how one iteration of the EntropySGD algorithm works, given the parameters α,β,γ,ρ, the number of Langevin iterations L, the step sizes η_y and η_x and the mini-batch size mb:

Algorithm 1: EntropySGD iteration

Input: current weights x_k

`/* SGLD iterations */`

1 $y,\langle y\rangle = x_k$
2 **for** $l = 1 \leq L$ **do**
3 $\quad dy = \nabla_{mb} f(y) - \frac{x_k - y}{\gamma}$
4 $\quad y = y - \eta_y dy + + \sqrt{\frac{\eta_y}{\beta}} \mathcal{N}(0,1)$
5 $\quad \langle y\rangle = (1-\alpha)\langle y\rangle + \alpha y$
6 **end**

`/* Update weights */`

7 $x_{k+1} = x_k - \eta_x \left(\rho \nabla_{mb} f(x_k) + \frac{x_k - \langle y\rangle}{\gamma}\right)$

Note that with ∇f_{mb} we mean the stochastic gradient computed on a mini-batch of dimension mb. When solving problem (15), the parameter ρ should be set equal to zero. Actually in some cases ρ will be set to a small value, to make the algorithm converge more accurately to the local minimum of the initial loss function $f(x)$. β and γ should be tuned accordingly to balance the two terms in line 4 of the algorithm. Parameter γ won't be constant during the algorithm but will decrease as the algorithm proceeds. The law used for the evolution of γ is:

$$\gamma = \gamma_0 (1 - \gamma_1)^{\frac{k}{L}} \tag{29}$$

where k is the iteration number and L is the number of inner iterations. γ should decrease because the problem described by (15) approaches the real problem as γ is reduced, so the aim is to regularize the problem to help convergence but reduce regularization when the algorithm is close to convergence. The stepsize η_x is also not constant but will be stepwise reduced, by dividing it by 5 after 3 epochs each time (this is a thumb rule that has been taken by the papers analyzed).

3.2 Heat

The starting point for the second algorithm is actually very similar to the previous one. Starting from (21)

$$dx(t) = -\nabla v(x(t);\gamma)\, dt = -\nabla f_H(x(t);\gamma)\, dt$$

the discretization employed is the same, so the variation in x becomes the difference of two subsequent iterations, and the term dt becomes the stepsize. The critical point is once again the computation of the gradient of the regularized function. Also in this case the function $f_H(x)$ and its gradient with respect to x can be expressed thorugh a convolution, as already showed in equations (19) and (20). In particular the second equation here reported can help in understanding how to treat the gradient term:

$$\nabla v(x,t) = \int_y \nabla_x f(x-y) \left(\frac{\beta}{4\pi t}\right)^{\frac{d}{2}} \exp\left(-\frac{\beta \parallel y \parallel^2}{4t}\right) dy.$$

As in the previous case, the gradient term will be computed explicitly using the current parameters value x_k. The convolution will be seen as the average of a certain function with respect to a distribution that in this case is simply a gaussian distribution with zero mean and standard deviation depending on t or on γ (we can say that the standard deviation of the distribution will determine the extent of the regularization performed on the loss function). So in this second algorithm the gradient term will be approximated by an average performed in this way:

$$\nabla f_H(x) \approx \frac{1}{L} \sum_{l=i}^{L} \nabla_{mb} f\left(x - \sqrt{\gamma}\mathcal{N}(0,1)\right) \tag{30}$$

Where the parameter L has the same function of the number of inner Langevin iterations of the previous algorithm. Note that the gradient has been substituted with the stochastic gradient computed on a random mini-batch of size mb. The parameter γ, as already said, changes the grade of smoothing performed by the algorithm, by increasing or decreasing the standard deviation of the gaussian perturbation of the parameters. One iteration of the Heat algorithm can be simplified by this pseudocode:

Algorithm 2: Heat iteration

Input: current weights x_k

`/* Computation of the average for the gradient */`

$g = 0$

for $l \leq L$ **do**

  $g = g + \nabla_{mb} f\left(x_k - \sqrt{\gamma}\mathcal{N}(0,1)\right)$

end

$dx = \frac{1}{L} g$

`/* Update weights */`

$x_{k+1} = x_k - \eta_x dx$

The parameter γ won't be constant through the iterations but will be set with the same rule as the EntropySGD algorithm, so with equation (29), with different values of γ_0 and γ_1. The stepsize for x will also be reduced during the

algorithm. The rule that has been chosen for the Heat algorithm is to divide an initial stepsize η_0 by the square root of the epochs number. So like this:

$$\eta_x = \frac{\eta_0}{\sqrt{\frac{kLmb}{D}}} \tag{31}$$

where D is the dimention of the dataset.
Both algorithms will be initialized with a random starting point x_0. Both algorithms are provided with the same termination conditions: maximum number of epochs (an epoch is counted as a visit on the complete dataset) and minimum upgrade of the value of the loss function. The number of epochs is usually set at 200 and the tolerance on the change of the function is set at 10^{-4}.

4 C++ implementation

The algorithms have been implemented both in Matlab and in C++. The Matlab implementation is simple and won't be discussed in this report, as it treats only the low-dimentional cases. The C++ implementation, instead, need a little more attention, and it would cover also the application to Neural Networks.

