Contents

Chapter 1

Introduction

In this report we analyze a variety of gradient-based optimization algorithms, with a focus on the class of accelerated gradient methods (AGM). This class of methods, building on the work of Polyak[1] and Nesterov, achieves better convergence speed than basic gradient descent while not requiring heavier assumptions. Although the convergence of AGM is definitely not in the same league as second order methods[2], as it is at best linear, they have shown great success in practice and particularly in machine learning. In fact we will see how their implementation in the training of neural networks compares to the classical gradient approach. However it must be kept in mind that optimization algorithms applied to machine learning differ from pure optimization as in the first case the function, called cost function, we are optimizing is not the true goal itself but only a, hopefully good, approximation of the real performance on the task at hand. Therefore reaching the global optimum of said function might not even be desirable as it often leads to poor generalization -the heart of machine learning- capabilities. In fact it has been shown[3], that besides becoming increasingly difficult as the size of the neural network increases, finding the global optimum of the cost function over the set of given examples becomes irrelevant as its value becomes increasingly decorrelated from the value of unlabeled examples.

Nevertheless, improving gradient descent is still crucial whether we want to global optimum or not, as large problems can quickly become intractable. The main intuition behind the accelerated methods that we discuss here is that of using information from past direction to speed up the descent on down-hill surfaces while improving the stability on fluctuating ones. Imagine starting from the long horizontal end of an ellipsoid surface: while gradient descent may repeatedly bounce off the vertical edges whilst moving towards the center, we would like to reduce this tendency and speed up the horizontal pace. An illustration of this can be seen in figure 1.1.

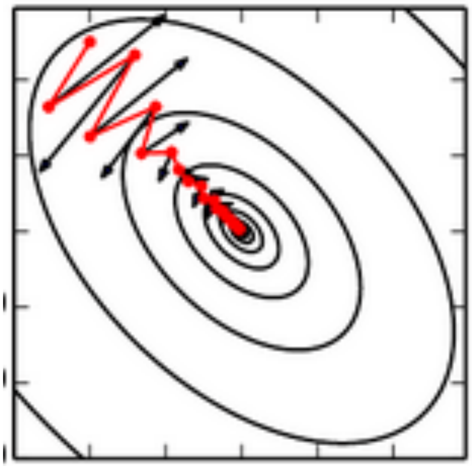


Figure 1.1: The black path is the path that gradient descent would take. Red path is obtained by using past information with the momentum technique.[4]

1.1 Objective

Our objective in this report is that of applying some advanced optimization methods for the training of neural networks in order to solve a machine learning task. Some premises are necessary before proceeding further: optimization theory and machine learning are connected but their goals often do not match. In pure optimization the goal is that of finding the optimal solution of some given function, be it convex, non-convex, constrained or unconstrained. While it is true that in machine learning we want to find the optimal solution of a function as well, in this case the function may not be an accurate representation of what we would really like to optimize. By the time optimization theory comes into play, our function might already be affected by some errors thus compromising the end goal: first off we need to decide a representation for it and this might already be a source of errors as a proper representation is often dependant on the task at hand - and we do not have the means to characterize it properly. Moreover the available data to build the model on is often limited, possibly noisy thus adding a bias into the resulting solution.

The intricate interplay between optimization and machine learning [5] requires careful consideration as treating a machine learning problem as a plain optimization one would most likely lead to poor results in the resulting model.

For these reason our goal for this project cannot be a fully joint one - in this report we mostly put our attention on the optimization part but also consider the results obtained from a machine learning point of view albeit with some extra care.

1.2 The optimization problem

We can define an optimization problem by means of a set of variables (with or without constraints) and a objective function[6]. The goal is then defined as finding the values for the variables that lead to the optimal value of the objective function.

In our case the objective function is the "Mean Euclidean Error". This function expresses the error of our neural network over a given set of labeled samples $< x, t >$. More precisely it is average of the euclidean distances over the set of samples between the predicted output of the neural network and the given correct(possibly noisy) output t. In order to minimize this error what we can work on is the set of parameters defining the neural network output, that is the set of weights Θ. For this purpose we express the prediction of the network by means of some function g which is defined by the network architecture and is a function of the weights. Refer to the following 1.1 for a more rigorous definition.

$$\min_{\Theta} f(\Theta) = \min_{\Theta} E_{MSE}(\Theta) = \min_{\Theta} \frac{1}{N} \sum_{p=1}^{N} ||g(\Theta, x_p) - t_p||_2^2 \tag{1.1}$$

Furthermore although we do not have any constraint on the set of weights Θ, we would like to keep it relatively small as this leads to better machine learning models in practice. It is out of the scope of this report to accurately motivate this statement, so let it suffice to say that it can find justification in Vapnik's statistical learning theory[7].

In order to keep the value of the weights small we introduce a penalty that discourages solutions characterized by high value for the weights. This technique is known as regularization and it mainly comes into the L1 and L2 forms, or their linear combination *elastic net*[8], as depicted in 1.2.

$$L1 : \lambda||\Theta||_1 \qquad L2 : \lambda||\Theta||_2^2 \qquad \text{elastic net} : \lambda_1||\Theta||_1 + \lambda_2||\Theta||_2^2 \tag{1.2}$$

The use of regularization changes the initial objective defined in 1.1, resulting into the new objective defined as follows:

$$\min_\Theta h(\Theta) = \min_\Theta \left(\frac{1}{N} \sum_{p=1}^{N} ||g(\Theta, x_p) - t_p||_2^2 \right) + \lambda||\Theta||_1 \tag{1.3}$$

While in machine learning we would try a variety of different models, that is different $g(\Theta, x_p)$ and λ parameters, as to find the one that appears to generalize better, the objective for our project in this case is that of analyzing the behaviour of a number of optimization algorithms and therefore we will set a specific architecture for the network and investigate on the resulting function.

In particular we choose a setting with 1 hidden layer and 50 neurons. That means our parameter Θ is of dimension $(2,)$, with $\Theta_0 \in \mathbf{R}^{50\times10}$ and $\Theta_1 \in \mathbf{R}^{2\times50}$. This is the architecture that appeared to generalize better by placing 1st in the machine learning cup (ML cup) competition. As regularization term for the ML cup we used a slightly different one, an elastic net, but for the purposes of this project a L1 is required therefore we will take L1 part of the original regularization, that is 10^{-4} as value for λ. Moreover, $X = x_1, ..x_{762} \in \mathbf{R}^{762\times10}$ matrix and $x_i \in \mathbf{R}^{1\times10}$ $\forall i$. Just to give some context, matrix X represent the 762 labelled samples that we would train and validate our machine learning model on, Θ_1 is the weights matrix connecting the inputs of the network to the hidden layer and Θ_2 the weights matrix connecting the hidden layers to the output layer. The output is 2-dimensional therefore t_p and $g(\Theta, x_p)$ in the following are actually vectors $\in \mathbf{R}^{1\times2}$ and f_0, f_1 are respectively element-wise functions. The general form of the objective defined in 1.3 can now be instantiated with our chosen architecture and be written as in 1.4.

$$\begin{aligned} &f_0(x) = tanh(x) = \frac{e^x + e^{-x}}{e^x - e^{-x}} \\ &g(\Theta, x) = f_1(f_0(x\Theta_0^T)\Theta_1^T) \\ &\min_\Theta h(\Theta) = \min_\Theta \left(\frac{1}{762} \sum_{p=1}^{762} ||g(\Theta, x_p) - t_p||_2^2 \right) + 10^{-4}||\Theta||_1 \end{aligned} \tag{1.4}$$

Note that in the next sections we discuss the implemented gradient based methods to minimize the function h in 1.4, but first let us take a look at how the gradients are determined and what other relevant properties might characterize h.

1.2.0.1 Determining the gradients

Our h can seen as a sum of two functions: let us call the first part "MSE" and the second one L1 norm or regularization term. We can focus on determining the gradients of each part separately and then simply sum the results.

(Sub)gradients of L1 norm

Trying to determine the gradient of the L1 norm, we soon notice that this function is not fully differentiable. To tackle this problem we could either cheat and smooth the function around its point of non-differentiability or resort to subgradients. We take the latter path: remember that s is a subgradient of a convex function f at k if the inequality in 1.5 holds.

$$f(j) \geq f(k) + s(j-k) \quad \forall j \in domain(f) \tag{1.5}$$

The subgradients of the regularization term can be determined as in 1.6. Note that if $j = k = 0$ 1.5 is trivially satisfied for any s.

$$\begin{aligned}
& f(j) \geq f(k) + s(j-k) \quad \forall j \in domain(f) \\
& f(j) \geq s(j) \quad \forall j \in domain(f) && \{\text{f(k)=0, k=0 }\} \\
& \text{if } j > 0\text{: } \frac{\lambda|j|}{j} \geq s(j) \rightarrow s \leq \lambda && \{f(j) = \lambda|j| \ \} \\
& \text{else : } \frac{\lambda|j|}{j} \leq s(j) \rightarrow s \geq -\lambda
\end{aligned} \tag{1.6}$$

Therefore the subgradient of the L1 norm can be characterized as in 1.7

$$\begin{cases} \lambda & if x > 0 \\ -\lambda & if x < 0 \\ [-\lambda, \lambda] & if x = 0 \end{cases} \tag{1.7}$$

Gradients of MSE

This one requires a bit more thought to be done efficiently at every step of our iterative methods. Luckily someone has already done the thinking for us, therefore we limit ourselves to report how it is done.

In neural networks optimization we usually calculate the gradients with respect to the weights by using the backpropagation algorithm [9]. We first illustrate this algorithm for the case of a general neural network, with the aid of [10], as depicted in figure 1.2 and then apply it to our specific architecture shown in 1.3. Consider a weight θ^l_{kj} in figure 1.2 where the l index indicates that this is a weight connecting unit j in layer $l-1$ to unit k in layer l.

