Università degli Studi di Padova

Dipartimento di Ingegneria dell'Informazione

University of Padua
Department of Information Engineering

Master's Degree in Computer Engineering

Metaheuristic frameworks for deep neural network training

Master's Thesis

Supervisor:
Prof. Matteo Fischetti

Candidate:
Marco Calandro
1206155

Padua, 1 December 2020
Academic Year: 2019/2020

Contents

List of Algorithms

Abstract

In our work, we have proposed new techniques for the training of Convolutional Neural Network (CNN) based on the embedding of Stochastic Gradient Descent (SGD) within metaheuristic frameworks; in particuar, Simulated Annealing (SA) and Variable Neighborhood Search (VNS). Their performances have been tested by training a deep neural network in solving the image classification problem. Afterwards, the results were compared with the ones of state-of-art techniques. Moreover, an in-depth analysis highlighted the most effective features and the main flaws of the examined metaheuristics in the context of neural network training. The test results have shown that the application of metaheuristics in general improves the generalizability of the solutions. In terms of performances, SA-based algorithms typically performed slightly worse than state-of-art techniques; VNS-based ones, instead, have demonstrated their potential in outperforming state-of-art techniques.

1. Introduction

The purpose of this work is to design, analyze, and test new approaches to the Neural Network Training Problem (NNTP) based on metaheuristic strategies. Unfortunately, this problem is too wide to be fully analyzed here since its formulation changes significantly due to different learning definitions, network structures, and dataset types. The analysis is then limited to the supervised definition of learning, applied to the image classification problem performed by convolutional neural networks (CNN). This choice is supported by the amount of data available for this version of the problem and by the suitability for the application of metaheuristic elements.

The most popular state-of-art algorithms used to solve the NNTP are based on Gradient Descent (GD): a local search algorithm that iteratively moves the network in the direction of the maximum decrease of the cost function. These algorithms introduce sophisticated mechanisms to enhance the effectiveness of the GD basic procedure with the underlying purpose of converging to the closest local optimum, whereas standard optimization algorithms try to avoid getting stuck in local minima and search for the global optimum instead. This unconventional behavior can be justified by the difficulty of finding global optima derived from the curse of dimensionality that affects the solution space.

Metaheuristics are problem-independent frameworks designed to approach generic optimization problems, based on the concepts of intensification and diversification: respectively, move the solution toward a local minimum to improve its cost and move the solution toward unexplored sectors of the solution space to search for the global optimum. From this point of view, the state-of-art algorithms for the NNTP implement intensification only, whereas metaheuristics balance these two features to obtain better performance.

The new optimization algorithms are designed to exploit both approaches, hence their design can be intuitively described as state-of-art algorithms with diversification and intensification techniques. The resulting algorithms have a metaheuristic structure, while the majority of the components not explicitly defined by the framework are embeddings of state-of-art algorithms or strongly inspired by them.

The performance of the newly introduced algorithms compared to the state-of-art ones depends on many factors: the relationship between intensification and diversification, the effectiveness of diversification in reducing the overfitting, and the role of the curse of dimensionality. The algorithms are tested on multiple datasets and multiple networks to evaluate the impact of these factors: the purpose is to tell if the overhead introduced to implement the metaheuristics structure and the diversification leads to significant results or if the problem itself is not suitable to be tackled with strategies different from local searches.

2. State-of-art training techniques

The most popular optimization algorithms for neural network training implement Gradient Descent (GD) with additional mechanisms to improve its performance. The idea of gradient descent is to perform downhill steps, with respect to the loss function, until a minimum is found: this process leads to a global minimum for convex surfaces and to a local minimum for non-convex surfaces. The downhill direction is the anti-gradient of the loss function with respect to the weights, which indicates the direction of the maximum decreasing of the loss function. For simplicity, consider the training process as a sequence of moves and introduce a subscript to parameters to indicate explicitly the move they refer to. Denoting with θ the weights of the network, with ∇L the gradient of the loss function L, and with η the learning rate, we can define the update rule for the weights as follows.

$$\theta_{t+1} = \theta_t - \eta \; \nabla L(\theta_t) \tag{2.1}$$

Although the resulting rule is simple and quite intuitive, it clearly highlights two problems: the efficient computation of the gradient and the choice of the learning rate.

2.1. Common strategies

The gradient is a key element in all state-of-art techniques: the way it is exploited in combination with the learning rate and, eventually, with advanced features distinguishes an optimizer from another. However, the intermediate steps in the computation of the gradient do not change the results of the algorithms since only the final result has a role in the updating rule. On the other hand, the specific procedure chosen influences how the dataset is analyzed and how the training is performed. The consequence of these observations is that different optimizers share the most efficient strategy to compute the gradient and the resulting management of the training process accordingly.

2.1.1. Gradient computation

An exact computation of the gradient requires an analysis of the whole training set, resulting in a very complex procedure that is unfeasible for most of the problems; such implementation of GD is called batch GD. A stochastic approach, called stochastic GD, consists in estimating the real gradient on expectation: this can be easily done by computing the gradient using a single sample taken uniformly at random from the training set. The procedure has very low complexity but, on the other hand, provides very noisy gradients. A compromise between these two approaches consists in computing the gradient on a mini-batch, a set of elements sampled from the training set. Although this is still a stochastic approach, it is typically named “mini-batch GD”, even though mini-batch stochastic GD would be more appropriate. If the sampling is performed appropriately, the result is still equal to the gradient on expectation; moreover, the larger number of samples reduces the noise and leads to more accurate results with a feasible impact on complexity. The use of mini-batches implies the introduction of a hyper-parameter to indicate the number of samples in each mini-batch: its value depends on the situation but is typically in the range 50$\sim$500. Since this third solution is more efficient than the first one and less unstable than the second one, this procedure has become a standard part of all the most common techniques.

2.1.2. Epochs and mini-batches management

From the optimization algorithms point of view, the training process is a sequence of updates to the weights of the network, each one depending on the gradient and few technique-specific parameters, both obtained by the analysis of a mini-batch. Besides, some algorithms assume that the mentioned sequence of updates is divided into epochs to perform a periodic hyper-parameter update. As a consequence, the training is divided into epochs: each epoch analyses the whole training set by dividing it into mini-batches and analyzing them one by one. The process can be described effectively with two loops: the outer loop counts the epochs while the inner loop iterates over the mini-batches obtained from the training set. Observe that the mini-batches must be sampled from the training set hence it is unlikely to have the same set of mini-batches in different epochs.

2.2. Stochastic Gradient Descent (SGD)

In the literature and machine learning libraries, the stochastic gradient descent (SGD) optimizer is an implementation of the GD algorithm with the mini-batch technique as described above. In its simplest version, the updating rule used is the one presented in Equation 2.1 and the learning rate is constant during the whole execution. Although more sophisticated versions introduce additional elements in the update rule, as SGD with momentum, or use a finer control on the learning rate, as scheduled SGD, the main structure of these algorithms is the same as SGD.

The standard structure of a training algorithm based on SGD is described with pseudocode in Algorithm 1.

2.2.1. SGD with momentum

In vanilla SGD, each move depends only on the current status of the network and the current mini-batch analyzed, ignoring the information carried by previous moves. Although the management of the full history of the moves executed would introduce a huge overhead, storing some essential information may be useful to improve the convergence. An effective realization of this idea requires storing only cumulative information about

Algorithm 1: Stochastic gradient descent pseudocode.

```
Parameters: $\theta$: network weights, $D$: training set, $L$: loss function, $E$: number
            of epochs, $\eta$: learning rate.
Function SGD_training($\theta$, $D$)
    $t \leftarrow 0$
    $\theta_t \leftarrow \theta$
    foreach epoch $e \in \{0, \ldots, E-1\}$ do
        foreach mini-batch $b \in D$ do
            $\nabla L(\theta_t) \leftarrow$ compute_gradient($\theta_t$, $b$, $L$)
            $\theta_{t+1} \leftarrow \theta_t - \eta \nabla L(\theta_t)$
            $t \leftarrow t + 1$
    return $\theta_t$
```

the direction of previous moves in an additional variable called momentum. The momentum appears then as an additional term in the update rule with the role of enhancing the movements in the directions followed by the previous moves, roughly implementing a notion of inertia.