The problem has been divided in two main issues: the characterization of the functions to be minimized and the algorithms that carry on the minimization procedure. For both of this aspects the approach has been similar. An abstract class has been defined, in a logic of Promise-no-Less Require-no-More. From the abstract class, containing pure virtual methods, the concrete classes has been publicly derived, and the virtual methods have been overridden.

4.1 OptimizationFunction

The abstract class for the concept of function to be minimized has been called OptimizationFunction. It represents the concept of a function that has a particular form, as stated in the introduction in equation (1). The structure of the dataset will depend on the specific implementation.

The class contains a state dimention, that will be used by the algorithm for the dimention of the state vectors and gradient vectors. The class provides contructors, getters and setters and a set of pure virtual methods that need to be implemented by the derived classes:

- **evaluate** is the method that returns the value of the function, given a state, that represents the point at which the function has to be evaluated. This will be used by the algorithm, but can also be used after the algorithm has been performed to compute the residual error, or to run the trained neural network on a different dataset.
- **stochastic_gradient** is the method that returns the stochastic gradient given the state x where it has to be computed and the size of the mini-batch. This will be used by the algorithms.

- **get_data_dim** will be implemented by the derived classes, after the dataset structure will be specified.

A choice that has been made a-priori is to work with the Eigen/Dense library for the handling of the vectors of parameters, so the functions will need an Eigen::VectorXd for the evaluation and will return an Eigen::VectorXd as stochastic gradient. This choice was made to simplify the sum and product operations needed by the algorithms. The state dimension stored in the class is useful also for control reasons: as the dimention of the passed state is dynamic it will be checked before any computation is made. The linked version of the Eigen library is eigen/3.3.3

Two classes are then derived from OptimizationFunction: one for the 1D and 2D cases and one for the Neural Networks cases.

FunctionData1D

The function for the low dimensions cases is called FunctionData1D and is based on a one-dimensional dataset, implemented as a shared pointer to a std::vector⟨double⟩. The choice of the shared pointer has been made because there is the possibility that different function would need to work on the same dataset, so in a logic of memory saving. The dataset can also be built up by the function itself with the method **make_dataset** by setting the dimension of the dataset, the minimum and maximum value. It would be created by randomly picking the elements from a uniform distribution. In addition to the dataset, this class contains two std::function members that allow the class to bind with any callable object. The two functional members are in fact the function g and the gradient of g with respect to the parameters, where g has the meaning of g in (1):

$$f(x) = \frac{1}{N} \sum_{i=1}^{N} g(\xi_i; x).$$

The function will be evaluated by summing up $g(\xi_i; x)$ over the dataset, and the stochastic gradient will return the sum of $\nabla_x g(\xi_i; x)$ computed on the selected mini-batch. This class also provides the full gradient method, that is computed just like the stochastic gradient but summing on all the dataset.

FunctionOnNeuralNetwork

The function for the implementation of NeuralNetworks is called FunctionOnNeuralNetwork and is based on the open source library OpenNN. The library contains a lot of classes, tools and algorithms for neural networks, much more than needed for the purpose of this project. There is though a main difference regarding the implementation of the set of parameters of the neurl network. The decision in the library has been to derive publicly a new class from std::vector⟨double⟩, instead of using Eigen::VectorXd. The interaction between the classes alredy defined and the classes from the library has some difficulties.

FunctionOnNeuralNetwork contains a pointer to a OpenNN::LossIndex which is an abstract class of the library, that represents the loss function used to determine the loss, or the error, of a neural network. From OpenNN::LossIndex are derived for example CrossEntropyError or MeanSquareError. Every LossIndex instance has a pointer to a Dataset instance and a NeuralNetwork instance.

The constructor of FunctionOnNeuralNetwork gets directly the pointer to an already built LossIndex. Then with the set methods one can modify the NeuralNetwork or the Dataset. The NeuralNetwork has to be built outside the FunctionOnNeuralNetwork class and then set by calling the setter, or can be done by calling the **get_neural_network_ pointer** method and modifying the neural network already linked.

The **evaluate** and **stochastic_gradient** methods have been overridden according to the methods of OpenNN::LossIndex.

4.2 MinimizationAlgorithm

The abstract class that makes a base from witch the various algorithms are derived is called MinimizationAlgorithm. The main member of this class is the pointer to an OptimizationFunction. The fact that the pointer is to a generic OptimizationFunction allowes the link with both Function1D and FunctionOnNeuralNetwork. The pure virtual method **solve** needs to be overridden by every derived class, that will stand for a specific algorithm. Another virtual method is the getter of the epochs performed during the run of the algorithm. The other members are the starting and minimum points, that are implemented as Eigen::VectodXd as stated before, the computation time, the number of iterations performed and the sequence of function values if one wants to see or display how it changed during the algorithm.

Each derived algorithm class should override the **solve** and **get_epochs** methods and should contain the specific parameters for the algorithm (learning rates, tolerances for the stopping criteria and various parameters).

StochGradDesc

The first derived algorithm is the class that performs the stochastic gradient descent. This algorithm is implemented to make a comparison with the new algorithms introduced. The parameters are:

- **tolerance_t**, **tolerance_t** and **max_iterations** that are used in the stopping criteria that takes into account the number of iterations, the change in the optimized function and the length of the parameter update.
- **t_zero** is the initial sizestep, that is reduced by **beta** at each iteration.
- **mb_size** is the dimention of the mini-batch for the copmutation of the stochastic gradient.