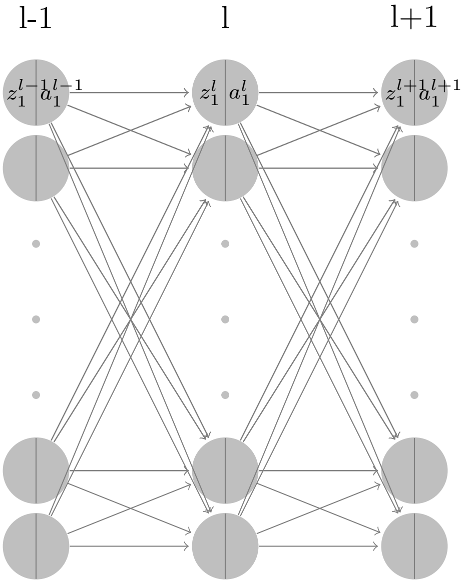


Figure 1.2: Schematic representation of three layers with an arbitrary number of units in a neural network

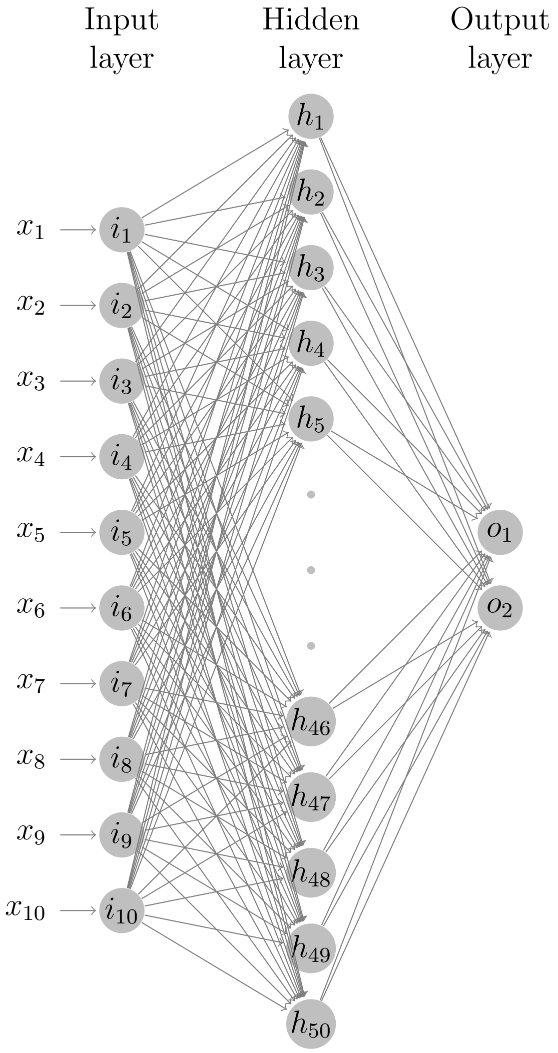


Figure 1.3: Architecture of our network. We have 10 inputs, 1 hidden layer with 50 units and 2 output units. Each unit h_i applies a *tanh* to the received input before propagating it forward. The output units are characterized by a linear activation.

We are interested in $\frac{\partial C}{\partial \theta^l_{kj}}$, the partial derivative of a weight with respect to a cost function C (MSE in our case). First we define z^l_k, that is the input of unit k in layer l, as in 1.8. Then the activation function σ of the unit is applied and its output defined as in 1.9. Similarly we can define the input of a generic unit m in layer $l + 1$ as in 1.10. We now have all we need to define $\frac{\partial C}{\partial \theta^l_{kj}}$ as in 1.11, and since the first term of the expanded equation 1.11 can be written as in 1.12, the second one as in 1.13 and the third as in 1.14, we obtain 1.15. Let us now define the error at a unit k in layer l as γ^l_k as in 1.16, and notice that it can be defined in function of the errors at layer $l + 1$. Finally once we have the final form of the partial derivative with respect to θ_{kj} defined as in 1.17, we only need to explicitly compute γ_j^L for the units j in the last layer L.

$$z_k^l = \sum_{j=1}^{n1} \theta_{kj}^l a_j^{l-1} \tag{1.8}$$

$$a_k^l = \sigma(z_k^l) \tag{1.9}$$

$$z_m^{l+1} = \sum_{j=1}^{n2} \theta_{mj}^{l+1} a_j^l \tag{1.10}$$

$$\frac{\partial C}{\partial \theta_{kj}^l} = \frac{\partial C}{\partial z_k^l}\frac{\partial z_k^l}{\partial \theta_{kj}^l} = \frac{\partial C}{\partial a_k^l}\frac{\partial a_k^l}{\partial z_k^l}\frac{\partial z_k^l}{\partial \theta_{kj}^l} \tag{1.11}$$

$$\frac{\partial C}{\partial a_k^l} = \sum_{i}^{n3} \frac{\partial C}{\partial z_i^{l+1}}\frac{\partial z_i^{l+1}}{\partial a_k^l} = \left(\frac{\partial z_i^{l+1}}{\partial a_k^l} = \frac{\partial \sum_{j=1}^{n2} \theta_{ij}^{l+1} a_j^l}{\partial a_k^l}\right) = \sum_{i}^{n3} \frac{\partial C}{\partial z_i^{l+1}} \theta_{ik}^{l+1} \tag{1.12}$$

$$\frac{\partial a_k^l}{\partial z_k^l} = \sigma'(z_k^l) \tag{1.13}$$

$$\frac{\partial z_k^l}{\partial \theta_{kj}^l} = \frac{\partial \sum_{j=1}^{n1} \theta_{kj}^l a_j^{l-1}}{\partial \theta_{kj}^l} = a_j^{l-1} \tag{1.14}$$

$$\frac{\partial C}{\partial \theta_{kj}^l} = \sum_{i}^{n3} \frac{\partial C}{\partial z_i^{l+1}} \theta_{ik}^{l+1} \sigma'(z_k^l) a_j^{l-1} \tag{1.15}$$

$$\gamma_k^l = \frac{\partial C}{\partial z_k^l} = \left(\sum_{i}^{n3} \frac{\partial C}{\partial z_i^{l+1}} \theta_{ik}^{l+1}\right) \sigma'(z_k^l) = \left(\sum_{i}^{n3} \gamma_i^{l+1} \theta_{ik}^{l+1}\right) \sigma'(z_k^l) \tag{1.16}$$

$$\frac{\partial C}{\partial \theta_{kj}^l} = \gamma_k^l a_j^{l-1} \tag{1.17}$$

Let us now define the partial derivatives specifically for our network (figure 1.3). Notice that we shall consider two cases: the weights connecting the input to the hidden layer and the weights connecting the hidden layer to the output one. Moreover for simplicity let use temporary neglect the regularization term which can simply be added to the resulting gradient. In the first case the partial derivative of a weight θ_{1mn} (we brought the layer index down) can be determined as in 1.18.

$$z_i^0 = \sum_{j=1}^{10} \theta_{0ij} x_{pj} \tag{1.18}$$

$$
\begin{aligned}
&a_i^0 = tanh(z_i^0)\\
&z_o^1 = \sum_{i=1}^{50} a_i^0 \theta_{1oi}\\
&a_o^1 = z_o^1\\
&\frac{\partial C}{\partial \theta_{1mn}} =\\
&= \frac{\partial \sum_{p=1}^{762} \sum_{o=1}^{2} (a_o^1 - t_{po})^2}{\partial \theta_{1mn}} =\\
&= \frac{\sum_{p=1}^{762} 2\,(a_m^1 - t_{pm})\, \partial a_m^1}{\partial \theta_{1mn}} =\\
&= \sum_{p=1}^{762} 2\left(a_m^1 - t_{pm}\right) a_n^0
\end{aligned}
$$

Note, with reference to equation 1.16, that in the last step of equation 1.18, $2\,(a_m^1 - t_{pm})$ is actually the error γ_m^1 on unit m in the output layer. Once we have this we can easily determine the partial derivative with respect to a weight $\theta_{0mn} \in \Theta_0$. Refer to 1.19 for a precise definition.

$$
\begin{aligned}
z_m^0 &= \sum_{j=1}^{10} \theta_{0mj} x_{pj} && (1.19)\\
a_m^0 &= tanh(z_m^0)\\
\gamma_m^0 &= \left(\sum_{i=1}^{2} \gamma_i^1 \theta_{im}^1\right)\left(1 - tanh^2(z_m^0)\right) && \text{(according to 1.16)}\\
\frac{\partial C}{\partial \theta_{0mn}} &= \gamma_m^0 x_n && \text{(according to 1.17)}
\end{aligned}
$$

Properties

Now that we have determined the gradients with respect to all the weights we might wonder if h has any favorable property that would let us define some clear expectations on the behaviour of the methods we apply in an attempt to optimize it. For instance we might inquire whether h is convex. Convex optimization is characterized by plenty of theoretical results, some of which we discuss later, and moreover remember that we resorted to subgradients - which has been proven to be a good idea in the convex setting (for instance, in [11] it is proven that by following the subgradient directions we will indeed end up close to the optimal value, for appropriate choices of the step size) whereas not nearly as much can be said in the non convex setting. Unfortunately neural networks models characterized by non-linear activation functions, such as *tanh* in our case, most of the time do not have this desirable property. In fact it can easily be seen, although we do not do it explicitly here, that this does not happen in our case either as the hessian of our problem is not positive (semi)definite.

Although it is not convex, it might be Lipschitz - that is its steepness does not suddenly increase arbitrarily. While it is easy to see that the L1 norm is Lipschitz by simply exploiting the triangle inequality, determining whether the first part of our function is Lipschitz calls for a different approach. We know that a everywhere-differentiable function is Lipschitz if and only if its derivative is bounded. We have just computed the gradients so it is easy to verify that this does not hold: refer to 1.20 to concretize this statement.

$$\begin{aligned}\lim_{\theta_{1mn}\to\infty}\frac{\partial C}{\partial\theta_{1mn}} = \lim_{\theta_{1mn}\to\infty}\sum_{p=1}^{762} 2(a^1_{mn} - t_{pm})a^0_n = \qquad (1.20)\\ = \lim_{\theta_{1mn}\to\infty}\sum_{p=1}^{762}\sum_{i=1}^{50} 2\left(tanh(z^0_i)\theta_{1mi} - t_{pm}\right) tanh(z^0_n)\end{aligned}$$

In 1.20 we can see how the limit may very well go to infinity as θ_{1mn} does. Therefore the derivative is not bounded which implies our h is not Lipschitz.

Chapter 2

Beyond gradient descent

In this part we describe the implemented algorithms from a theoretical perspective, providing the main intuitions on their operating principles with a focus on their application in machine learning.

Before discussing more advanced methods, we first briefly look at the classical gradient approach adopted in machine learning. Let Θ be the weight matrix of our neural network which represents some function $g(\Theta)$. The goal is that of minimizing some cost function $C(g, D)$ where D is the set of available data that we want to fit and hopefully represents the problem's true data distribution. In order to do this we can either use an exact solution in certain situations or proceed in an iterative fashion. In most cases the exact solution is not viable due to the definition or size of the problem. We therefore focus on the second approach.

With gradient descent, to minimize the cost function C we take small steps on its surface using the information provided by the current location. A step is defined as an update of the weight matrix Θ, of the following form:

$$\Theta_t = \Theta_{t-1} - \eta \Delta C(\Theta_{t-1}) \tag{2.1}$$

Intuitively we adopt a greedy approach that is taking a step in the steepest descent direction, with respect to the current location, which is given by the gradient, the opposite of the partial derivatives of the cost function with respect to the weights. The η parameter in 2.1 denotes the step size along the direction given by the gradient. There exist a multitude of ways[12] to determine the step size, ranging from a chosen fixed value to a more dynamic search such as line search[13].