Classic momentum

The classic momentum implementation stores only the last move, requiring a single additional variable v, which is then used to introduce a new term in the update rule, whose impact can be controlled with an additional hyper-parameter μ. The update rule must be extended to update both the weights θ and the momentum v, as shown in the equation below.

$$\begin{aligned} v_{t+1} &= \theta_{t+1} - \theta_t \\ \theta_{t+1} &= \theta_t + \mu v_t - \eta \nabla L(\theta_t) \end{aligned} \tag{2.2}$$

This rule formulation is simple to understand but for it to be implemented efficiently some minor modification must be applied, as described in Algorithm 2.

Nesterov Accelerated Gradient (NAG)

A more advanced technique that exploits the momentum is Nesterov Accelerated Gradient (NAG). Whereas the classic momentum exploits only the previous moves history, the NAG also looks forward to the consequences of the next move to anticipate the corrections. The procedure followed by the update is conceptually divided into three steps: firstly, the parameters move to an intermediate point by applying the momentum, secondly, the gradient is computed on the intermediate point, finally, the gradient

Algorithm 2: SGD with momentum pseudocode.

```
Parameters: $\theta$: network weights, $D$: training set, $L$: loss function, $E$: number
            of epochs, $v$: momentum, $\eta$: learning rate, $\mu$: momentum
            coefficient.
Function SGD_with_momentum_training($\theta$, $D$)
    $t \leftarrow 0$
    $\theta_t \leftarrow \theta$
    $v_t \leftarrow 0$
    foreach epoch $e \in \{0, \dots, E-1\}$ do
        foreach mini-batch $b \in D$ do
            $\nabla L(\theta_t) \leftarrow$ compute_gradient($\theta_t$, $b$, $L$)
            $v_{t+1} \leftarrow \mu v_t - \eta \nabla L(\theta_t)$
            $\theta_{t+1} \leftarrow \theta_t + v_{t+1}$
            $t \leftarrow t + 1$
    return $\theta_t$
```

is used to move the parameters to the final position. The update rule can be formulated as follows:

$$\begin{aligned} v_{t+1} &= \theta_{t+1} - \theta_t \\ \theta_{t+1} &= \theta_t + \mu v_t - \eta \nabla L(\theta_t + \mu v_t) \ . \end{aligned} \tag{2.3}$$

Although is not strictly required by the formulation, the implementation can be simplified by introducing a variable y to explicitly store the value of the intermediate position

$$y_{t+1} = \theta_t + \mu v_t \ . \tag{2.4}$$

The relation between these parameters and their updates is described in Figure 2.1.

Moreover, the presented formulation indicates that the gradient is computed within the optimization move, whereas machine libraries typically assume that the gradient is computed before the optimization move. In particular, the move resulting from the update rule follows the steps $\theta_t \rightarrow y_{t+1} \rightarrow \theta_{t+1}$, where the gradient is computed on the middle step. An alternative, that keeps unaltered the overall sequence of moves, consists of following the steps $y_t \rightarrow \theta_t \rightarrow y_{t+1}$, with the advantage that the gradient is computed on the first step; however, this implies that the y variable refers to the weights of the network, while θ becomes a utility variable. The pseudocode that considers these aspects is provided in Algorithm 3.

The techniques presented till now (SGD, SGD with momentum, and NAG) exploit the gradient and the momentum to explore effectively the solution space; although the

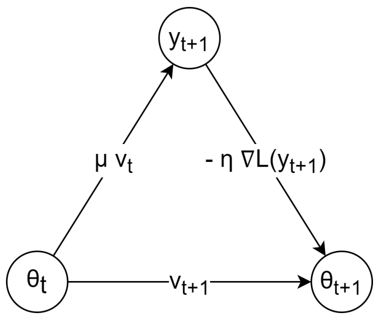


Figure 2.1.: Relationship between θ, y and v parameters over time.

update rules of the resulting strategies are all similar, their performance and behavior present significant differences. The example reported in Figure 2.2 shows how the different techniques compute the sequence of moves when the parameters are in a ravine area, where the loss function decreases much more steeply in a dimension as compared with the others. SGD is improved with the use of momentum, obtaining a faster convergence;

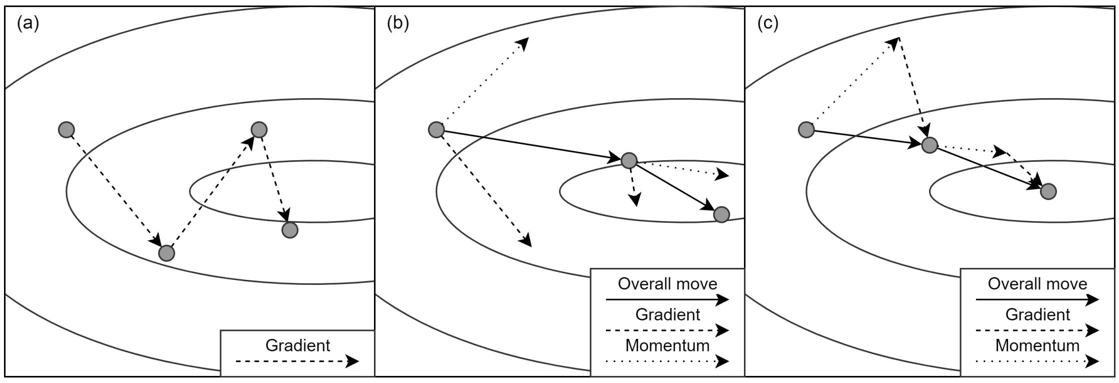


Figure 2.2.: Comparison of update rules for (a) SGD, (b) SGD with momentum, and (c) NAG.

then it is further improved with the look-ahead of the gradient, correcting the moves before they are performed: the obtained technique, NAG, is the best performer among the three.

2.2.2. Scheduled SGD (SSGD)

The Scheduled SGD (SSGD) is an improvement of SGD that, differently from momentum variants, does not modify the update rule of SGD, but rather introduces a more sophisticated mechanism to control the learning rate. The learning rate and the gradient influence directly how far the new solution will be from the previous one, since the difference between them is $\Delta\theta = -\eta\ \nabla L(\theta_t)$. As a consequence, the learning rate

```
Algorithm 3: Nesterov accelerated gradient pseudocode.
  Parameters: $y$: network weights, $D$: training set, $L$: loss function, $E$: number
              of epochs, $\theta$: intermediate weights, $v$: momentum, $\eta$: learning
              rate, $\mu$: momentum coefficient.
  Function NAG_training($y$, $D$)
    $t \leftarrow 0$
    $y_{t+1} \leftarrow y$
    $\theta_t \leftarrow y_{t+1}$
    $v_t \leftarrow 0$
    foreach epoch $e \in \{0, \dots, E-1\}$ do
      foreach mini-batch $b \in D$ do
        $\nabla L(y_{t+1}) \leftarrow$ compute_gradient($y_{t+1}$, $b$, $L$)
        $\theta_{t+1} \leftarrow y_{t+1} - \eta \nabla L(y_{t+1})$
        $v_{t+1} \leftarrow \mu v_t - \eta \nabla L(y_{t+1})$
        $y_{t+2} \leftarrow \theta_{t+1} + \mu v_{t+1}$
        $t \leftarrow t + 1$
    return $y_{t+1}$
```

can be tuned to travel large distances obtaining a faster convergence, but making it harder to get close to the local optimum, or to travel small distances resulting in a slow convergence, but with the opportunity of getting close to the local optimum.