The parameters can be set by the constructor or by a specific method.

EntropySGD

This is the class that implements the first algorithm described in the previous section. The parameters are:

- **tolerance_f** and **max_epochs** that are used in the stopping criteria.
- **mb_size** is the dimention of the mini-batch for the copmutation of the stochastic gradient.
- **L_iterations** is the number of inner Langevin Dynamics iterations.
- **y_step** and **x_step** are the learning rates that in the algorithm were indicated by η_y and η_x.
- **gamma_0** and **gamma_1** are two parameters that are used to set the value of γ at each iteration.
- **alpha**, **beta** and **rho** are respectively the parameters for the average update, the parameter for the normalization of the noise and the coefficient of the gradient in the update of x_{k+1}.

The parameters can be set by the constructor or by a specific method.

Heat

This is the class that implements the second algorithm described in the previous section, the algorithm that is based on the Heat equation regularization. The parameters are:

- **tolerance_f** and **max_epochs** that are used in the stopping criteria.
- **mb_size** is the dimention of the mini-batch for the copmutation of the stochastic gradient.
- **L_iterations** is the number of iterations to be performed for the computation of the gradient term.
- **x_step** is the learning rates that in the algorithm was indicated by and η_x.
- **gamma_0** and **gamma_1** are two parameters that are used to set the value of γ at each iteration, that determines the standard deviation of the gaussian noise for the average computation.

The parameters can be set by the constructor or by a specific method.

4.3 GradientDescent

The algorithm class that implements the gradient descent method is not derived from MinimizationAlgorithm because it doesn't require a generic OptimizationFunction (that provides only the stochastic gradient method), but a function that provides also the full gradient method. So the main member of this class is a pointer to a FunctionData1D.

The structure is really similar to the structure of the other algorithms: with parameters, solve method, getter for the number of epochs, starting and minimum points implemented as Eigen::VectorXd.

The parameters are:

- **tolerance_f** and **tolerance_t** that are used in the stopping criteria, that takes into account the change in loss function but also the change in the parameters.
- **t_zero** is the initial learning rate.
- **alpha** and **beta** that are the parameters for the backtraking line search.

The Doxygen documentation is also made available to clarify the relationships between the classes. To summarize we can say that the link between the optimization functions and the algorithms is given by the pointer to a function stored in the algorithm class. Through this pointer the algorithm can access all the needed methods that are basically the evaluation and the stochastic gradient. Direct access to the dataset is not needed by the algorithm, as it is only used for the aforementioned operations.

4.4 Compilation and Run

The code is provided divided in four folders: Headers, Sources, Data and Libraries. In the Headers folder one can find all the headers of the defined classes and also the headers needed by the OpenNN library. The static library libopennn.a can be found in the Libraries folder. The Source folder contains all the source code, with also three different main functions and the makefile. The Data folder, eventually, contains the dataset used for the example run in the Neural Network case, saved in csv format. The two files differ in size: the number of instances is indicated in the name of the files. The whole material needed to use the OpenNN library can be found on Git at this link https://github.com/Artelnics/OpenNN.git, but the provided material is sufficient for the run of the examples. The three provided main function can be used to run test that are divided by dimension:

- **main_1D** can be used to run a test on the 1D function defined before. The function g and its gradient are declared and defined in the same file. Their prototypes are the following:

```
double f_1D(Eigen::VectorXd, double);
Eigen::VectorXd df_1D(Eigen::VectorXd, double);
```

The dataset and the FunctionData1D are initialized inside the main function. After that, all the four algorithms are performed and the results are shown in terms of convergence to the global minimum, convergence to the wider minima and computation time.

- **main_2D** can be used to run a test on the 2D function defined before. The structure is the same that **main_1D**. A difference is that the dataset is constructed by the function itself by using the method **mke_dataset** while in the previous script the dataset was built outside and passed to the function with its pointer. Another difference is that in the results only the convergence rate to the global minimum is displayed.

- **main_NN** can be used to run a test on a simple Neural Network. The dataset is loaded from the Data folder, the Neural Network is built using the classes provided by the OpenNN library, by defining the layers and adding them to the network with their pointer. Then the LossIndex is defined and the FunctionOnNeuralNetwork is built. In the script can be found the operations to run the three possible algorithms (Gradient Descent is not available for this class of function).

A makefile is made available for the compilation of the files. The makefile contains three different targets, one for each main script. Note that to compile properly, the first two scripts need to be linked to Eigen library while the last one also needs the OpenNN library link.

5 Tests

The algorithms have been implemented and tested both with Matlab and C++. The Matlab implementation has been used only for the 1D and 2D cases while the C++ code has been developed also for a Neural Network experiment for the recognition of hand-written digits of the well-known dataset MNIST. The two implementations have been compared when possible.