For this method to work we assume that the cost function is differentiable. In this regard some established functions, to be used based on the task at hand, have emerged, such as Mean Squared Error[14] and Cross Entropy. However the simplicity of the basic gradient descent algorithm as sketched in 2.1 is both a blessing and a curse as its convergence can be quiet slow. For this reason there is a lot of interest in improving and speeding up this method while retaining the same basic assumptions.

Generally the theory of optimization gives many guarantees for (strongly) convex functions but in practice, most of the time, in machine learning the problems cannot be assumed to fit in this setting and therefore most of the so long-sought guarantees fall short. Although some work[15] has been done in the non-convex setting as well, there is currently, to the best of our knowledge, a much less solid theoretical ground than the one found in the (strongly) convex case.

In the next sections we discuss a number of methods that usually achieve an improvement over the simple gradient descent method.

2.1 Accelerated gradient methods

2.1.1 Gradient descent with momentum

Looking at the update of the gradient descent algorithm we can clearly notice that it depends only on the current location. An idea, attributed to Polyak, is to add a sort of memory to our short-sighted wandering on the cost function's surface. To see why this might be useful think of how a basic gradient descent approach, without smart step size adaption, would move on a downhill surface - it can either use a fixed step size and risk to overshoot the minimum or apply a step size decay over the iterations and risk incurring in a slow convergence or early stopping. In this same situation momentum, even with a smaller step size, can benefit from its memory of agreeing past directions to reinforce the step size thus ultimately converging. Momentum reduces the overall greediness of the algorithm by taking into a account exponentially weighted average of the previous directions.

To get a better grasp on how this is done, consider the following classical momentum method:

$$z_t = \beta z_{t-1} - \eta \Delta C(\Theta_{t-1}) \tag{2.2}$$

$$\Theta_t = \Theta_{t-1} + z_t \tag{2.3}$$

In 2.2 and 2.3 we can see how the update z_t depends on the current gradient as well as the past direction z_{t-1}, scaled by a factor β. When β is zero we obtain the gradient descent equation in 2.1 once again, when it is one the direction quickly becomes insensitive to the local direction given by the gradient. A typical value for β is around 0.9.

There is also another type of momentum method, attributed to Nesterov, defined as follows:

$$z_t = \beta z_{t-1} - \eta \Delta C(\Theta_{t-1} + \beta z_{t-1}) \tag{2.4}$$

$$\Theta_t = \Theta_{t-1} + z_t \tag{2.5}$$

Written in this form we can see that Nesterov's momentum differs from classical momentum in the point where gradient is computed. In 2.2 and 2.3 the gradient is computed in the current location and then summed up with the accumulated past momentum while in 2.4 and 2.5 the gradient is computed in the point where the current accumulated momentum would lead, thus effectively "looking ahead" before deciding the contribution of the new direction.

Interesting convergence properties of the momentum approach can be derived from a physics analogy of a particle subject to Newton's laws of motion: it can be shown that the total energy of the so resulting system is a decreasing Lyapunov[16] function that converges to the same equilibrium found without the addition of the momentum and moreover, its convergence speed is accelerated by the momentum. Much deeper and darker details of this can be found in [17].

A perhaps more concrete result can be show when we consider the convex quadratic setting where the function to optimize is of the form $f(x) = x^T Qx + qx$. For properly chosen β and η as shown in 2.6 we have that the distance from the optimum at iteration $i+1$ can be constrained by the one at iteration i as shown in 2.7 [18]:

$$\eta = \frac{4}{(\sqrt{\lambda^1} + \sqrt{\lambda^n})^2}, \quad \beta = max\{|1 - \sqrt{\alpha\lambda^n}|, |1 - \sqrt{\alpha\lambda^1}|\}^2 \tag{2.6}$$

$$||x^{i+1} - x_*|| \leq \left(\frac{\sqrt{\lambda^1} - \sqrt{\lambda^n}}{\sqrt{\lambda^1} + \sqrt{\lambda^1}} \right) ||x^i - x_*|| \tag{2.7}$$

Although it might not seem that much of an improvement over gradient, where we do not have the square roots, if we perform a sufficient number of iterations like we usually have in machine learning then this result proves to be quite significant. A more precise analysis of the convergence of this method in the convex case, showing a $O(1/k)$, with k being the number of iterations, speed can be found in [19]. Once again however this is not guaranteed at all when we consider the non-convex case.

Turning our attention to the case of interest, the non-convex one, we mention the convergence analysis done in [20] where the proposed iPiano algorithm yields a convergence rate of $O(1/\sqrt{k})$ for the problem:

$$\min_{x \in \mathbb{R}^N} h(x) = f(x) + g(x) \tag{2.8}$$

where f is some smooth, possibly non-convex, function and g is a convex, possibly non-smooth function: $\mathbb{R}^N \to \mathbb{R} \cup \{+\infty\}$. In particular stronger results are shown for h satisfying the Kurdyka-Lojasiewicz property[21]. Without delving further into the somewhat tricky details we also mention that the analysis for the non-convex case has also been extended to the stochastic case, an important result when working with large-scale machine learning problems, in [22].

Much more work than what mentioned here has been done, but it still feels like the gap between the theory and practice for the non-convex case has not been closed yet as for instance in [23] empirical results demonstrate how a proper initialization and momentum term can achieve performances comparable to those obtained by hessian-free optimization[24] in the training of deep networks.

2.1.2 Adam

Adam[25] is a first order gradient based method that uses adaptive first and second order moments of the gradient. Consider the following snippet extrapolated from [25] :

$$\begin{aligned}
&g_t = \Delta C(\Theta_{t-1}) \\
&m_t = \beta_1 * m_{t-1} + (1 - \beta_1) * g_t \\
&v_t = \beta_2 * v_{t-1} + (1 - \beta_2) * g_t^2 \\
&\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \\
&\hat{v}_t = \frac{v_t}{1 - \beta_2^t} \\
&\Theta_{t+1} = \Theta_t - \frac{\eta * \hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}
\end{aligned} \tag{2.9}$$

Similarly to momentum, Adam keeps a exponentially moving average m_t of the past gradients. Additionally, it also keeps an average of the past squared gradients v_t. Since m_t and v_t are initialized as 0 vectors and the initial values for $\beta_1, \beta_2 \in [0, 1)$ are typically quite high, in the order of 0.9 for β_1 and 0.999 for β_2, these moving averages are biased towards 0, hence the corrected estimates $\hat{m}_t$ and $\hat{m}_t$. Adam updates result in an improvement over non-adaptive step methods and can be shown to converge faster than gradient descent with momentum under certain conditions, related to the sparseness of features. We do not repeat the convergence analysis done in the original paper, which is valid in the convex setting only. No guarantees of improvement over standard gradient descent are given in the non-convex case although it has been empirically shown to greatly outperform it in terms of speed.

Despite the recent success of adaptive step size methods, uncertainties about their performance in terms of generalization have arisen. In [26], Wilson et al.' investigation demonstrate how stochastic gradient descent(SGD) leads to significantly better generalization capabilities although often outperformed at the training level. In [27] an algorithm that dynamically switches from Adam to SGD in order to improve generalization is proposed. This is justified by observing that Adam often outperforms SGD in the initial stages but loses the edge as the iterations progress.

2.1.2.1 Adamax

Adamax is a modification of Adam substituting the update for v_t with the infinity norm. In Adam the L_2 is used but we can think of replacing it with any other norm, however this is not done in practice for norms other than the infinity one as it would lead to unstable updates. Adamax update rule for v_t then becomes:

$$v_t = \max\{\beta_2 * v_{t-1}, |g_t|\} \tag{2.10}$$

2.1.3 RMSprop

The core idea of RMSprop builds on the rprop[28] algorithm, that does not take into account the magnitude of the gradients but only their sign as to make balanced updates, as well as the adaptive learning methods. Although rprop works fairly well in the case of full-batch optimization, that is using the full dataset to determine the gradient at each step, by not taking into account the magnitude of the gradients the updates over a series of mini-batches do not balance out thus breaking the core assumption of SGD. For instance consider the case of a series of nine mini-batches yielding a 0.01 values for the gradient and a subsequent tenth mini-batch yielding a gradient of -0.2. Rprop would update the weight ten times with the same step size, positively nine times and negatively once, resulting in an increment of the weight. But looking at the magnitude of the gradients we notice that the desirable direction would actually be the other one.

RMSprop[29] concretizes this intuition replacing the magnitude-blind updates with updates scaled by a moving average of the past squared gradients. Formally:

$$\begin{aligned} R_t &= \delta R_{t-1} + (1-\delta)\Delta C^2 \\ \Theta_{t+1} &= \Theta_t - \frac{\eta}{\sqrt{R_t + \epsilon}}\Delta C \end{aligned} \tag{2.11}$$

In 2.11 δ is a hyperparameter that determines the influence of the past gradients. Empirically a value of 0.9 has been proven to work well, and ϵ is a stability term whose typical value is around 10^{-6}.

2.1.4 Adine

We briefly present a recent (Dec. 2017) algorithm based on adaptive momentum updates introduced by Srinivasan et al[30]. The idea is based on the fact that in high-dimensional non-convex optimization problems, although local minima become less of a issue, the number of saddle points rapidly increase[31]. Srinivasan et al. argue that a momentum term ≥ 1 may prove helpful in escaping such points. We report in 2.12 the proposed algorithm.

$$
\begin{aligned}
&l_{t-1} = C(\Theta_{t-1}) \\
&\hat{l}_t = \frac{\hat{l}_t + l_{t-1}}{2} \\
&\text{if } \hat{l}_t > \zeta \hat{l}_{t-1}\text{: } \beta = \beta_s \\
&\text{else: } \beta = \beta_g \\
&z_t = \beta z_{t-1} - \eta \Delta C(\Theta_{t-1} + \beta z_{t-1}) \\
&\Theta_t = \Theta_{t-1} + z_t
\end{aligned}
\tag{2.12}
$$

With reference to 2.12, β_s is the usual standard momentum term $\in [0, 1)$ and β_g is the greater momentum term ≥ 1. The momentum is adaptive in the sense that it dynamically switches between β_s and β_g, according to the progression of the function value and a tolerance hyperparameter $\zeta > 0$. Theoretical results have not been published yet, therefore we limit ourselves to empirically investigate the behaviour of this algorithm and compare it with the other approaches taken.

2.2 Additional techniques

2.2.1 Line search

In the introduction we briefly suggested that there are a multitude of ways of determining the step size η. While machine learning people usually do not use this, we feel it is worthwhile discussing it.