The SSGD algorithm exploits large learning rates at the beginning of the training to reach the neighborhood of the local optimum, whereas later it exploits small learning rates to get as close as possible to the local optimum. The learning rate is managed by an additional element, the scheduler, that implements an annealing procedure to gradually decrease the learning rate during the training. Although it is possible to design different annealing procedures, in the present work only the multi-step strategy will be taken into consideration. The pseudocode for SSGD, independent from the implementation of the scheduler, is provided in Algorithm 4.

The multi-step scheduler management of the learning rate can be mathematically expressed as a linear combination of step functions. The learning rate is not decreased gradually: it is kept constant for most of the training and updated only in specific epochs that performs a step operation, which decreases significantly the learning rate. A simple example of the learning rate curve and how it influences the solution space exploration is provided in Figure 2.3.

A generic implementation of the multi-step scheduler requires two lists: the list of

Algorithm 4: Scheduled SGD pseudocode.

```
Parameters: $\theta$: network weights, $D$: training set, $L$: loss function, $E$: number
            of epochs, $\eta$: learning rate.

Function SSGD_training($\theta$, $D$)
    $t \leftarrow 0$
    $\theta_t \leftarrow \theta$
    foreach epoch $e \in \{0, \dots, E-1\}$ do
        $\eta \leftarrow$ scheduler($e$)
        foreach mini-batch $b \in D$ do
            $\nabla L(\theta_t) \leftarrow$ compute_gradient($\theta_t$, $b$, $L$)
            $\theta_{t+1} \leftarrow \theta_t - \eta \nabla L(\theta_t)$
            $t \leftarrow t + 1$
    return $\theta_t$
```

epochs at which steps are performed and the list of corresponding learning rates. However, assuming that each learning rate should be used for the same number of epochs and that the list of learning rates is a geometric progression, it is possible to define a multi-step procedure with only three constants: the number of epochs per learning rate, the initial learning rate and the common ratio of the progression. Under these reasonable assumptions, the resulting scheduler is very simple and its pseudocode is presented in Algorithm 5.

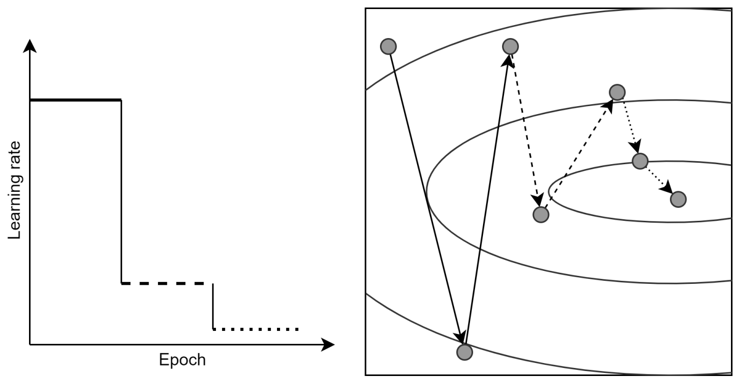


Figure 2.3.: Example of learning rate curve and solution space exploration for SSGD algorithms.

Algorithm 5: Multi-step scheduler pseudocode.

```
Parameters: $e$: epoch, $\eta_0$: initial learning rate, $s$: epochs per learning rate, $d$:
            learning rate decay.
Function multistep_scheduler($e$)
    $n \leftarrow \lfloor e \ / \ s \rfloor$
    $\eta \leftarrow \eta_0 \cdot d^{\,n}$
    return $\eta$
```

2.3. Adaptive Moment Estimation (ADAM)

Adaptive Moment Estimation (ADAM) is a technique based on the application of the estimation of mean and variance of the gradient to the basic update rule presented in Equation 2.1. Gradient's first and second moment estimations are exploited to tune the learning rate of each parameter of the network individually. Moreover, the mean estimation replaces the actual gradient in the update rule. Although the learning rate is still a hyper-parameter, it is auto-tuned by the algorithm, hence its initial value has little importance and can be left at a default value. Moreover, the default values of the constants, that will be introduced for the estimation and for numerical stability, lead to good performance and, since their correlation with the results is not as obvious as for the learning rate, tuning these hyper-parameters is likely to result in overfitting. Hence, ADAM can be used without the need for tuning any hyper-parameter, since - although they are present in the algorithm - their impact is auto-corrected or already leads to good performance.

The estimations are performed by storing the exponentially decaying averages of the past gradients and squared gradients, as described by the following equations.

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1-\beta_1)\nabla L(\theta_t) \\ v_t &= \beta_2 v_{t-1} + (1-\beta_2)\nabla L^2(\theta_t) \end{aligned} \tag{2.5}$$

The moment estimations are initialized as 0's vectors, introducing a bias toward zero, especially in the early stages of the algorithm. This effect is compensated by computing bias-corrected estimations as follows.

$$\begin{aligned} \widehat{m_t} &= \frac{m_t}{1-\beta_1^t} \\ \widehat{v_t} &= \frac{v_t}{1-\beta_2^t} \end{aligned} \tag{2.6}$$

The values suggested by the authors for the constants are $\beta_1 = 0.9$ and $\beta_2 = 0.999$ [2].

As anticipated, each move begins with the computation of the estimations as their values are used to update the weight of the network: the variance estimation tunes the learning rate while the mean estimation indicates the direction of the move. The update rule follows.

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\widehat{v_t}} + \epsilon}\,\widehat{m_t} \tag{2.7}$$

The equation introduces the smoothing term ϵ to avoid the division by zero and increase the numerical stability of the updating rule; typically this constant is in the order of 10^{-8}.

The operations performed to estimate moments and to perform the update rule are numerous compared with other optimization algorithms, but also element-wise. Hence, for time-constrained applications, they can be parallelized to improve the time complexity. The pseudocode for ADAM is provided in Algorithm 6 for completeness since it follows closely the steps described in Equations 2.5, 2.6, and 2.7.

Algorithm 6: Adaptive moment estimation pseudocode.

```
Parameters: $\theta$: network weights, $D$: training set, $L$: loss function, $E$: number
            of epochs, $m$: mean estimation, $v$: variance estimation, $\eta$:
            learning rate, $\beta_1, \beta_2, \epsilon$: constants.

Function ADAM_training($\theta$, $D$)
    $t \leftarrow 0$
    $m_t,\ v_t \leftarrow 0$
    $\theta_t \leftarrow \theta$
    foreach epoch $e \in \{0, \dots, E-1\}$ do
        foreach mini-batch $b \in D$ do
            $\nabla L(\theta_t) \leftarrow$ compute_gradient($\theta_t$, $b$, $L$)
            $m_t \leftarrow \beta_1\, m_{t-1} + (1-\beta_1)\, \nabla L(\theta_t)$
            $v_t \leftarrow \beta_2\, v_{t-1} + (1-\beta_2)\, \nabla L^2(\theta_t)$
            $\widehat{m_t} \leftarrow m_t/(1-\beta_1^t)$
            $\widehat{v_t} \leftarrow v_t/(1-\beta_2^t)$
            $\widehat{\eta} \leftarrow \eta/(\sqrt{\widehat{v_t}} + \epsilon)$
            $\theta_{t+1} \leftarrow \theta_t - \widehat{\eta}\, \widehat{m_t}$
            $t \leftarrow t+1$
    return $\theta_t$
```

3. Metaheuristics

The definition of metaheuristic used in this work is provided by Sörensen and Glover in [3]: the following quotation, extracted from their paper, describes shortly and effectively the concept of metaheuristic applied both to algorithms and to frameworks.

> A metaheuristic is a high-level problem-independent algorithmic framework that provides a set of guidelines or strategies to develop heuristic optimization algorithms. The term is also used to refer to a problem-specific implementation of a heuristic optimization algorithm according to the guidelines expressed in such a framework.

The definition does not provide indications on how the frameworks should be designed, hence many different formulations of metaheuristics were published. On the other hand, the problem-independence condition is strong and forces the frameworks to use concepts that can be applied to a generic optimization problem. For this reason, metaheuristics perform the optimization through the exploration of the solution space of the problem; since such set is required to define any optimization problem, its existence is guaranteed.