5.1 1D Function

The first step has been the definition of a function f of a scalar variable x of the form defined in the introduction by equation (1). The dataset Ξ is made of $D = 1000$ items ξ_i with $i = 1, ..., 1000$, divided in two subsets: 500 numbers picked randomly from a uniform distribution between 0 and 50 and 500 numbers again picked from a random uniform distribution between 70 and 120. The base function g is structured as the quadratic loss between a target function $h(\xi)$

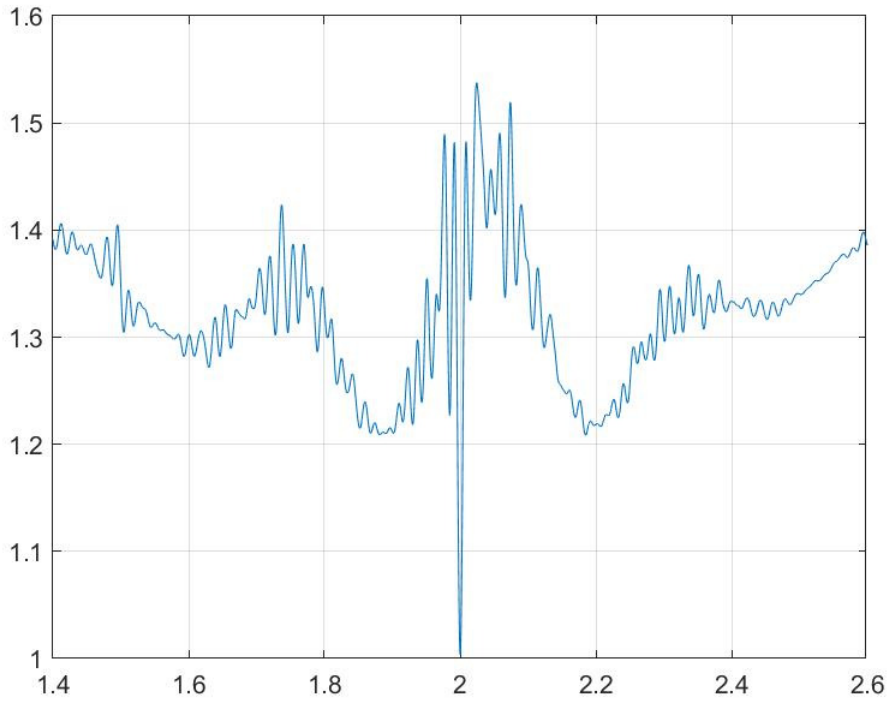

Figure 1: 1D function to be minimized

and a function $w(\xi; x)$ parametrized by x:

$$f(x) = \frac{1}{D} \sum_{i=1}^{D} \frac{1}{2} \left(h(\xi_i) - w(\xi_i; x) \right)^2 . \tag{32}$$

To obtain a non-convex and oscillating behaviour the function w has been constructed as a non-linear combination of sin and cos functions, where the dependency from x was like a frequency argument in the sinusoidal functions. h instead has been set in this way:

$$h(\xi) = w(\xi, 2) + \frac{1}{50}|\xi|. \tag{33}$$

The resulting f is strongly non-convex, with a very narrow and deep global minimum around $x = 2$ and two wider but less deep local minima in the surroundings, as can be seen in figure 1.

The total gradient will be then computed as the sum of the gradient of g

with respect to x:

$$\nabla f(x) = \frac{1}{D} \sum_{i=1}^{D} - \left(h(\xi_i) - w(\xi_i; x) \right) \nabla w(\xi_i; x) . \tag{34}$$

while the stochastic gradient will be computed by summing only some of the items ξ_i picked up to form the mini-batch.

For this test the minimization has been performed with four different algorithms:

1. **Gradient Descent**: tolerance on f decrease set at 10^{-6}, tolerance on the change of x set at 10^{-6}. Parameters for the backtracking: $\alpha = 0.3$, $\beta = 0.8$. Initial learning rate $t_0 = 0.4$.

2. **Stochastic Gradient Descent**: tolerance on f decrease set at 10^{-5}, tolerance on the change of x set at 10^{-6}, maximum number of iterations 500. Initial learning rate $t_0 = 0.15$, coefficient for the reduction of the sizestep $\beta = 0.8$.

3. **EntropySGD**: tolerance on f change set at 10^{-4}, maximum number of epochs 200. Learning rate $\eta_y = 0.01$, initial learning rate $\eta_x = 0.005$. Coefficient for the noise normalization $\beta = 800$, coefficients for the regularization $\gamma_0 = 0.2$, $\gamma_1 = 0.01$. Number of Langevin iterations $L = 15$, $\alpha = 0.8$, $\rho = 0.9$.

4. **Heat**: tolerance on f change set at 10^{-4}, maximum number of epochs 200. Initial learning rate $\eta_x = 0.01$. Coefficients for the regularization $\gamma_0 = 0.0001$, $\gamma_1 = 0.00001$. Number of iterations for the gradient average computation $L = 20$.

Every algorithm has been started with a random starting point, picked from a uniform distribution between 1.4 and 2.6. The results are shown for $N = 1000$ trials for each algorithm, and the histogram below shows the percentage of convergence on the domain. The computation time is also reported in the caption.