The basic idea of line search is that of choosing the ”right” step size in the gradients’ direction at every iteration. Ideally, in a convex setting, we would like to choose a step size η, at iteration k, such that:

$$\min_{\eta} f(x_k + \eta p_k) \tag{2.13}$$

where p_k is a descent direction. This is called exact line search but although tempting, it easily becomes computationally unfeasible and in practice we have to settle for a compromise. For instance a much more reasonable alternative is backtracking line search[32] defined as in 2.14.

$$f(x_k + \eta p_k) \leq f(x_k) + c_1 \eta_k p_k^T \Delta f(x_k) \tag{2.14}$$

where p^k is a descent direction. In 2.14 $c_1 \in (0, 1)$ is some fixed parameter, and at each step we update the step size η multiplying it by some constant $\rho \in (0, 1)$ until the condition is satisfied. This is line search with the Armijo condition[33]. However we typically need additional constraints to avoid taking steps that are too small in terms of decrease in f, and that is when the Wolfe[34] conditions come into play. In 2.15 we report the strong variant that is typically used in combination with the Armijo condition.

$$-p_k^T \Delta f(x_k + \eta_k p_k) \leq -c_2 p_k^T \Delta f(x_k) \tag{2.15}$$

The use of line search in machine learning meets conflicting opinions. While it takes off the user the burden of tuning the step size, it appears not to perform as well in the presence of noise[35]. On another note, besides the improved efficiency, in machine learning we usually use noisy gradients as they often lead to better generalization [36]. Moreover one might argue that taking possibly non-optimal steps is more fruitful than repeatedly evaluating the function along one direction.

In 3 we report, among others, our non-extensive experiments with simple line search approaches on the given optimization problem.

2.2.2 Conjugate Gradient

Here we take a look at a solidly theoretically grounded modification of gradient descent, for the quadratic class of problems, that is conjugate gradient. The main intuition supporting this method is given by noticing that two successive directions of standard gradient descent empowered by exact line search are orthogonal. However when we consider directions of steps farther apart, this argument no longer holds. We would like this orthogonality property to be shared by all the directions, that is sort of extending the memory of line search across all iterations and therefore not compromise earlier steps. It turns out that this can be done by exploiting the previous directions as shown in 2.16 [37]:

$$\beta_t = \frac{||\Delta C(\Theta_t)||^2}{||\Delta C(\Theta_{t-1})||^2} \tag{2.16}$$

In the non-linear setting things are not so easy and it turns out that convergence may vary significantly depending on how we determine β. In fact a multitude of ways to do so have been proposed, among which we mention and implement Fletcher-Reeves and Polak-Ribire[38] which are computed as shown in CG1.1. Once we have β the update of the weights is simply done as shown in 2.17.

$$\beta_t = \frac{\Delta C^T(\Theta_t)(\Delta C(\Theta_t) - \Delta C(\Theta_{t-1}))}{\Delta C^T(\Theta_{t-1})\Delta C(\Theta_{t-1})} \tag{CG1.1: Fletcher-Reeves}$$

$$\beta_t = \frac{\Delta C^T(\Theta_t)\Delta C(\Theta_t)}{\Delta C^T(\Theta_{t-1})\Delta C(\Theta_{t-1})} \tag{CG1.2: Polak-Ribire}$$

$$\begin{aligned} p_t &= -\Delta C(\Theta_t) + \beta_t p_t \\ \Theta_{t+1} &= \Theta_t + \eta p_t \end{aligned} \tag{2.17}$$

In practice a small tweak appears to help convergence, that is restarting every once in a while. Restarting simply means resetting the direction to the one indiated by the gradient. Some possible heuristics to decide when to restart are given by the space dimension, or in case we use the Polak-Ribire' version convergence in some settings is guaranteed by using an implicit restart setting $\beta = max\{0, \beta\}$.

[redacted]

[redacted]

[redacted]

[redacted]

Chapter 3

Experiments

In this chapter we present our experiments. First in 3.1 we assess the correctness of our implementation, then we show the results of some of the most relevant experiments performed in order to further understand the behaviour of the various algorithms when facing a non-convex non-linear problem given the lack of guarantees in this context. In 3.2 we analyze the behaviour of the accelerated gradient optimizers varying their parameters, while in 3.3 we compare some of the best settings of each optimizer with each other. Finally in 3.4 we investigate the effect of additional optimizers and techniques and in 3.5 we take a brief look at the generalization capabilities of the model.

3.1 Assessment of correctness

Before further analyzing the behaviour of the different optimization algorithms, we validated the correctness of our implementation in two different ways.

As a first step in the assessment of correctness of the implemented algorithms we tested them on a variety of well-known test functions such as, Matyas, Rosenbrock and Himmelblau functions, whose properties are clearly observable. Moreover we could easily plot and observe the behaviour of the different optimizers on these functions whereas for our given optimization problem this is not possible due to the high-dimensionality. In tables 3.1 and 3.2 we report some data from the tests on the different functions. We report only the results on a convex function (Matyas) and a non-convex one (Himmelblau). In the table the different parameter settings are shown as well as the number of iteration needed to arrive in a range of 10^{-5} around the minimum.

The template to run different optimization algorithms as well as to plot their progress during the epochs is available in the attached `optimize_2d.py` file. An example of the effect of the momentum term is shown in figure 3.1

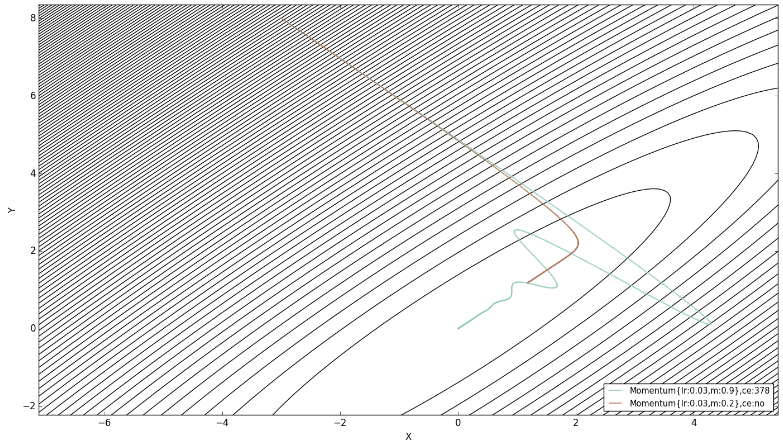


Figure 3.1: Comparison of SGD with a momentum term of 0.9 (light blue) vs a momentum term of 0.2 (brown). It can be seen how the the higher momentum term accelerates the convergence but oscillates more as it takes longer to decelerate once the direction changes.

To further convince ourselves that our implementation of different algorithms was indeed working as supposed to, we compared our results, on a number of tasks, with those obtained by using the same algorithms and the same hyperparameters with keras[44], a reliable high level library that provides a broad range of optimizers.

Keras implements many of the accelerated gradients methods among which SGD with momentum (2.1.1), adam (2.1.2), adamax (2.1.2.1), RMSProp (2.1.3). Therefore we were able to compare the behavior of our algorithms with keras' implementation on both the given problem as well as on randomly generated ones. To compare keras' behaviour with that of our optimizers, refer to the attached file `clean_keras_comp.py`. The minimal difference in this comparison with keras, coupled with the results obtained on the test functions, seemed like a sufficient confirmation of the correctness of our implementation and therefore we moved on to a deeper exploration of the behaviour by tuniing the various parameters on our real optimization problem (1.2).

Table 3.1: Himmelblau function. Starting point is $(-7, 8)$. This function is characterized by four identical local minima and a maximum. Although we do report it here, by playing around with the `optimize_2d.py` file, it can be seen how different optimizers converge to different local minima based on their characteristics such as the momentum term which may result into jumping over the closest stationary point.

Function	Parameters	Iteration to converge
ConjugateGrad	restart:2,ls:AW{η:0.0001,c_1:0.0001, c_2:0.9,r:0.9,i:1000}	3
SDG	ls:AW{η:0.0001,c_1:0.0001, c_2:0.9,r:0.9,i:1000}	3
Momentum	η:0.003, β:0.0	16
Adine	η:0.003, β_s:0.5,β_g:1.0001,t:1	25
Adine	η:0.003, β_s:0.0,β_g:1.0001,t:1	26
Momuentum	η:0.003, β:0.5	29
Adine	η:0.003, β_s:0.9,β_g:1.0001,t:1	41
Adam	η:0.15, β_1:0.5, β_2:0.5	47
Momentum	η:0.003, β:0.9	110
RMSprop	η:0.06, δ:0.9,	124
Adam	η:0.15, β_1:0.9, β_2:0.2	167

Table 3.2: Matyas function. Starting point is $(5, 5)$. This is a convex function with the global minimum in $(0, 0)$. Notice that starting from $(5, 5)$ our optimizers ideally only need to move in a straight line. Indeed in this case the momentum term speeds up the convergence. Line search as expected converged in 1 iteration.

Function	Parameters	Iteration to converge
ConjugateGrad	restart:-1,ls:AW{η:0.0001,c_1:0.0001, c_2:0.9,r:0.9,i:1000}	1
SDG	ls:AW{η:0.0001,c_1:0.0001, c_2:0.9,r:0.9,i:1000}	1
Adine	η:0.3, β_s:0.9,β_g:1.0001,t:1	14
RMSprop	η:0.3, δ:0.9,	15
Adine	η:0.3, β_s:0.5,β_g:1.0001,t:1	16
Adam	η:0.3, β_1:0.9, β_2:0.999	17
Momentum	η:0.3, β:0.9	18
Adine	η:0.3, β_s:0.0,β_g:1.0001,t:1	18
Adamax	η:0.3, β_1:0.9, β_2:0.999	23
Momentum	η:0.3, β:0.5	94
Momentum	η:0.3, β:0.0	191

3.2 Parameters exploration

After an initial testing to rule out excessively high and excessively low step sizes, we experimented with 48 configurations covering a reasonable range for each optimizer. In total we ran 5 trials, with each trial characterized by a different initialization of the parameters. We then averaged the results as to avoid being mislead as a consequence of a lucky or unlucky initialization. Besides visually looking at how the value of the function progresses over the iteration, the so-called learning curve in machine learning terms, we thought some numerical measure describing the "goodness" of the optimization process would prove to be an useful addition that would enrich the analysis by, for instance, making it more scalable. To this purpose in the following we consider three measure in an attempt to capture the different aspects that characterize a good optimization progression:

- *final value achieved*: this is simply the last value of the function achieved at the end of the iterations. Note that this value would not be as significant from a machine learning point of view as it gives little information on the generalization capabilities. Moreover it may not correspond to the lowest value achieved over the iterations in case of unstability;

- *unstability*: this is a measure of the unstability of the algorithm. More precisely it takes into account the relative increases of the function value over the iterations. Let x be an array whose elements x_i represent the value of the function at iteration i. Refer to the following 3.1 for a precise definition of the unstability:

  $$unst(x) = \frac{10^5 \sum_{i=2}^{|x|} max(0, \frac{x[i]-x[i-1]}{x[i-1]})}{|x|} \tag{3.1}$$

  Basically we sum up the relative increases of the function' values and then normalize based on the number of iterations performed. Note that this a rough heuristic measure with some properties that based on the situation could not be desirable: for instance it gives the same weight to increases occurring at the beginning of process as to those towards the end.