The solution space exploration purpose is to find a global optimum of the problem. However, this task implies two different sub-tasks: the first one, called diversification, consists in exploring a large part of the solution space, and derives from the "global" property of the desired solution; the second one, called intensification, consists in searching for local optima and derives from the "optimum" property of the desired solution. Observe that intensification alone is likely to produce local optima, while diversification alone is likely to produce non-optimal solutions, consequently the aim of each metaheuristic is to balance these two approaches to maximize their effectiveness.

Although it is not specified in the definition of a generic optimization problem, metaheuristics assume a basic regularity of the solution space to introduce additional concepts, such as the distance between solutions and the neighborhood of a solution. The distance between solutions is a measure of their differences and, under the assumption that close solutions have similar costs, it is meaningful to define the neighborhood of a solution as the set of solutions that are within a fixed distance from it. Metaheuristics explore of the solution space exploiting these concepts. The overall strategy is shared between them and can be described as follows: the algorithm starts with an initial solution, then a sequence of moves is performed until a stopping condition is satisfied. Each move consists in replacing the current solution with a new one, that can be taken from its neighborhood, sampled at a fixed distance from it or computed following a framework-specific strategy.

Metaheuristic frameworks are, by definition, problem-independent but metaheuristic algorithms are not. The gap between a problem-independent description and a problem-specific algorithm requires an interface that maps the framework's concepts to specific definitions, compliant with the problem characteristics. The implementation of this interface influences significantly the performance of the resulting algorithm, besides, an optimal interface for a specific framework may be not optimal for another one.

The metaheuristics applied to the neural network training problem were chosen according to their suitability with the problem itself. The main peculiarity of the problem is the size of the solution space: the huge number of dimensions imposes tight bounds on both time and space complexities. Hence, any feasible framework should introduce a small overhead and store only minimal information on the solutions. These requirements are satisfied by simulated annealing and variable neighborhood search.

3.1. Simulated Annealing (SA)

Simulated annealing (SA) [4] is a metaheuristic framework inspired by the ability of physical systems to reach the state of minimal energy during a cooling process. The search for the minimum is modeled as a system that is cooling: the exploration of the solution space is divided into a sequence of stages, influenced by a temperature parameter that gradually decreases.

An implementation of SA requires two global variables to save the state of the system: one to track the solution currently under examination and the other to store the current temperature of the system. A stage explores the solution space by sampling the neighborhood of the current solution a fixed number of times. Each sampled solution has a chance to become the current solution: if its cost is good enough, it is accepted and becomes the current solution; otherwise, it is rejected and simply ignored. More in detail, the acceptance test takes into account the current temperature of the stage and the cost difference between the sampled solution and the current solution. Improving solutions are always accepted while worsening solutions have only a certain probability to be accepted. This probability is lower for worse solutions and decreases as the temperature decreases, implying that accepting worsening solutions gets less likely as the algorithm proceeds. The possibility of accepting a worsening solution is a key element of SA since it allows escaping local minima.

The behavior of the algorithm in each stage changes during the computation: when the temperature is high, worsening moves can easily be accepted, hence the solution space can be explored widely (diversification). On the contrary, when the temperature is low, almost only improving solutions are accepted (intensification). These observations lead to the conclusion that, at first, the exploration is performed mainly with a diversification approach but, as the stages proceed, the dominant approach becomes the intensification.

Both intensification and diversification are present in all stages, but their impact on the algorithm changes according to the current temperature.

The pseudocode for simulated annealing is reported in Algorithm 7. Observe that many elements used are not defined within the framework, and are left to the developer: the neighborhood and the cost function are problem-dependent elements, which could not be specified, while the initial temperature, the termination condition, the value of k, and the updating rule for the temperature are problem-independent elements that can be seen as degrees of freedom to customize the implementation of SA. All these elements have a significant impact on the performance of the algorithm, hence they must be set carefully.

Algorithm 7: Simulated annealing pseudocode.

```
Parameters: $S_0$: initial solution, $T$: temperature.
Function SA_framework($S_0$)
    $S \leftarrow S_0$
    $T \leftarrow$ initial_temperature($S_0$)
    while ⋆ termination condition ⋆ do
        for ⋆ $k$ times ⋆ do
            $S^* \in$ neighborhood($S$)
            $\Delta \leftarrow$ cost($S^*$) - cost($S$)
            $P \leftarrow e^{-\Delta/T}$
            if random(0,1) $< P$ then
                $S \leftarrow S^*$
        $T \leftarrow$ function($T$)
    return $S$
```

3.1.1. Implementation details

The definition of all the problem-independent elements is based on the guidelines proposed by M. W. Park and Y. D. Kim in [5], whereas the definitions of neighborhood and cost are postponed to later sections because these may change for different implementations of the framework.

The initial temperature T_0 is based on the average worsening, Δ^*, of a sampled solution and an initial acceptance probability P_0. Park and Kim suggest to compute the average worsening on a trial stage where only uphill moves contribute to the average; on the other hand, this procedure is sensitive to outliers, that occur frequently in the early stages of neural network training due to the random initialization of the network. This sensitivity can be mitigated by removing the top 10% and the bottom 10% of the worsening values, that are likely to contain outliers. The initial acceptance probability typical value varies from 0.1 to 0.9; in this work, the default value is 0.5, even if there may be cases where a different value is preferred. Finally, the initial temperature is computed by inverting the equation used in the acceptance test, as follows:

$$T_0 = -\frac{\Delta^*}{\log P_0} \ . \tag{3.1}$$

The pseudocode for the computation of the initial temperature is reported in Algorithm 8.

Algorithm 8: Temperature initialization pseudocode.

```
Parameters: $S_0$: initial solution.
Function initial_temperature($S_0$)
    S ← $S_0$
    delta_list ← ⋆ empty list ⋆
    for ⋆ $k$ times ⋆ do
        S* ∈ neighborhood(S)
        $\Delta$ ← cost(S*) - cost(S)
        if $\Delta > 0$ then
            delta_list.append($\Delta$)
    delta_list.sort()
    ⋆ remove top 10% and bottom 10% from delta_list ⋆
    $\Delta^*$ ← delta_list.mean()
    $P_0 \leftarrow 0.5$
    $T_0 \leftarrow -\Delta^* / \ln P_0$
    return $T_0$
```

The choice of the termination condition is influenced by the peculiarities of the problem and by the structure of the training. The training process is divided into epochs, whose number is chosen to avoid seeing too many or too few times the training set, that would result in, respectively, overfitting or underfitting. In this work, the number of training epochs is always supposed to be fixed, hence the most appropriate termination condition consists in repeating the while cycle once for each epoch.

The number of sampling in each stage, denoted as k in the pseudocode, should be a value proportional to the size of the instance of the problem. However, for the training problem, each stage corresponds to an epoch that scans the full training set, resulting in a fixed number of iterations, given by the size of the training set divided by the size of each mini-batch. As a consequence, the value of k can be computed automatically following these indications, resulting in a choice that respects the guidelines since k is proportional to the size of the dataset.

One of the most common temperature updating rule is the proportional cooling function, that fundamentally bases the evolution of the temperature to a geometric progression. The progression is fully defined by the initial value and by the common ratio: the former is the already defined initial temperature T_0 while the latter is an additional parameter α, which controls the speed of the cooldown. The definition of the progression, by recurrence and in closed form, follows.

$$T_i = \alpha\, T_{i-1} \quad \leftrightarrow \quad T_i = \alpha^i\, T_0 \tag{3.2}$$

Typical values for α stand between 0.8 and 1.0 excluded; in this work, the default value is 0.9, even though there may be cases where a different value is preferred.

3.2. Variable Neighborhood Search (VNS)

Variable neighborhood search (VNS) [6] is a metaheuristic framework based on the assumption that the effectiveness of metaheuristics is due to the ability of the algorithms to change the neighborhood during the search. Consequently, the authors designed a simple framework that exploits the mentioned assumption without introducing complex elements.