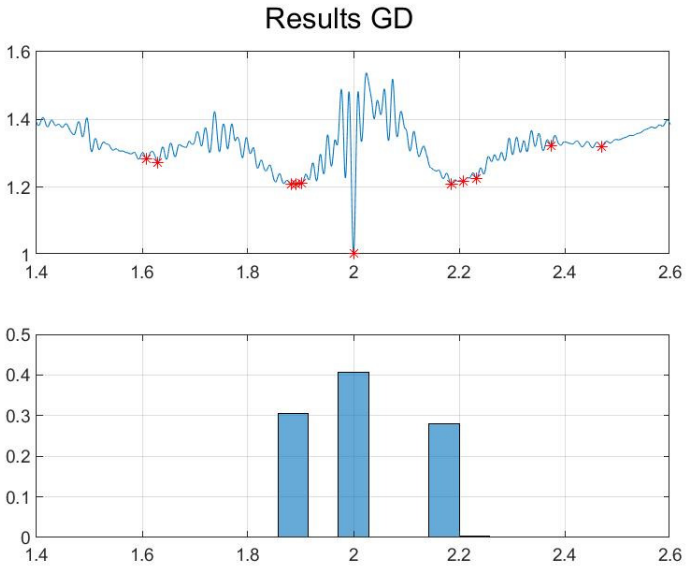

Figure 2: GD: Mean computation time 0.0661s

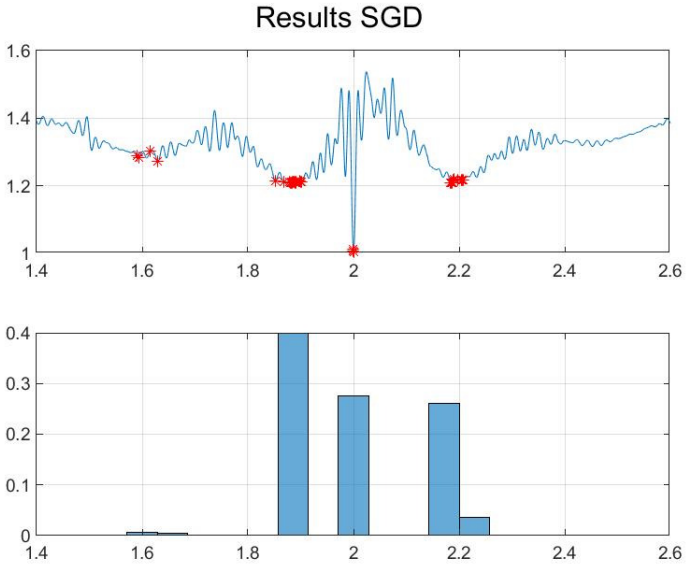

Figure 3: SGD: Mean computation time 0.0219s

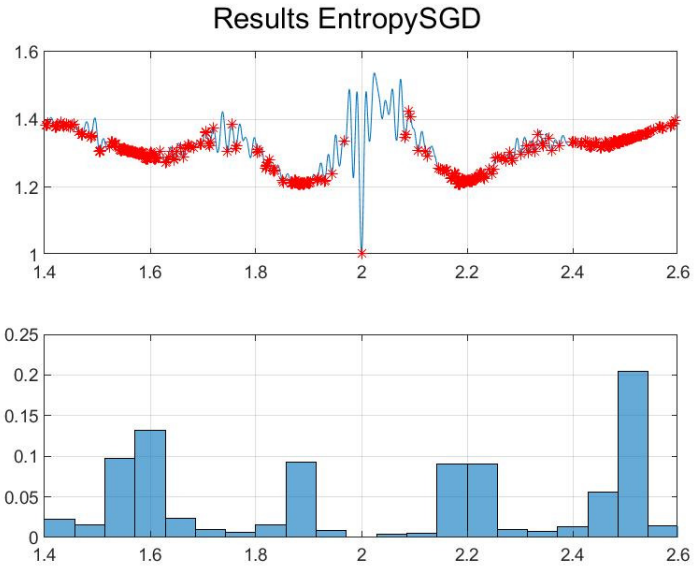

Figure 4: EntropySGD: Mean computation time 0.0171s

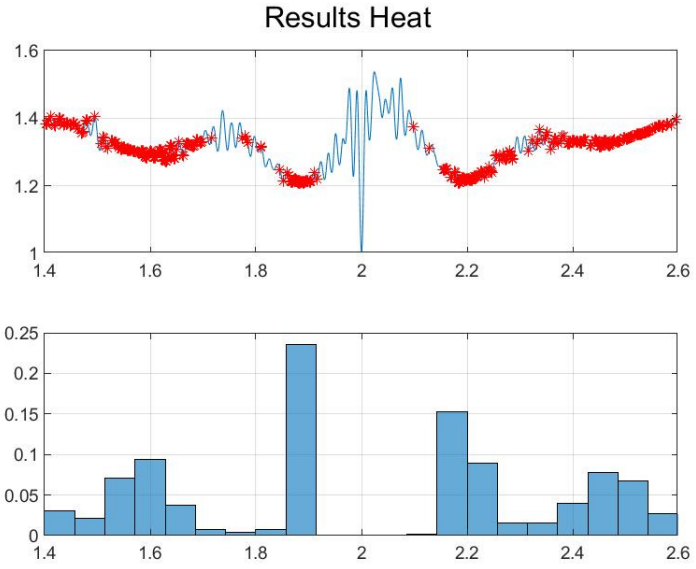

Figure 5: Heat: Mean computation time 0.0208s

Most of the effort in running these test has been focused on the search of the values for the parameters. In particular for the EntropySGD and the Heat algorithms.