- *convergence speed*: this is a measure indicating within how many iterations the algorithm first reached a value within some range of the final optimal value achieved. This measure ranges from 0 to 100, where 100 indicates that on the first iteration the function value was already within the desired range of the final value reached. Note that this measure is not significant on its own and must be combined with the final value achieved measure. Refer to equation 3.2 for a more precise definition.

$$cs(x) = 100 - \frac{\arg\max_i \left(x[i] <= min(x) * \left(1 + \frac{\rho}{100}\right)\right) * 100}{|x|} \tag{3.2}$$

$\rho >= 0$ *is a percentage indicating the desired range within the final value.*

Moreover we consider the correlation between the parameters of the various optimizers, the final value achieved, the unstability and the convergence speed. To do so we resort to Pearson and Spearman correlation coefficients. Since the results proved to be quite similar we only report Pearson (PCC) for brevity. Remember that PCC between two variables containing a set of observations is defined as in 3.3:

$$PCC(x, y) = \frac{cov(x, y)}{std(x)std(y)} \tag{3.3}$$

PCC is a measure of linear correlation that ranges between -1 and 1. A value of -1 indicates that as x increases y decreases, a value of 1 indicates that as x increases so does y and a value of 0 indicates the absence of linear correlation.

In the following paragraphs we look at the accelerated gradient methods individually. Keep in mind that all the results are to be considered as the average over 5 trials consisting of 10^5 iterations each. Before proceeding, some general considerations can be made:

- The step size is directly proportional with the unstability: that is taking a bigger step may result into going too far in one direction and thus moving into a point where the function' value is higher.

- The step size is directly proportional to the the final value reached: as per the previous point, by going too far into one direction then too far into another one, we might just never hit the good spots. No surprise here.

- The step size is directly proportional to the convergence speed: by taking big step we are most likely to arrive close to our final location sooner.

However that is most likely not a desirable place to end up into as we will see.

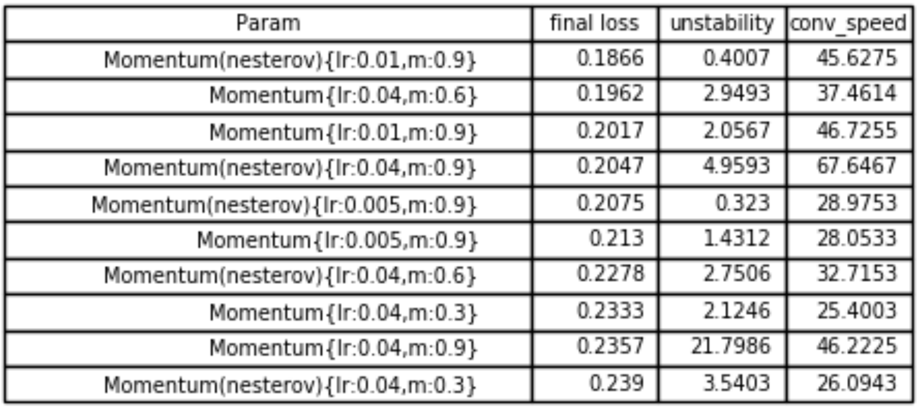
- As the unstability increases the final value reached tends to be a worse one: high unstability means we keep going the wrong way, so once again it is no surprise that we are not able to reach a good value.

- Unstability is directly proportional to convergence speed: similarly to the argument made for step size and convergence speed, this is most likely due to the final location being a bad one. When the final value is taken into account this can easily be seen.

The following optimizer-specific considerations are to be taken with caution as they are highly related to the given problem as well as to the range of parameters employed. They may serve as an indication on to how to modify each specific algorithm so that it performs better on the given task. In the next paragraphs we denote the last value function value at the end of the optimization process by μ, the unstability by *unst* and the convergence speed by *cs*.

SGD with momentum

The 10 best results for SGD with momentum in terms of last function value achieved are shown in figure 3.2. The PCC matrix of {*step size, β, nest, last function value achieved, unstability, convergence speed*} is shown in table 3.3. We start by contradicting one the general considerations made just earlier as in this case a higher step size leads to a lower final value of the function. However this is not all too surprising as after 10^5 iterations most of the configurations with a lower step size were still descending. The momentum term speeds up the convergence and leads to better final values but appears to lead to higher unstability, thus failing to deliver that stabilizing effect we would wish for. The nesterov version, while not affecting the final value reached, appears to get there with a cleaner path: this is to be expected as we recall that the idea of nesterov is that of looking ahead first, which means possibly avoiding taking bad paths that would lead to higher unstability.



Param	final loss	unstability	conv_speed
Momentum(nesterov){lr:0.01,m:0.9}	0.1866	0.4007	45.6275
Momentum{lr:0.04,m:0.6}	0.1962	2.9493	37.4614
Momentum{lr:0.01,m:0.9}	0.2017	2.0567	46.7255
Momentum(nesterov){lr:0.04,m:0.9}	0.2047	4.9593	67.6467
Momentum(nesterov){lr:0.005,m:0.9}	0.2075	0.323	28.9753
Momentum{lr:0.005,m:0.9}	0.213	1.4312	28.0533
Momentum(nesterov){lr:0.04,m:0.6}	0.2278	2.7506	32.7153
Momentum{lr:0.04,m:0.3}	0.2333	2.1246	25.4003
Momentum{lr:0.04,m:0.9}	0.2357	21.7986	46.2225
Momentum(nesterov){lr:0.04,m:0.3}	0.239	3.5403	26.0943

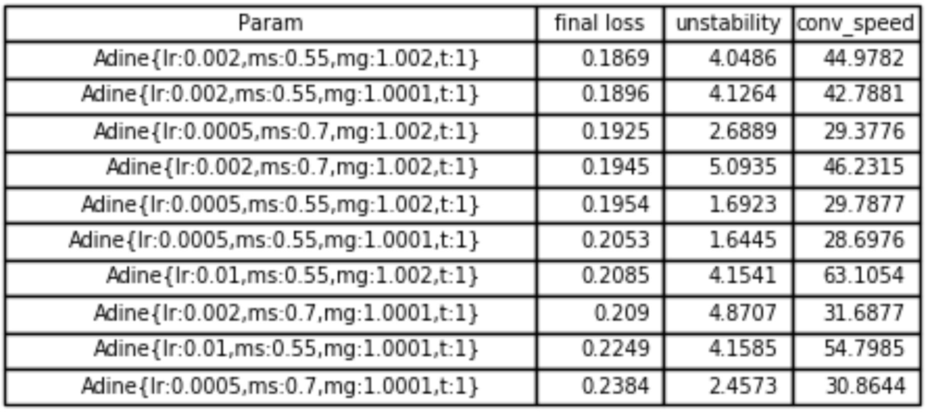
Figure 3.2: Table of the 10 best momentum configurations

Table 3.3: Parameters correlation of SGD with momentum. The momentum term is β.

	η	β	nest	μ	*unst*	*cs*
η	1	0	0	-0.6252	0.6641	0.0239
β	0	1	0	-0.3005	0.1589	0.2112
nest	0	0	1	0.0003	-0.1134	0.0283
μ	-0.6252	-0.3005	0.0003	1	-0.3513	0.4592
unst	0.6641	0.1589	-0.1134	-0.3513	1	0.2228
cs	0.0239	0.2112	0.0283	0.4592	0.2228	1

Adine

The 10 best results for adine in terms of last function value achieved are shown in figure 3.3. The PCC matrix of {*step size*, β_s, β_g, *last function value achieved, unstability, convergence speed*} is shown in table 3.4. First of all we can clearly see in appendix A that all the settings in which the tolerance parameter ζ is equal to 0.95 were not able to reach a satisfying final value. Recall that ζ determines when the change from standard momentum to greater momentum occurs: the higher it is, the more we can risk temporarily increasing the loss value without switching back to the standard momentum. Thus a higher tolerance may be seen as taking bigger risks, and it this case it appears to pay off. Much like standard momentum, here too the momentum term β_s leads to better final value but increases the unstability. The trade-off however is not quite as good and a more moderate value of β_s appears to work best.



Param	final loss	unstability	conv_speed
Adine{lr:0.002,ms:0.55,mg:1.002,t:1}	0.1869	4.0486	44.9782
Adine{lr:0.002,ms:0.55,mg:1.0001,t:1}	0.1896	4.1264	42.7881
Adine{lr:0.0005,ms:0.7,mg:1.002,t:1}	0.1925	2.6889	29.3776
Adine{lr:0.002,ms:0.7,mg:1.002,t:1}	0.1945	5.0935	46.2315
Adine{lr:0.0005,ms:0.55,mg:1.002,t:1}	0.1954	1.6923	29.7877
Adine{lr:0.0005,ms:0.55,mg:1.0001,t:1}	0.2053	1.6445	28.6976
Adine{lr:0.01,ms:0.55,mg:1.002,t:1}	0.2085	4.1541	63.1054
Adine{lr:0.002,ms:0.7,mg:1.0001,t:1}	0.209	4.8707	31.6877
Adine{lr:0.01,ms:0.55,mg:1.0001,t:1}	0.2249	4.1585	54.7985
Adine{lr:0.0005,ms:0.7,mg:1.0001,t:1}	0.2384	2.4573	30.8644

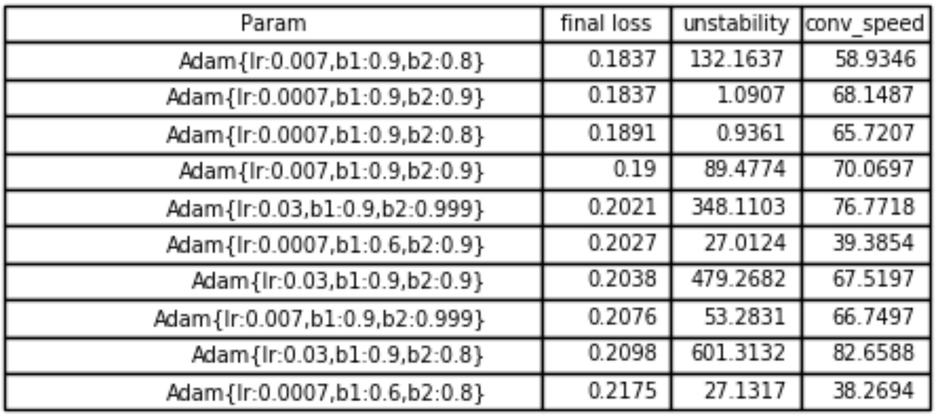
Figure 3.3: Table of the 10 best adine configurations

Table 3.4: Parameters correlation of Adine

	η	β_s	β_g	μ	*unst*	*cs*
η	1	0	0	-0.3325	0.5179	0.3838
β_s	0	1	0	-0.1698	0.2733	-0.2227
β_g	0	0	1	-0.0019	0.0093	0.0825
μ	-0.3325	-0.1698	-0.0019	1	-0.2595	-0.1573
unst	0.5179	0.2733	0.0093	-0.2595	1	0.454
cs	0.3838	-0.2227	0.0825	-0.1573	0.454	1

Adam

The 10 best results for Adam in terms of last function value achieved are shown in figure 3.4. The PCC matrix of {*step size, b1, b2, last function value achieved, convergence speed*} is shown in table 3.5. Adam appears to largely benefit from its β_1 and β_2 parameters. Looking at the PCC matrix in 3.5 we notice how an increase in β_1 leads to a significant decrease in the final value while barely affecting the unstability. On the other hand β_2, while providing a more moderate decrease in the final value appears to have a stabilizing effect, although slowing the convergence.