An implementation of VNS requires a global variable to store the best solution found so far. Differently from SA, the exploration does not produce a single sequence of solutions, where the last one is the best. The algorithm iteratively explores a neighborhood, finds its local optimum, and then moves to the next one. The neighborhood to explore next is relative to the best solution found: each solution generates a list of neighborhood structures that are sequentially explored, hence multiple definitions of neighborhood are required. The neighborhood exploration, having the objective of finding a local optimum, follows a local search procedure: if the resulting optimum outperforms all previously found solutions, it is used to update the best solution variable. The exploration of a neighborhood may lead to a new best solution, and to the analysis of a new list of neighborhoods, or it may leave unaltered the best solution, hence forcing one to explore the same set of neighborhoods.

The exploration of a neighborhood searching for a local optimum is an archetypal intensification procedure, whereas the change of neighborhood is purely a diversification move. Hence, the behavior of the VNS framework is an alternation of intensification and diversification procedures, where each one is clearly separated from the others.

The pseudocode for VNS is reported in Algorithm 9. Observe that the framework leaves four elements undefined: the cost, the neighborhood structures, the local optimization procedure, and the termination condition.

Algorithm 9: Variable neighborhood search pseudocode.

```
Parameters: $S_0$: initial solution.
Function VNS_framework($S_0$)
    $S^B \leftarrow S_0$
    $k \leftarrow 1$
    while ⋆ termination condition ⋆ do
        $S \in$ neighborhood($S^B$, k)
        $S^* \leftarrow$ local_optimization(S)
        if $cost(S^*) < cost(S^B)$ then
            $S^B \leftarrow S^*$
            $k \leftarrow 1$
        else
            $k \leftarrow k + 1$
    return $S^B$
```

3.2.1. Implementation details

As for SA, we temporarily postpone the definitions of purely problem-dependent elements, as the neighborhood structures and cost, since they may change for different implementations of the framework. Although the termination condition and local optimization procedure can be expressed in a problem-independent fashion, they must be adapted to the peculiarities of the neural network training problem. The required modifications are discussed below while their actual definition will be provided when used in an actual algorithm.

The local optimization procedure searches for a solution that cannot be further improved with local changes. Observe that it is not feasible to prove that a solution is a local optimum for the neural network training problem due to the high dimensionality of the solution space. However, we can relax the definition of local optimum by assuming that a solution whose neighborhood was analyzed deeply without finding any improving solution is a local optimum. The definition of "analyzed deeply" cannot be provided mathematically, so it must be decided by the developer according to the confidence he requires for declaring a solution to be a local optimum.

The termination condition originally suggested by N. Mladenović and P. Hansen, the authors of VNS, is based on the number of different neighborhoods computable for each solution: defining a fixed number N of neighborhoods, the algorithm terminates when $k > N$, meaning that it is not possible to retrieve a new neighborhood to explore. On the other hand, the definition of the neighborhood of a solution may be parametric in k and may be defined for any natural or real value of k (e.g., the k^{th} neighborhood of a reference solution is the set of solutions at a distance lower than k from the reference solution, given a certain concept of distance). In this case, the above mentioned condition fails. A parametric definition for the neighborhood structures is suitable for the neural network training problem, so it is now necessary to use another stopping condition. The exploration of the solution space for this problem is performed through epochs, consequently it is reasonable to assign to the VNS algorithm as many iterations of the while loop as epochs, resulting in the same termination condition of SA.

4. Training techniques based on metaheuristics

In the literature, the application of metaheuristic frameworks to the training of neural networks is known to be an effective strategy for some specific formulations of the problem. However, the majority of the publications to date analyses only small Artificial Neural Networks (ANN) [7], whereas very little work has been done for other types of network. An important exception to this trend is the study of a SA-based implementation of SGD and its application on convolutional neural networks (CNN) by M. Fischetti and M. Stringher [8]. Their technique, named SGD-SA, is described below and represents the starting point for the SA-based techniques proposed in the present work.

The neural network training problem peculiarity of distinguishing a training set for the optimization algorithm and a test set for the evaluation of its results, introduces the concepts of underfitting and overfitting. These can be related to intensification and diversification, the key concepts of metaheuristic procedures, through the following observations. The intensification phase is performed on the training set with the purpose of reducing both training and test errors. However, if this phase is applied too aggressively, the solution will fit very well the training set, but will perform poorly on the test set, leading to overfitting. On the other hand, the diversification phase is, typically, less dependent on the training set when it explores the solution space, suggesting that training and test errors should be similar. However, if this phase is too aggressive, the solution space will be "randomly" explored and it is unlikely that the solution obtained is any good, leading to underfitting.

The training techniques that will be introduced below indirectly balance the effects of overfitting and underfitting by acting on the intensification and diversification procedures. The aim is to obtain the best from both strategies: get a low training error through the intensification phase, and a comparable test error thanks to the diversification phase. The design of these techniques takes into account the metaheuristic frameworks, the dataset analysis, and the state-of-art techniques. The overall structure of each algorithm is defined by the metaheuristic it implements, but it is adapted to be suitable for the iterative analysis of the training set in the form of mini-batch. When not explicitly defined by the framework, the details of the algorithms are inspired by the problem's peculiarities or by the state-of-art techniques.

The descriptions of the algorithms below are focused on the details that characterize each one of them, while the already described structures deriving from the metaheuristic frameworks or from state-of-art techniques will only be swiftly referred to. Moreover, a pseudocode very close to actual code is provided for each algorithm, with the purpose of highlighting problems that can occur in actual implementations, but can be masked by high level pseudocode.

4.1. Simulated Annealing based techniques

The simulated annealing framework, described in Section 3.1, introduces little overhead to the optimization procedures, hence it is suitable for the design of training algorithms. SA does not require to store a large number of solutions and very few operations are performed on them at each iteration. The result is an algorithm with low space and time complexities, which allows the management of large solutions with feasible resources. Moreover, the double loop of SA can be adapted to implement the strategy of state-of-art algorithms to analyze the training set by means of epochs and mini-batches.

4.1.1. Common elements

The implementation of SA-based metaheuristics for the neural network training requires many design choices to be made. However, the overall structure of such algorithms is inherited by the framework, hence it is shared among the algorithms; meaning that there are choices common to all the algorithms of this kind. The most important common elements are described below.

In-place network modification

Consider the pseudocode for SA presented in Algorithm 7, and analyze the space and time complexity with respect to the size of a solution. The current solution examined is stored in a global variable and, at each iteration, a new solution is generated from the current one and saved in a variable. Then, if the acceptance test is successful, the value of the previous solution is overwritten with the value of the new one. Simplistically, the algorithm requires to store two solutions and performs two operations on solutions.

Suppose that the new solution generated in each iteration is not stored in a new variable but straightaway overwrites the current solution, before the acceptance test is taken. If the generated solution passes the acceptance test no further operations are required, whereas the previous solution must be reset in case of test failure. This version of the algorithm requires to store only one solution and performs one operation on solutions if the acceptance test is passed, two otherwise. Since it is reasonable to assume that the majority of tests are passed, this strategy is more efficient both in time and space.

Observe that if the previous assumption does not hold, it is likely that the number of moves performed is too small to lead to a good solution. This situation may occur for bad parameters (e.g., too low initial temperature) or unwise design (e.g., bad definition of neighborhood). In such cases, the algorithm cannot work well and must be redesigned. A second observation is related to the solution reset. The information required to restore the previous solution, represented by the gradient, is the same used to generate the new solutions, hence both strategies implicitly need to store it. Since such information is required by both strategies, it is ignored in the comparison.