It's clear that the two new algorithms EntropySGD and Heat fail in finding the global minimum of the function. But it is also clear how for both algorithms the minimum points found are more concentrated in the wide valleys in the surroundings of the deep sink. Due to the low dimentionality of the problem, we can see that the difference in time taken by the Gradient Descent is not huge, but we can already appreciate that the other three algorithms are faster.

The same results are found by the C++ implementation. With the following results obtained by running $N = 1000$ trials starting from random point picked from the same distribution and counting the points for which the algorithm has converged to the global minimum or to the other two wide minima, the computation time is also reported and we can see that is lower than the matlab computation time:

1. **Gradient Descent**
   - percentage of global minimum = 0.66
   - percentage of wider minima = 0.34
   - computation time = 0.0206s
2. **Stochastic Gradient Descent**
   - percentage of global minimum = 0
   - percentage of wider minima = 0.58
   - computation time = 0.0152s
3. **EntropySGD**
   - percentage of global minimum = 0
   - percentage of wider minima = 0.28
   - computation time = 0.0061s
4. **Heat**
   - percentage of global minimum = 0
   - percentage of wider minima = 0.36
   - computation time = 0.0119s

5.2 2D function

A similar test has been set up for 2D function. The dataset ha the same dimension $D = 1000$, and the values are uniformly picked between 0 and 20. The

structure of the function is the same of (32), with the only difference that now x has dimension 2, so we can write:

$$f(x_1, x_2) = \frac{1}{D} \sum_{i=1}^{D} \frac{1}{2} \left(h(\xi_i) - w(\xi_i; x_1, x_2) \right)^2 . \tag{35}$$

As in the 1D case, to obtain a non-convex result, w has been set by a non-linear combination of sin and cos, while $h(\xi) = w(\xi; 2, 2)$. So $x = (2, 2)$ is the global minimum.

The gradient is computed in the same way expressed by equation (34).

The resulting shape of the function is shown in figure 6. The function is not oscillating as much as the 1D case, but it still has many local minima, even if in this case the global minimum in $(2, 2)$ is quite wide so we would expect also the new algorithms to concentrate in that point.

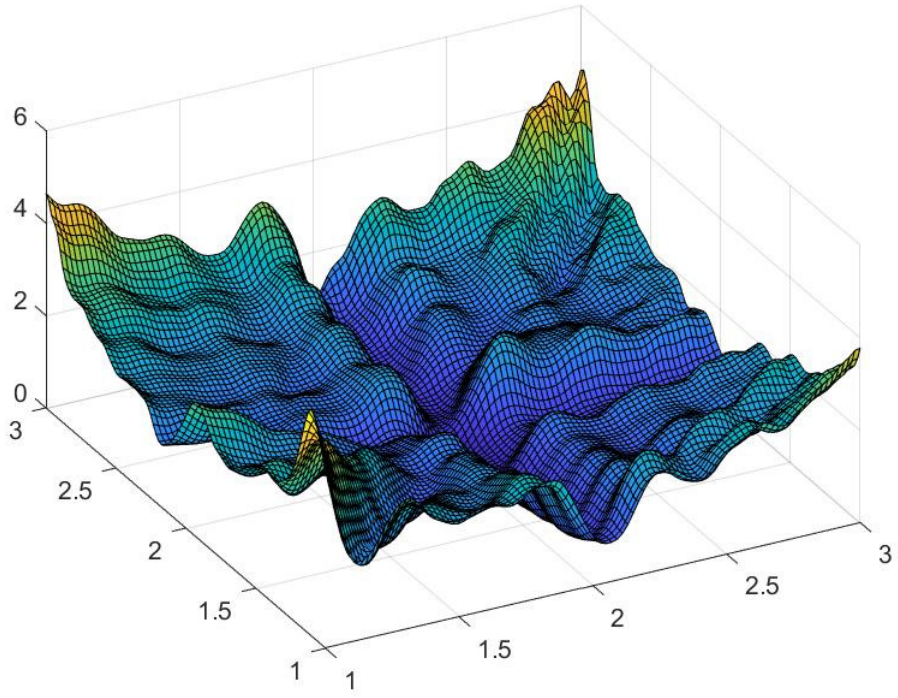

Figure 6: 2D function to be minimized

The used algorithms are exactly the same: Gradient Descent, Stochastic Gradient Descent, EntropySGD and Heat. The first two algorithms are also performed with the same values for the parameters while the othe two have different parameters for the 2D experiments:

3. **EntropySGD**: tolerance on f change set at 10^{-4}, maximum number of epochs 200. Learning rate $\eta_y = 0.04$, initial learning rate $\eta_x = 0.1$. Coefficient for the noise normalization $\beta = 100$, coefficients for the regularization $\gamma_0 = 1$, $\gamma_1 = 0.1$. Number of Langevin iterations $L = 10$, $\alpha = 0.9$, $\rho = 0.01$.