Param	final loss	unstability	conv_speed
Adam{lr:0.007,b1:0.9,b2:0.8}	0.1837	132.1637	58.9346
Adam{lr:0.0007,b1:0.9,b2:0.9}	0.1837	1.0907	68.1487
Adam{lr:0.0007,b1:0.9,b2:0.8}	0.1891	0.9361	65.7207
Adam{lr:0.007,b1:0.9,b2:0.9}	0.19	89.4774	70.0697
Adam{lr:0.03,b1:0.9,b2:0.999}	0.2021	348.1103	76.7718
Adam{lr:0.0007,b1:0.6,b2:0.9}	0.2027	27.0124	39.3854
Adam{lr:0.03,b1:0.9,b2:0.9}	0.2038	479.2682	67.5197
Adam{lr:0.007,b1:0.9,b2:0.999}	0.2076	53.2831	66.7497
Adam{lr:0.03,b1:0.9,b2:0.8}	0.2098	601.3132	82.6588
Adam{lr:0.0007,b1:0.6,b2:0.8}	0.2175	27.1317	38.2694

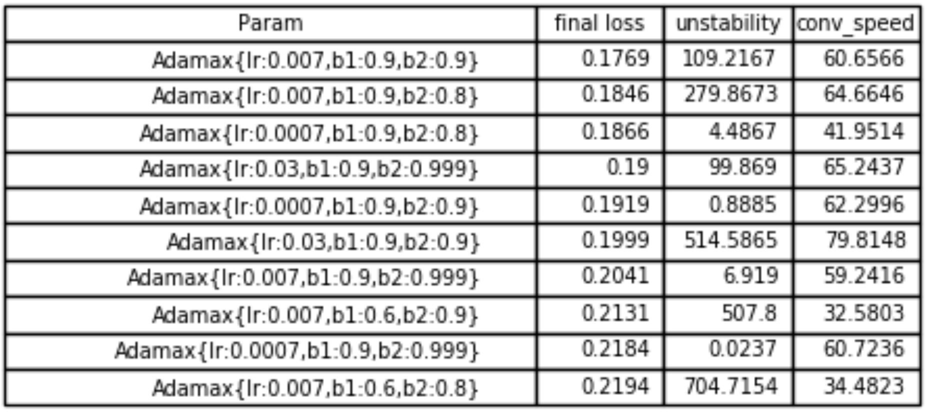
Figure 3.4: Table of the 10 best Adam configurations

Table 3.5: Parameters correlation of Adam

	η	β_1	β_2	μ	*unst*	*cs*
η	1	0	0	0.7245	0.8766	0.751
β_1	0	1	0	-0.3605	-0.0816	0.2116
β_2	0	0	1	0.0222	-0.2375	0.0126
μ	0.7245	-0.3605	0.0222	1	0.6433	0.5661
unst	0.8766	-0.0816	-0.2375	0.6433	1	0.676
cs	0.751	0.2116	0.0126	0.5661	0.676	1

Adamax

The 10 best results for Adamax in terms of last function value achieved are shown in figure 3.5. The PCC matrix of {*step size, b1, b2, last function value achieved, convergence speed*} is shown in table 3.6. We can see that the correlation between the parameters is pretty similar to the one in Adam: this is to be expected due to the similarity of the two algorithms. The only relevant difference is given by the β_2 parameter which here appears to improve the final value as well as to maintain its stabilizing effect.



Param	final loss	unstability	conv_speed
Adamax{lr:0.007,b1:0.9,b2:0.9}	0.1769	109.2167	60.6566
Adamax{lr:0.007,b1:0.9,b2:0.8}	0.1846	279.8673	64.6646
Adamax{lr:0.0007,b1:0.9,b2:0.8}	0.1866	4.4867	41.9514
Adamax{lr:0.03,b1:0.9,b2:0.999}	0.19	99.869	65.2437
Adamax{lr:0.0007,b1:0.9,b2:0.9}	0.1919	0.8885	62.2996
Adamax{lr:0.03,b1:0.9,b2:0.9}	0.1999	514.5865	79.8148
Adamax{lr:0.007,b1:0.9,b2:0.999}	0.2041	6.919	59.2416
Adamax{lr:0.007,b1:0.6,b2:0.9}	0.2131	507.8	32.5803
Adamax{lr:0.0007,b1:0.9,b2:0.999}	0.2184	0.0237	60.7236
Adamax{lr:0.007,b1:0.6,b2:0.8}	0.2194	704.7154	34.4823

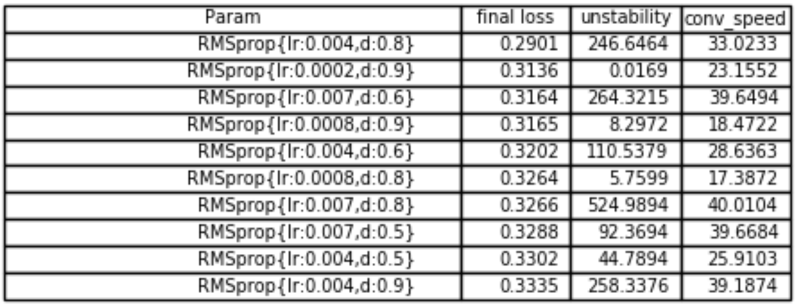
Figure 3.5: Table of the 10 best Adamax configurations

Table 3.6: Parameters correlation of Adamax

	η	β_1	β_2	μ	$unst$	cs
η	1	0	0	0.6193	0.8727	0.553
β_1	0	1	0	-0.4058	0.0019	-0.0819
β_2	0	0	1	-0.2149	-0.2542	-0.2214
μ	0.6193	-0.4058	-0.2149	1	0.7152	0.4515
$unst$	0.8727	0.0019	-0.2542	0.7152	1	0.5164
cs	0.553	-0.0819	-0.2214	0.4515	0.5164	1

RMSprop

The 10 best results for RMSprop in terms of last function value achieved are shown in figure 3.6. The PCC matrix of {*step size, delta, last function value achieved, convergence speed*} is shown in table 3.7. RMSprop proved to be unable to reach a satisfying final value within the maximum number of iterations. Moreover while increasing the step size is highly correlated with the increase in unstability and final value achieved, increasing the δ parameter is only loosely correlated to a better function value while the increase in unsability is once again more significant. The settings that perform best are those with low step size and moderate δ, but require an unsettling number of iterations.



Param	final loss	unstability	conv_speed
RMSprop{lr:0.004,d:0.8}	0.2901	246.6464	33.0233
RMSprop{lr:0.0002,d:0.9}	0.3136	0.0169	23.1552
RMSprop{lr:0.007,d:0.6}	0.3164	264.3215	39.6494
RMSprop{lr:0.0008,d:0.9}	0.3165	8.2972	18.4722
RMSprop{lr:0.004,d:0.6}	0.3202	110.5379	28.6363
RMSprop{lr:0.0008,d:0.8}	0.3264	5.7599	17.3872
RMSprop{lr:0.007,d:0.8}	0.3266	524.9894	40.0104
RMSprop{lr:0.007,d:0.5}	0.3288	92.3694	39.6684
RMSprop{lr:0.004,d:0.5}	0.3302	44.7894	25.9103
RMSprop{lr:0.004,d:0.9}	0.3335	258.3376	39.1874

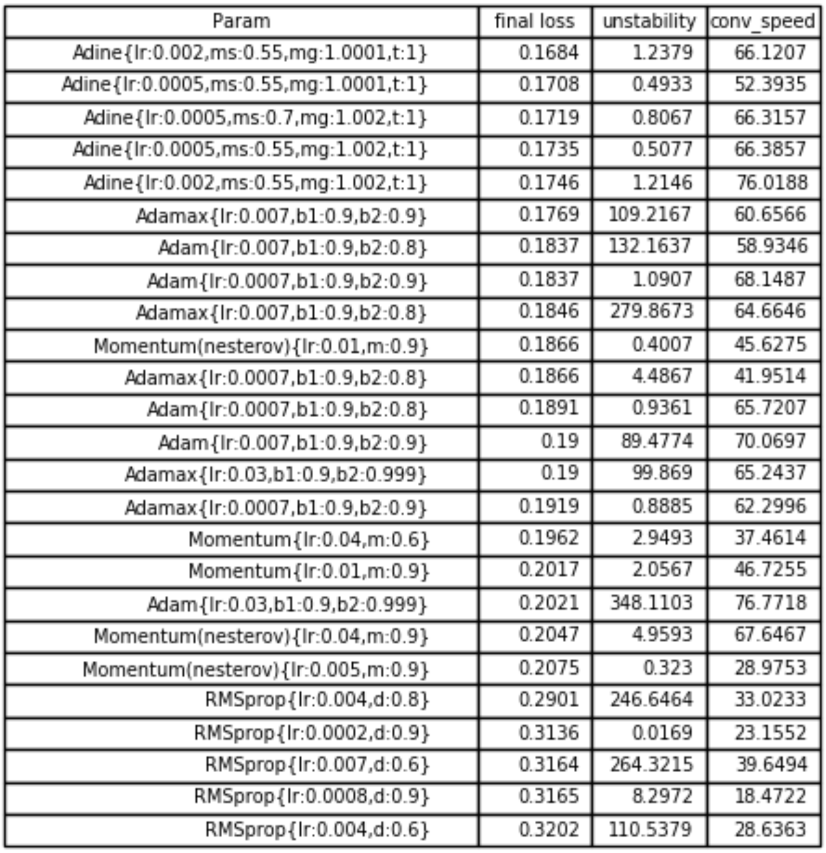
Figure 3.6: Table of the 10 best RMSprop configurations

Table 3.7: Parameters correlation of RMSprop

	η	δ	μ	*unst*	*cs*
η	1	0	0.9819	0.9436	0.6733
δ	0	1	-0.0731	0.1565	0.0328
μ	0.9819	-0.0731	1	0.9259	0.6003
unst	0.9436	0.1565	0.9259	1	0.5849
cs	0.6733	0.0328	0.6003	0.5849	1

3.3 Optimizers comparison

In table 3.7 we show the ranking among the 5 configuration with lowest final value of each optimizer. As we can see Adine comes out as a clear winner across the board, characterized by a great stability as the momentum' while also maintaining an impressive convergence speed in the same league as the more sophisticated Adam and Adamax methods. As we have seen earlier the best settings are the ones with a moderate momentum term, which might allow the algorithm to slow down quickly enough when heading towards a bad path, then accelerate back just as quickly thanks to the greater momentum term. Adam and Adamax are able to compete only at the cost of sacrificing stability with the configurations seen. Last and least, RMSprop disappoints on all fronts: recall that as discussed earlier the main issue appears to be its slowness which calls for a much higher number of iterations to compete.