Stochastic cost computation

The cost associated to the solutions vary from technique to technique, but it is possible to distinguish two types of cost function: dataset independent and dataset dependent. Cost functions belonging to the dataset independent type can be computed directly from a solution, without any additional knowledge (e.g., norm functions), whereas dataset dependent costs require the training set as an additional information (e.g., loss functions). In the SA pseudocode, the cost of a solution is computed with the solution only, hence ignoring the dataset dependent costs, whose complexity must be carefully managed while adapting the framework.

Typically, the computation of an exact dataset dependent cost requires to scan the whole training set. However this would make SA-based algorithms unfeasible in practice since cost computations are performed at each move. The problem of solution cost computation is identical to the problem of gradient computation analyzed in Section 2.1.1, hence the same stochastic approach can be used. Instead of scanning the whole training set, a single mini-batch is analyzed for the computation of the cost: the procedure loses accuracy but decreases significantly the complexity, making SA-based algorithms feasible. The dataset dependent costs are then computed stochastically, as it is done for the gradient, whereas dataset independent costs can be computed directly as indicated in the SA pseudocode.

Gradient-related neighborhood

The neighborhood of a solution is defined as the set of solutions that, according to the gradient, may lead to an improvement. The anti-gradient points towards the direction of the maximum decrease of the loss function, hence the solutions found in this direction are likely to improve the previous solution. Formally, the neighborhood of a solution is a half-line in the solution space, starting from the reference solution and extending in the direction of the anti-gradient, which can be expressed as follows:

$$N(S) = \{S' \mid \exists\, \eta \in \mathbb{R}^+,\ S' = S - \eta\nabla\} \tag{4.1}$$

The sampling of an element from this set is performed indirectly through parameter η, since there is one-to-one correspondence with the solutions in the neighborhood. The random generation of η follows a specific probability distribution that is inspired by the role of learning rate in SGD, whose update rule is similar to the structure of

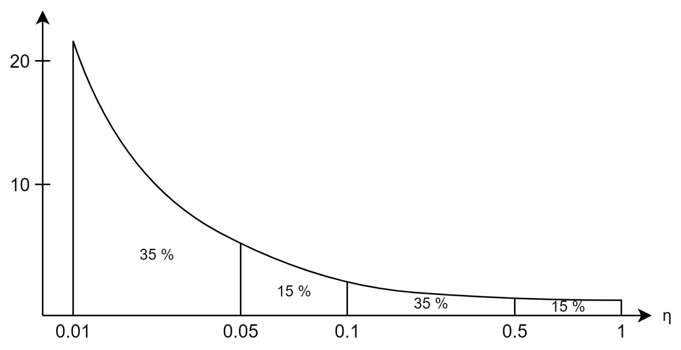


Figure 4.1.: Plot of the probability density function of η, in logarithmic scale for the x-axis. The percentages indicate portions of the area under the curve, hence the probabilities of the correspondent intervals.

neighborhood. The typical learning rate for SGD spans between multiple orders of magnitude, hence the probability distribution of η must be able to assign a reasonable probability to very different values. For example, a naïve uniform distribution would annihilate the probability of choosing small values and enhance it for large values. The strategy used in our work consists in generating the magnitude order, with an uniform probability distribution, and then performing an exponentiation to get the actual value. Supposing a range for η from 0.01 to 1, its value is computed as follows.

$$\begin{cases} m \backsim U(-2,0) \\ \eta = 10^m \end{cases} \tag{4.2}$$

The resulting distribution can be expressed by means of its Probability Density Function (PDF) as follows:

$$f(\eta) = \begin{cases} 0 & x < 10^{-2} \\ \frac{1}{2\ln(10)x} & 10^{-2} < x < 10^0 \\ 0 & x > 10^0 \ . \end{cases} \tag{4.3}$$

A simple plot of the function, reported in Figure 4.1, shows that the probability at which a sampled learning rate belongs to any specific order of magnitude i, expressed as the interval $[10^{-i}, 10^{-i+1}]$, is the same for all the values of i, hence the learning rate can assume all the values in the desired range with a reasonable probability. The same property holds also for wider ranges of η, however for our purposes the range $[10^{-2}, 10^0]$ is enough.

Temperature initialization

The temperature initialization procedure adopted by SA-based techniques follows closely the pseudocode provided in Algorithm 8; however, some cost functions are less sensitive to outliers and do not require to apply the outlier removal strategy presented above. The structure of the temperature initialization procedure is obtained by applying trivial modifications to a training epoch, thus a detailed description of the procedure is omitted (it can be inferred by the training epochs, whose pseudocode is provided, and by the SA framework description). Moreover, the variances between different techniques' temperature initialization procedures are due to differences in the training epoch, hence the analysis of these procedures would be redundant.

4.1.2. SGD-SA

The SGD-SA technique is the result of introducing in simple SGD the ability of rejecting moves, according to a SA criterion. As for SGD, each mini-batch is used to compute a move, but the replacement of the previous solution occurs only if the SA acceptance test is passed. The cost function is the loss: it is computed before and after the modification of the weights and the difference is fed to the acceptance test. Moreover, the SA procedure requires that for each move the solution to test is sampled randomly from the current solution neighborhood: differently from SGD, the learning rate is chosen randomly for each move. Observe the extreme cases of SGD-SA: if the temperature is infinite, the algorithm accepts all moves, hence it behaves as SGD with randomized learning rate, whereas if the temperature tends to zero, all worsening moves are rejected, hence the algorithm behaves as a randomized SGD which avoids any worsening move.

The SA ability of accepting worsening moves and the randomization in the learning rate are diversification mechanisms added to the SGD algorithm. The more balanced exploitation of intensification and diversification reduces the overfitting, however, the less effective intensification reduces the ability of the model to improve the solutions. The overall performance depends on the tradeoff between these two aspects.

The pseudocode presented in Algorithm 10 describes in detail the SGD-SA technique. The most noticeable differences from the SA framework pseudocode are due to the in-place modification of the network: the costs of the solutions are computed in different moments and the acceptance test has become a rejection test.

4.1.3. SGD-SA-R

The SGD-SA-R technique, where the 'R' stands for regularization function, is a new method designed as an improvement of SGD-SA, following its structure but changing the cost function. This technique patches some flaws of SGD-SA which introduce a bias on the acceptance test, making it easier to be passed. The SGD-SA cost function is the same function whose gradient is used to compute the moves, therefore it is unlikely that the cost of the new solution, generated following the anti-gradient of the cost function, will be worse. Moreover, the cost is computed with respect to the same mini-batch, hence the new solution cost, which took advantage of the information carried by the cost gradient, is not a fair estimation of the real cost on the training set. The problems

Algorithm 10: SGD-SA pseudocode.

```
Parameters: $\theta$: network weights, $D$: training set, $L$: loss function, $E$: number
            of epochs, $C$: cost, $T$: temperature, $\alpha$: cooling factor.
Function SGD-SA($\theta$, $D$)
    $T \leftarrow$ initial_temperature($\theta$)
    foreach epoch $e \in \{0, \dots, E-1\}$ do
        foreach mini-batch $b \in D$ do
            $C_i \leftarrow$ compute_loss($\theta$, b, L)
            $\nabla L \leftarrow$ compute_gradient($\theta$, $b$, $L$)
            $\eta \leftarrow 10^{rand(-2,0)}$
            $\theta \leftarrow \theta - \eta \nabla L$
            $C_f \leftarrow$ compute_loss($\theta$, b, L)
            $\Delta \leftarrow C_f - C_i$
            $P \leftarrow e^{-\Delta/T}$
            if $rand(0,1) \geq P$ then
                $\theta \leftarrow \theta + \eta \nabla L$
        $T \leftarrow T\alpha$
    return $\theta$
```

of SGD-SA can be overcame by changing the cost function: if the cost function and the generation of new solutions are independent, then the comparison between the solution costs can be performed more fairly.

The cost function decides which solutions are accepted, so it must measure the properties aspired in a solution. A good solution should have a low training error and should generalize well: the first property is expressed by the loss, while the second property cannot be measured directly. However, simpler models are more likely to generalize well because it is harder to overfit the training set without complicating the model. The choice of the new cost function is inspired by the regularization, which adds a term to the loss in order to take into account the complexity of the model: the most common functions used to measure the complexity are the L1 and L2 norms.