4. **Heat**: tolerance on f change set at 10^{-4}, maximum number of epochs 200. Initial learning rate $\eta_x = 0.08$. Coefficients for the regularization $\gamma_0 = 0.05$, $\gamma_1 = 0.001$. Number of iterations for the gradient term $L = 20$.

As a thumb rule in the parameters for the Heat algorithm we can say that γ_0 should be related to the wildedness of the function: if the function is extremely non-convex and heavily oscillating as in the 1D case, γ_0 should be smaller. Indeed since the 2D function is not as wild γ_0 has been set with a bigger value. γ_1 should always be set with a value at least one order of magnitude smaller than γ_0.

The results for the 2D case are shown in the following figures, where the function has been represented by the colours of the surface instead that with the 3D plot to make it more clear.

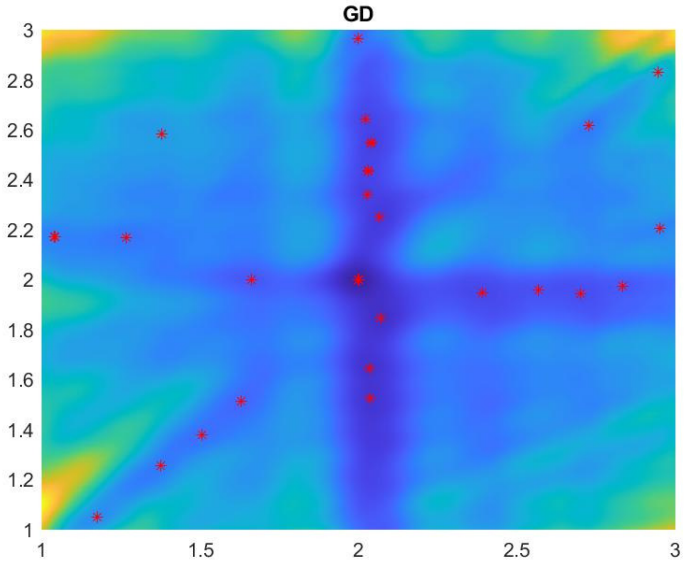

Figure 7: GD: Mean computation time 0.3447s

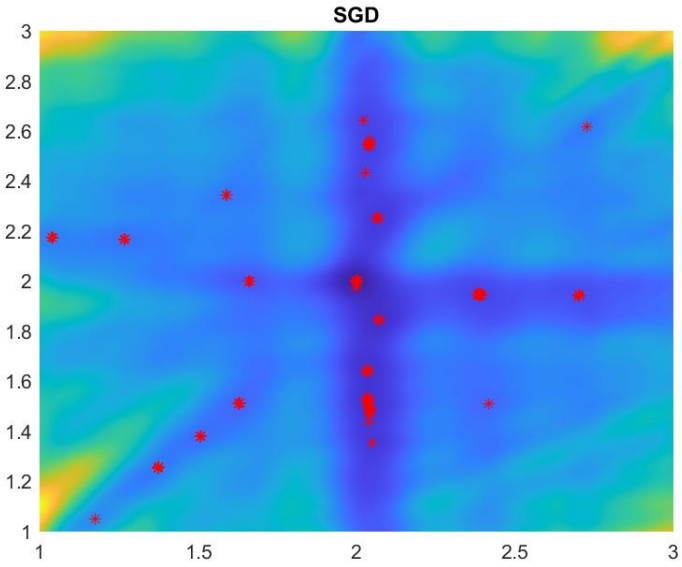

Figure 8: SGD: Mean computation time 0.0415s

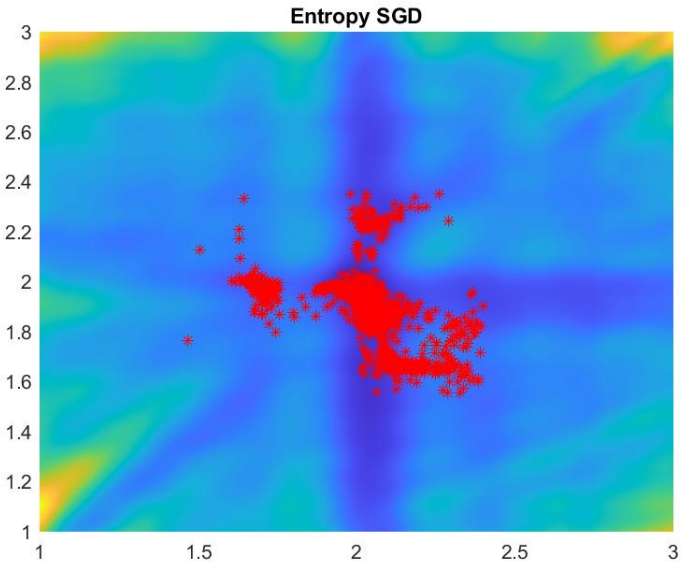

Figure 9: EntropySGD: Mean computation time 0.1125s

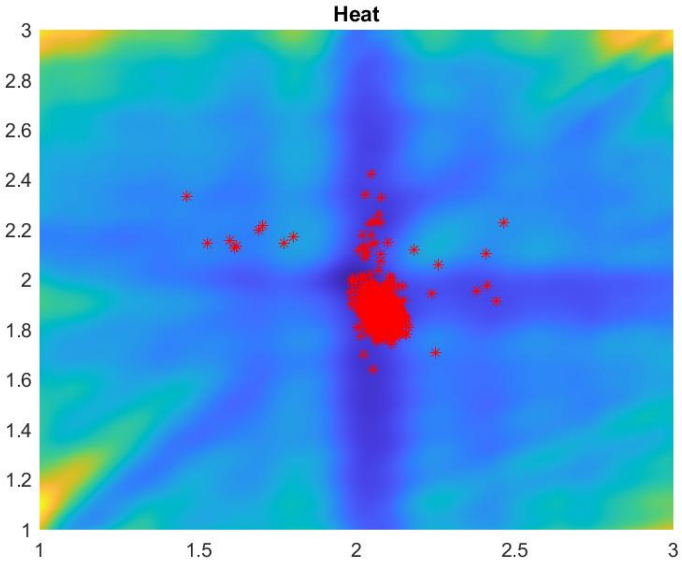

Figure 10: Heat: Mean computation time 0.1657s

The results show how the first two algorithms are a lot more precise in finiding a minimum, but often reach the wrong minimum, so the points found are spread in the space, maybe even far from the global minimum. The two new algorithms instead fail in reaching a minimum but are more concentrated in the global minimum surroundings. As regards the computation times, the stochastic algorithms win in general with the full Gradient Descent, but the new algorithms loose with the classical Stochastic Gradient Descent. We have also in this case to consider the low-dimentionality of the problem, as the new algorithms have been studied to cope with high dimensions.

The C++ results are analogous, here are reported the convergence to the global minimum rates and the computation times for both the implementations:

1. **Gradient Descent**
   - percentage of global minimum: matlab = 0.131, C++ = 0.074
   - computation time: matlab = 0.3447s, C++ = 0.1606s
2. **Stochastic Gradient Descent**
   - percentage of global minimum: matlab = 0.198, C++ = 0.190
   - computation time: matlab = 0.0415s, C++ = 0.0160s
3. **EntropySGD**
   - percentage of global minimum: matlab = 0.080, C++ = 0
   - computation time: matlab = 0.1125s, C++ = 0.0088s
4. **Heat**
   - percentage of global minimum: matlab = 0.001, C++ = 0.001
   - computation time: matlab = 0.1657s, C++ = 0.1346s

5.3 Neural Network

The last trials have been made on a Neural Network for the identification of hand-written digits. This task is very classical and is widely found in literature. The dataset is called MNIST and was originally made of 60000 images in grey-shades of dimention 28x28 pixels. Each image represented a digit from zero to nine. The dataset has been rescaled for this project. The data used are contained in the Data folder, saved in csv format.