Param	final loss	unstability	conv_speed
Adine{lr:0.002,ms:0.55,mg:1.0001,t:1}	0.1684	1.2379	66.1207
Adine{lr:0.0005,ms:0.55,mg:1.0001,t:1}	0.1708	0.4933	52.3935
Adine{lr:0.0005,ms:0.7,mg:1.002,t:1}	0.1719	0.8067	66.3157
Adine{lr:0.0005,ms:0.55,mg:1.002,t:1}	0.1735	0.5077	66.3857
Adine{lr:0.002,ms:0.55,mg:1.002,t:1}	0.1746	1.2146	76.0188
Adamax{lr:0.007,b1:0.9,b2:0.9}	0.1769	109.2167	60.6566
Adam{lr:0.007,b1:0.9,b2:0.8}	0.1837	132.1637	58.9346
Adam{lr:0.0007,b1:0.9,b2:0.9}	0.1837	1.0907	68.1487
Adamax{lr:0.007,b1:0.9,b2:0.8}	0.1846	279.8673	64.6646
Momentum(nesterov){lr:0.01,m:0.9}	0.1866	0.4007	45.6275
Adamax{lr:0.0007,b1:0.9,b2:0.8}	0.1866	4.4867	41.9514
Adam{lr:0.0007,b1:0.9,b2:0.8}	0.1891	0.9361	65.7207
Adam{lr:0.007,b1:0.9,b2:0.9}	0.19	89.4774	70.0697
Adamax{lr:0.03,b1:0.9,b2:0.999}	0.19	99.869	65.2437
Adamax{lr:0.0007,b1:0.9,b2:0.9}	0.1919	0.8885	62.2996
Momentum{lr:0.04,m:0.6}	0.1962	2.9493	37.4614
Momentum{lr:0.01,m:0.9}	0.2017	2.0567	46.7255
Adam{lr:0.03,b1:0.9,b2:0.999}	0.2021	348.1103	76.7718
Momentum(nesterov){lr:0.04,m:0.9}	0.2047	4.9593	67.6467
Momentum(nesterov){lr:0.005,m:0.9}	0.2075	0.323	28.9753
RMSprop{lr:0.004,d:0.8}	0.2901	246.6464	33.0233
RMSprop{lr:0.0002,d:0.9}	0.3136	0.0169	23.1552
RMSprop{lr:0.007,d:0.6}	0.3164	264.3215	39.6494
RMSprop{lr:0.0008,d:0.9}	0.3165	8.2972	18.4722
RMSprop{lr:0.004,d:0.6}	0.3202	110.5379	28.6363

Figure 3.7: Ranking of top 5 configurations of each optimizer.

3.4 Additional techniques comparison

[redacted]

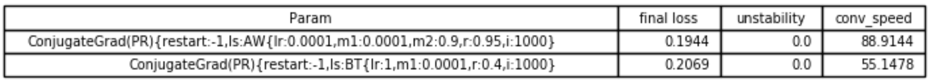
[redacted]

[redacted]

[redacted]

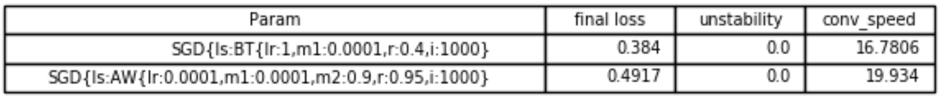
3.4.2 Conjugate gradient and line search

Additionally to the accelerated gradient methods we have also shortly investigated the behaviour of gradient descent with line search as well as that of conjugate gradient. In table 3.8, the results of conjugate gradient over $2 * 10^4$ iterations and 5 trials are shown: as to be expected Armijo-Wolfe line search outperforms backtracking both in term of final value reached as well as convergence speed. Moreover both configurations are completely stable which is again a known trait of line search. In table 3.9 we show the results of the basic version of gradient descent with the same line searches: while displaying the same stability as conjugate gradient, the final value achieved as well as the convergence speed are much worse.



Param	final loss	unstability	conv_speed
ConjugateGrad(PR){restart:-1,ls:AW{lr:0.0001,m1:0.0001,m2:0.9,r:0.95,i:1000}	0.1944	0.0	88.9144
ConjugateGrad(PR){restart:-1,ls:BT{lr:1,m1:0.0001,r:0.4,i:1000}	0.2069	0.0	55.1478

Figure 3.8: Table of PR version of conjugate gradient with Armijo-Wolfe and backtracking line searches.



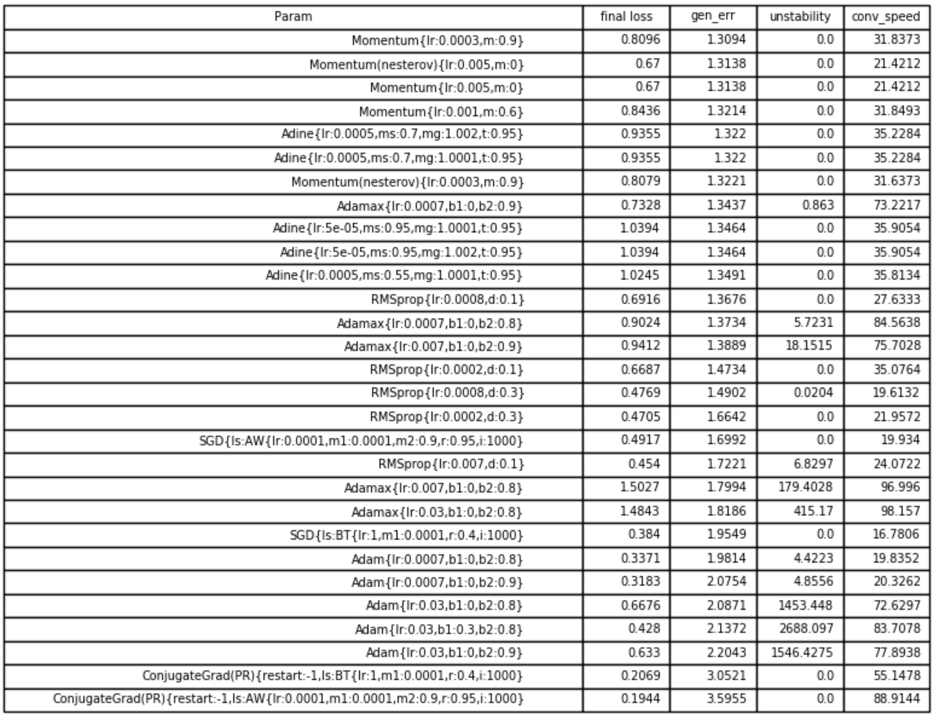
Param	final loss	unstability	conv_speed
SGD{ls:BT{lr:1,m1:0.0001,r:0.4,i:1000}	0.384	0.0	16.7806
SGD{ls:AW{lr:0.0001,m1:0.0001,m2:0.9,r:0.95,i:1000}	0.4917	0.0	19.934

Figure 3.9: Table of SGD with Armijo-Wolfe and backtracking line searches.

3.5 Generalization capability

We now briefly discuss and illustrate the generalization capabilities of our model. Keep in mind that here we employed a simplistic approach - that is randomly sampling the dataset and use a part of it (25% in our case) as a guideline to judge to what degree overfitting occurs. Machine learning theory provides much more solid techniques to search for the right model, such as cross-validation, nested cross-validation and bagging. Refer to the attached `MLreport.pdf` for a more rigorous approach including the exploration of different architectures, loss functions, regularization terms, activation functions and data preprocessing techniques.

In our task here most of the mentioned aspects were fixed beforehand and the only relevant remaining variable is given by the optimization algorithm. While this can still have an high impact on the generalization capabilities, remember that such an algorithm merely seeks for an hypothesis - that is a set of parameters θ ultimately defining our guess of the real function to be learned - in the given hypothesis space. Such a space however is pre-determined by the network architecture and activation functions employed, and should it not include hypotheses of a form resembling the one of the real function, no optimization algorithm would ever be able to produce a satisfactory result. A prime example of this is given by the linear least squares solvers: the linear hypothesis defined in this way lags behind the more ample one defined through our neural network shown in figure 1.3. Refer to table 3.10 for a comparison of the generalization capabilities obtained through the different methods. The 25 optimizers characterized by the lowest generalization error are shown. The generalization error of least squares solver, that we remind is solving a slightly different problem by using the L2 regularization, is 3.539: as we have seen this model is too simple for the problem to be solved and this is also confirmed by the generalization error which is a clear sign of underfitting when compared to the ones of the iterative methods. Generally speaking, none of the configurations that behaved best at the training level - that is solving our optimization problem, was also able to generalize well. This is common in machine learning and is a sign of overfitting: the model chosen is probably too powerful and able to fit the data by memorizing part of it without truly capturing the underlying distribution.