As a result, new solutions are generated exploiting the gradient of the loss while the acceptance test is based on their complexity. This formulation makes the acceptance test completely independent from the training set, hence reducing the overfitting, while at the same time exploits the gradient to provide potentially good solutions. Although the behavior of the norm, applied to a sequence of solutions following the loss gradient, has a major impact on the algorithm performance, it is unpredictable. Hence it is hard to estimate a priori the performance of SGD-SA-R: even if it was designed to repair SGD-SA flaws, no performance improvement can be ensured.

The pseudocode for SGD-SA-R is provided in Algorithm 11; it differs from SGD-SA pseudocode only in the cost computation. In early epochs, the norm function is likely to generate a large number of outlier solutions, with respect to their costs, hence the temperature initialization must perform the outliers' removal procedure.

Algorithm 11: SGD-SA-R pseudocode.

```
Parameters: $\theta$: network weights, $D$: training set, $L$: loss function, $E$: number
            of epochs, $C$: cost, $T$: temperature, $\alpha$: cooling factor, $i$: norm
            type.
Function SGD-SA-R($\theta$, $D$)
    $T \leftarrow$ initial_temperature($\theta$)
    foreach epoch $e \in \{0, \dots, E-1\}$ do
        foreach mini-batch $b \in D$ do
            $C_i \leftarrow$ compute_norm($\theta$, i)
            $\nabla L \leftarrow$ compute_gradient($\theta$, $b$, $L$)
            $\eta \leftarrow 10^{rand(-2,0)}$
            $\theta \leftarrow \theta - \eta \nabla L$
            $C_f \leftarrow$ compute_norm($\theta$, i)
            $\Delta \leftarrow C_f - C_i$
            $P \leftarrow e^{-\Delta/T}$
            if $rand(0,1) \geq P$ then
                $\theta \leftarrow \theta + \eta \nabla L$
        $T \leftarrow T\alpha$
    return $\theta$
```

4.1.4. SGD-SA-LR

The SGD-SA-LR technique, where "LR" stands for loss regularized, performs a trade-off between SGD-SA and SGD-SA-R by using a cost function which depends both on the loss and on the norm. The idea is that SGD-SA-R's independence from the loss slows down the improvement of the solution, albeit it is effective in reducing the overfitting.

Again, the cost function is inspired by the regularization strategy: the cost is a linear combination between the loss and the norm; assuming that none of them can have a null coefficient, it is possible to use the following formulation.

$$Cost = Loss + \lambda\ Norm \tag{4.4}$$

The parameter λ is a crucial hyper-parameter: it manages the impact which the two components have on the cost, influencing both the behavior and the performance of the algorithm. As an example, for $\lambda = 0$ the technique degenerates to SGD-SA, whereas for $\lambda \to +\infty$ it degenerates to SGD-SA-R.

Due to its importance, the tuning of λ is critical. However, it is hard to compute an ad-hoc value for each instance of the problem: the norm of a solution, and its behavior during the training, depend on the specific structure and size of the considered network. Consequently, the value of λ requires a parameter tuning process in order to be set to a reasonable value. Nevertheless, with some experience it is probably possible to design a rule-of-thumb to find an acceptable value, avoiding the overhead of the hyper-parameter tuning.

The pseudocode for SGD-SA-LR is reported in Algorithm 12 and, by construction, it is similar to both SGD-SA and SGD-SA-R ones. The norm component in the cost leads to the generation of outliers in early epochs, as previously discussed for SGD-SA-R, hence the temperature initialization procedure must remove the outliers before computing the temperature.

Algorithm 12: SGD-SA-LR pseudocode.

```
Parameters: $\theta$: network weights, $D$: training set, $L$: loss function, $E$: number
            of epochs, $C$: cost, $T$: temperature, $\alpha$: cooling factor, $i$: norm
            type, $\lambda$: loss norm coefficient.
Function SGD-SA-LR($\theta$, $D$)
    $T \leftarrow$ initial_temperature($\theta$)
    foreach epoch $e \in \{0, \dots, E-1\}$ do
        foreach mini-batch $b \in D$ do
            $C_i \leftarrow$ compute_loss($\theta$, $b$, $L$) + $\lambda$ compute_norm($\theta$, i)
            $\nabla L \leftarrow$ compute_gradient($\theta$, $b$, $L$)
            $\eta \leftarrow 10^{rand(-2,0)}$
            $\theta \leftarrow \theta - \eta \nabla L$
            $C_f \leftarrow$ compute_loss($\theta$, $b$, $L$) + $\lambda$ compute_norm($\theta$, i)
            $\Delta \leftarrow C_f - C_i$
            $P \leftarrow e^{-\Delta/T}$
            if $rand(0,1) \geq P$ then
                $\theta \leftarrow \theta + \eta \nabla L$
        $T \leftarrow T\alpha$
    return $\theta$
```

4.1.5. SGD-SA-DB

The SGD-SA-DB technique, where "DB" stands for double (mini-)batch, is an improvement of SGD-SA that removes the bias in the acceptance test, keeping the loss as cost function. As discussed for SGD-SA-R, the bias originates in the computation of the costs of the generated solutions: the dependency between the process that generated the solutions and the cost function leads to a stochastically unfair result. The bias is removed if and only if the computation of the cost is stocastically valid both for the initial solution and the corresponding generated one, allowing for a fair comparison.

Instead of using two different functions for the gradient and for the cost, SGD-SA-DB uses the loss for both, but with two different mini-batches. Since the loss of a given set of weights depends both on the function and on the mini-batch sampled, computing the loss with different mini-batches is similar to using two different functions. For each move, two mini-batches must be sampled from the training set: one to compute the gradient, hence the new solution, and the other to compute the cost of both the initial and the new solution. Although the number of mini-batches per move is doubled, it does not imply that the overall number of moves is halved: a mini-batch used to compute the gradient can still be used in a different move to compute the cost.

The pseudocode for SGD-SA-DB is reported in Algorithm 13. The main difference with respect to previously presented algorithms is the management of two mini-batches per move: given an ordered sequence of mini-batches composing the training set, the i-th move uses the pair formed by the i-th and $(i + 1)$-th mini-batches. This sliding window mechanism is implemented introducing a little overhead at the beginning and at the end of each move.

4.2. Variable Neighborhood Search based techniques

The variable neighborhood search, described in Section 3.2, provides a simple structure to perform intensification and diversification, and its complexity depends mostly on the specific strategies which implement these two operations. Whereas the diversification procedure must be designed ad-hoc, the intensification can be performed by effective state-of-art techniques, directly embedded in the framework.

The limit on the number of epochs is shared by both state-of-art techniques and by the framework, hence in VNS-based techniques such limit must be managed carefully. The idea is to perform, in each epoch of the VNS, a single optimization step instead of the full search for the local optimum, as indicated in the framework. If we consider a single optimization step as a training epoch of a state-of-art technique, we get that VNS stopping condition actually limits the number of training epochs performed by the embedded technique.

Algorithm 13: SGD-SA-DB pseudocode.

```
Parameters: $\theta$: network weights, $D$: training set, $L$: loss function, $E$: number
            of epochs, $C$: cost, $T$: temperature, $\alpha$: cooling factor.
Function SGD-SA-DB($\theta$, $D$)
    $T \leftarrow$ initial_temperature($\theta$)
    foreach epoch $e \in \{0, \dots, E-1\}$ do
        $b_{grad} \leftarrow \star$ "last" mini-batch $\in D \star$
        foreach mini-batch $b \in D$ do
            $b_{test} \leftarrow b$
            $C_i \leftarrow$ compute_loss($\theta$, $b_{test}$, $L$)
            $\nabla L \leftarrow$ compute_gradient($\theta$, $b_{grad}$, $L$)
            $\eta \leftarrow 10^{rand(-2,0)}$
            $\theta \leftarrow \theta - \eta \nabla L$
            $C_f \leftarrow$ compute_loss($\theta$, $b_{test}$, $L$)
            $\Delta \leftarrow C_f - C_i$
            $P \leftarrow e^{-\Delta/T}$
            if $rand(0,1) \geq P$ then
                $\theta \leftarrow \theta + \eta \nabla L$
            $b_{grad} \leftarrow b_{test}$
        $T \leftarrow T\alpha$
    return $\theta$
```

4.2.1. ADAM-VNS

The ADAM-VNS technique exploits the effectiveness of the ADAM optimizer to perform the intensification, while the diversification is performed by adding Gaussian noise to the weights.