The Neural Network has been set up accordingly to an example that was provided with the OpenNN library material, and is composed by several layers:

1. **scaling layer** that performs a Min-Max scaling putting all the values of the pizels between 0 and 1, while in the initial data the values are between 0 and 225.

2. three **convolutional layers** each one followed by a **pooling layer**. The convolutional layers are made of a number of filters of a certain dimension: the first has 8 filters of dimension 5x5, the second 4 3x3 filters and the last one 2 filters of again dimension 3x3. The pooling layer perform the average-pooling operation that consists in down-sampling the image by computing the average over a patch of size 2x2.

3. **perceptron layer** made of 18 perceprtons.

4. **probabilistic layer** to assign a single final digit to the result of the previous computations.

The number of parameters of this architecture is 11186.

Both the Dataset and the NeuralNetwork are used to define the LossIndex. The chosen rule for the computation of the LossIndex has been the MeanSquaredError.

The algorithms that have been tried are the three that make use of the stochastic gradient, so SGD, EntropySGD and Heat. The method can be chosen at run time following the instructions displayed on the stream.

The results for this experiment are unfortunately poor, due to the lack of sensitivity in setting the parameters and the long computation times needed. The best results obtained until now will be reported. More simulations will be performed before the discussion date, and will be shown during the presentation.

1. with **Stochastic Gradient Descent** the best result was the 9.17% of accuracy, that is even below the percentage of randomly guessing the digit. It was obtained by setting 250 maximum iterations, a mini-batch size of 100. $\beta = 0.6$, learning rate $\eta_x = 10$. It took 3511.21 seconds for the computation (almost one hour).

2. with **EntropySGD** the best accuracy has been 14.17%, just above the percentage of randomly guessing the digit. It was obtained performing only 1 epoch of iterations, with a mini-batch size of 60, 10 Langevin iterations, $\beta = 100$, $\gamma_0 = 1$ and $\gamma_1 = 0.01$. $\alpha = 0.9$, $\rho = 1$. Learning rates $\eta_y = 2$ and $eta_x = 8$. It took 349s, almost 6 minutes, and it was the shortest trial made, also because the epochs were very limited.

3. for **Heat** various trials have been performed with different values for γ_0 and for η_x but the algorithm always ended recognizing all images as zeros, and with an infinite value for the loss function.

References

[1] P.Chaudari et al. "Entropy-SGD: biasing Gradient Descent into wide valleys". In: *arXiv:1611.01838v3* (2016).

[2] P.Chaudari et al. "Deep relaxation: partial differential equations for optimizing deep neural networks". In: *arXiv:1704.04932v2* (2017).