Param	final loss	gen_err	unstability	conv_speed
Momentum{lr:0.0003,m:0.9}	0.8096	1.3094	0.0	31.8373
Momentum(nesterov){lr:0.005,m:0}	0.67	1.3138	0.0	21.4212
Momentum{lr:0.005,m:0}	0.67	1.3138	0.0	21.4212
Momentum{lr:0.001,m:0.6}	0.8436	1.3214	0.0	31.8493
Adine{lr:0.0005,ms:0.7,mg:1.002,t:0.95}	0.9355	1.322	0.0	35.2284
Adine{lr:0.0005,ms:0.7,mg:1.0001,t:0.95}	0.9355	1.322	0.0	35.2284
Momentum(nesterov){lr:0.0003,m:0.9}	0.8079	1.3221	0.0	31.6373
Adamax{lr:0.0007,b1:0,b2:0.9}	0.7328	1.3437	0.863	73.2217
Adine{lr:5e-05,ms:0.95,mg:1.0001,t:0.95}	1.0394	1.3464	0.0	35.9054
Adine{lr:5e-05,ms:0.95,mg:1.002,t:0.95}	1.0394	1.3464	0.0	35.9054
Adine{lr:0.0005,ms:0.55,mg:1.0001,t:0.95}	1.0245	1.3491	0.0	35.8134
RMSprop{lr:0.0008,d:0.1}	0.6916	1.3676	0.0	27.6333
Adamax{lr:0.0007,b1:0,b2:0.8}	0.9024	1.3734	5.7231	84.5638
Adamax{lr:0.007,b1:0,b2:0.9}	0.9412	1.3889	18.1515	75.7028
RMSprop{lr:0.0002,d:0.1}	0.6687	1.4734	0.0	35.0764
RMSprop{lr:0.0008,d:0.3}	0.4769	1.4902	0.0204	19.6132
RMSprop{lr:0.0002,d:0.3}	0.4705	1.6642	0.0	21.9572
SGD{ls:AW{lr:0.0001,m1:0.0001,m2:0.9,r:0.95,i:1000}	0.4917	1.6992	0.0	19.934
RMSprop{lr:0.007,d:0.1}	0.454	1.7221	6.8297	24.0722
Adamax{lr:0.007,b1:0,b2:0.8}	1.5027	1.7994	179.4028	96.996
Adamax{lr:0.03,b1:0,b2:0.8}	1.4843	1.8186	415.17	98.157
SGD{ls:BT{lr:1,m1:0.0001,r:0.4,i:1000}	0.384	1.9549	0.0	16.7806
Adam{lr:0.0007,b1:0,b2:0.8}	0.3371	1.9814	4.4223	19.8352
Adam{lr:0.0007,b1:0,b2:0.9}	0.3183	2.0754	4.8556	20.3262
Adam{lr:0.03,b1:0,b2:0.8}	0.6676	2.0871	1453.448	72.6297
Adam{lr:0.03,b1:0.3,b2:0.8}	0.428	2.1372	2688.097	83.7078
Adam{lr:0.03,b1:0,b2:0.9}	0.633	2.2043	1546.4275	77.8938
ConjugateGrad(PR){restart:-1,ls:BT{lr:1,m1:0.0001,r:0.4,i:1000}	0.2069	3.0521	0.0	55.1478
ConjugateGrad(PR){restart:-1,ls:AW{lr:0.0001,m1:0.0001,m2:0.9,r:0.95,i:1000}	0.1944	3.5955	0.0	88.9144

Figure 3.10: Ranking of the best 25 optimizers based on generalization error.

Chapter 4

Conclusions

In this report we documented our brief journey in the optimization world, beyond gradient descent. In particular our focus was on the class of accelerated gradient methods, that have found great success in machine learning. We investigated a variety of flavours in which such methods can be tweaked and modified, ranging from a simple momentum term, to the adaptive momentum of adine, to more complex approaches such as adam, adamax and RMSprop. By studying the behaviour of these methods on a specific machine learning problem, we soon noticed that the two different goals of optimization and machine learning cannot share a totally joint approach.

The configurations that performed best on our training data driven problem, proved to generalize poorly thus confirming once again that the closer we get to the optimum of the problem we specify most likely does not correspond to a good solution of the problem we are really trying to solve from a machine learning perspective. We have also extended our attention to a different set of techniques, ranging from different or extensions to the iterative approaches such as conjugate gradient and line search, [redacted]

[redacted]

Several additions can be made to the work presented here: for instance it would be interesting to investigate the behaviour of the somewhat unpopular, in machine learning, second order methods, or their approximation, and how they compare to the accelerated gradient ones.

Bibliography

(1) B. Polyak, "Some methods of speeding up the convergence of iteration methods", 1964, **4**, 1–17.

(2) S. Becker, Y. Le Cun et al., "Improving the convergence of back-propagation learning with second order methods", 1988, 29–37.

(3) A. Choromanska, M. Henaff, M. Mathieu, G. B. Arous and Y. LeCun, "The loss surfaces of multilayer networks", 2015, 192–204.

(4) I. Goodfellow, Y. Bengio and A. Courville, *Deep Learning*, `http://www.deeplearningbook.org`, MIT Press, 2016.

(5) K. P. Bennett and E. Parrado-Hernández, "The interplay of optimization and machine learning research", *Journal of Machine Learning Research*, 2006, **7**, 1265–1281.

(6) M. G. Lagoudakis, "Neural Networks and Optimization Problems".

(7) V. N. Vapnik and V. Vapnik, *Statistical learning theory*, Wiley New York, 1998, vol. 1.

(8) H. Zou and T. Hastie, "Regularization and variable selection via the elastic net", *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 2005, **67**, 301–320.

(9) D. E. Rumelhart, G. E. Hinton and R. J. Williams, "Learning internal representations by error propagation", 1985.

(10) E. Hallström, *Backpropagation from the beginning*, `https://medium.com/@erikhallstrm/backpropagation-from-the-beginning-77356edf427d`, 2016.

(11) S. Boyd and A. Mutapcic, "Subgradient methods", *Lecture notes of EE364b, Stanford University, Winter Quarter*, 2006, **2007**.

(12) R. A. Jacobs, “Increased rates of convergence through learning rate adaptation”, *Neural Networks*, 1988, **1**, 295–307.

(13) J. J. Moré and D. J. Thuente, “Line Search Algorithms with Guaranteed Sufficient Decrease”, *ACM Trans. Math. Softw.*, 1994, **20**, 286–307.

(14) Z. Wang and A. C. Bovik, “Mean squared error: Love it or leave it? A new look at signal fidelity measures”, *IEEE signal processing magazine*, 2009, **26**, 98–117.

(15) S. Ghadimi and G. Lan, “Accelerated gradient methods for nonconvex nonlinear and stochastic programming”, *Mathematical Programming*, 2016, **156**, 59–99.

(16) M. Malisoff and F. Mazenc, *Review of Lyapunov Functions*, 2009.

(17) N. Qian, “On the momentum term in gradient descent learning algorithms”, *Neural Networks*, 1999, **12**, 145–151.

(18) A. Frangioni, *Unconstrained optimization II: More-than-gradient methods*, University Lecture, 2017.

(19) E. Ghadimi, H. R. Feyzmahdavian and M. Johansson, “Global convergence of the Heavy-ball method for convex optimization”, 2015, 310–315.

(20) P. Ochs, Y. Chen, T. Brox and T. Pock, “iPiano: Inertial proximal algorithm for nonconvex optimization”, *SIAM Journal on Imaging Sciences*, 2014, **7**, 1388–1419.

(21) K. Kurdyka, “On gradients of functions definable in o-minimal structures”, 1998, **48**, 769–784.

(22) T. Yang, Q. Lin and Z. Li, “Unified convergence analysis of stochastic momentum methods for convex and non-convex optimization”, *arXiv preprint arXiv:1604.03257*, 2016.

(23) I. Sutskever, J. Martens, G. Dahl and G. Hinton, “On the importance of initialization and momentum in deep learning”, 2013, 1139–1147.

(24) J. Martens, “Deep learning via Hessian-free optimization.”, 2010, **27**, 735–742.

(25) D. Kingma and J. Ba, “Adam: A method for stochastic optimization”, *arXiv preprint arXiv:1412.6980*, 2014.

(26) A. C. Wilson, R. Roelofs, M. Stern, N. Srebro and B. Recht, "The Marginal Value of Adaptive Gradient Methods in Machine Learning", *arXiv preprint arXiv:1705.08292*, 2017.

(27) N. S. Keskar and R. Socher, "Improving Generalization Performance by Switching from Adam to SGD", *arXiv preprint arXiv:1712.07628*, 2017.

(28) M. Riedmiller and H. Braun, "RPROP-A fast adaptive learning algorithm", 1992.

(29) T. Tieleman and G. Hinton, "Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude", *COURSERA: Neural networks for machine learning*, 2012, **4**, 26–31.

(30) S. A. Bekessy, M. White, A. Gordon, A. Moilanen, M. A. Mccarthy and B. A. Wintle, "Transparent planning for biodiversity and development in the urban fringe", *Landscape and Urban Planning*, 2012, **108**, 140–149.

(31) Y. Dauphin, R. Pascanu, Ç. Gülçehre, K. Cho, S. Ganguli and Y. Bengio, "Identifying and attacking the saddle point problem in high-dimensional non-convex optimization", *CoRR*, 2014, **abs/1406.2572**.

(32) R. Hauser, *Line Search Methods for Unconstrained Optimisation*, University Lecture, 2007.

(33) L. Armijo, "Minimization of functions having Lipschitz continuous first partial derivatives", *Pacific Journal of mathematics*, 1966, **16**, 1–3.

(34) P. Wolfe, "Convergence conditions for ascent methods", *SIAM review*, 1969, **11**, 226–235.

(35) M. Mahsereci and P. Hennig, "Probabilistic line searches for stochastic optimization", 2015, 181–189.

(36) M. Hardt, B. Recht and Y. Singer, "Train faster, generalize better: Stability of stochastic gradient descent", *arXiv preprint arXiv:1509.01240*, 2015.

(37) A. Frangioni, *Unconstrained optimization II: Conjugate gradient methods*, University Lecture, 2017.

(38) J. R. Shewchuk et al., *An introduction to the conjugate gradient method without the agonizing pain*, 1994.

(39) S. Boyd and L. Vandenberghe, *Introduction to Applied Linear Algebra - Vectors, Matrices, and Least Squares*, Cambridge University Press, to be published(2018), 2017, vol. 1.

(40) R. Bridson, *CS 542G: Least Squares, Normal Equations*, University Lecture, 2008.

(41) Ä. Björck, "Numerics of Gram-Schmidt orthogonalization", *Linear Algebra and its Applications*, 1994, **197-198**, 297–316.

(42) F. Poloni, *QR factorization: Householder reflectors*, University Lecture, 2017.

(43) F. Poloni, *Stability of LS problems: Comparison of least squares algorithms*, University Lecture, 2017.

(44) F. Chollet et al., *Keras*, `https://github.com/keras-team/keras`, 2015.