The canonical VNS intensification stops when it reaches a local minimum; however, for this problem, it is not feasible to test the local optimality of a solution, hence a different stopping criterion is used. The local search stops when the current solution does not show any significant improvement in the last epochs: if the current cost has improved less than a fixed threshold in comparison with the cost of a fixed number of precedent epochs, then the current solution is declared an approximation of a local minimum.

The cost function used in ADAM-VNS is the loss: it has a key role for the behavior of the algorithm since it is the parameter used to decide if the local optimality criterion is satisfied by the current solution. The cost fluctuations derived from a stochastic computation may interrupt the improvement of a solution prematurely, or waste resources for the optimization of a solution stuck in a local optimum. The inaccuracy derived from a stochastic approach is not acceptable in this context, hence the loss must be computed on the whole training set. This choice is computationally heavy since in each epoch the training set is analyzed twice: once for the ADAM training and once for the cost computation. Although the computation of the loss is faster than the training, the overall complexity is roughly doubled, if compared with a stochastic approach.

The introduction of Gaussian noise in the diversification procedure leads to a probabilistic neighborhood structure, whose size is expressed parametrically with respect to the standard deviation of the noise: higher values enhance the probability of finding farther solutions. VNS diversification aims to randomly perturb the current solution, allowing it to escape from a local minimum; consequently, the actual structure of the neighborhood is not as important as the effectiveness of the perturbation applied.

The pseudocode for ADAM-VNS is presented in Algorithm 14. Although the code, if compared to VNS pseudocode, shows many differences, in term of structure, the resulting behavior is the same. The local search of ADAM is still performed until an approximated local optimum is found, even if here it is performed one epoch at the time. This implies that the local optimality must be checked at each iteration, since in each epoch the single optimization step may not lead to a local optimum. The algorithm can be rearranged to follow more closely the VNS pseudocode. However, this version is more similar to the previously presented technique and allows one to use common tools to manage the results.

Algorithm 14: ADAM-VNS pseudocode.

```
Parameters: $\theta$: network weights, $D$: training set, $L$: loss function.
Function ADAM-VNS($\theta$, $D$)
    $\theta^B \leftarrow \theta$
    $k \leftarrow 1$
    foreach epoch $e \in \{0, \ldots, E-1\}$ do
        $\theta \leftarrow$ ADAM_training_epoch($\theta$, $D$)
        if compute_loss($\theta$, $D$, $L$) $<$ compute_cost($\theta^B$, $D$, $L$) then
            $\theta^B \leftarrow \theta$
            $k \leftarrow 1$
        $I \leftarrow \star$ relative improvement in the last p epochs $\star$
        if $I < \star$ improvement threshold $\star$ then
            $\theta \leftarrow \theta^B$ + gaussian_noise$(0, \sigma k)$
            $k \leftarrow k + 1$
    return $\theta^B$
```

5. Implementation details

The training techniques described in the previous sections have been implemented and tested to evaluate their performance. The software and hardware tools used to achieve this goal are described below.

5.1. Hardware elements

The neural network training is known to be a resource-demanding process: the datasets and the networks involved are typically large structures that must be analyzed many times before obtaining significant results. On the other hand, the dataset analysis by means of mini-batches is suitable to be parallelized, so it can be performed efficiently by GPUs. However, general-purpose processing cannot be performed by GPUs without additional software: for NVIDIA's ones, such a technology is named CUDA. Hence, for efficiency purposes, all the trainings performed in this project were computed by CUDA-enabled NVIDIA GPUs.

The development of the project was performed locally on a Dell G3 provided with a i7-8750H Intel processor, 16 GB of RAM, and a NVIDIA GeForce GTX 1060. Although its technical specifications were more than acceptable for a single training, the large number of tests planned would have required too much time to be completed.

In order to get the required computing power, the Department of Information Engineering of the University of Padova granted us exclusive access to two NVIDIA Titan Xp (kindly provided by NVIDIA) installed on a remote machine in the department. Moreover, the technical characteristics of these GPUs allowed the management of multiple training process simultaneously, further decreasing the computation time.

5.2. Software tools

5.2.1. Languages

The main language chosen is Python: the simplicity of its syntax and the large number of libraries dedicated to machine learning made it the most suitable language for this project. Secondarily, the management of the tests execution and the interaction with the remote machine was performed with UNIX shell scripts.

5.2.2. Libraries

The most important library imported into the project is the one providing the machine learning tools. The choice was guided by the library suitability for designing and testing custom optimization techniques. The library that most fitted the requirements was PyTorch, an open source machine learning framework. It allows to manipulate the training process in great detail, still providing a developer-friendly environment. Moreover, PyTorch allows to easily move the computation to the GPUs, enhancing the performance of the training.

The graphical visualization of test results is useful to better understand their meaning. This functionality is provided by the Matplotlib library: a comprehensive plotting library for Python.

5.2.3. External programs

The project was developed in Windows 10 with Visual Studio Code as IDE, Git as version control software, and GitHub as host of the project repository. The tests were managed locally in Fedora 32, while the actual execution was performed remotely by the NVIDIA Titan Xp GPUs.

5.2.4. Version issue

The versions of programs and libraries on local and remote machines vary: the local one uses Python 3.8.2, PyTorch 1.5.0, Torchvision 0.6.0, and Matplotlib 3.2.2, whereas the remote one uses Python 3.6.4, PyTorch 0.4.0, Torchvision 0.2.1, and Matplotlib 2.2.2. Although such difference does not lead to any significant problem, some parts of the code had to be modified to work on both versions.

5.3. Code structure

The use of two different machines, one local and one remote, introduced the problem of the synchronization of the code. As a consequence, the source code has been divided in four files only, in order to prevent the occurrence of hard to spot errors related to the unsynchronization between the machines. The content of the files is described below.

main.py The *main* file contains executable code to perform a training, following the indications specified by the user through command line parameters. In particular, the program can be divided into four tasks: parse the command line, load the elements required (e.g., dataset), execute the training, save the results. However, almost all the operations performed by the *main* are actually calls to functions in *util*.

util.py The *util* file is the heart of the project since it contains all the functions of the project. In particular, the functions perform the following operations: parse the command line, load elements, define training techniques, plot training related results, import and export training results.

error.py The *error* file defines custom errors to improve the effectiveness of the debug.

model.py The *model* file contains the definition of ResNet34. The versions of PyTorch on local and remote machines manage the model loading differently; hence, to avoid two versions of the code, the ResNet34 network was not imported from the library but, rather, defined in the *model* file. The network implementation was taken from the source code of the latest version of PyTorch and then adapted to the older one.

6. Test results

The techniques presented above are tested by the training of a specific network on different benchmark datasets. The test results of each technique are analyzed individually with the purpose of understanding the algorithm behavior and the reasons behind its performance. Then, the test results of different techniques are compared to evaluate their relative effectiveness and to highlight the impact that small variations in the algorithms have on the performance.

6.1. Experimental setup

The scientific validity of the tests can be achieved only if they are performed identically with the sole exception of the technique used for the training, which is the variable under study. Excluding technique-related parameters, the other training elements are fixed for all tests and are defined below.

6.1.1. Generic training parameters

The parameters described in this section define important properties of the training which are independent from the dataset and the network. In particular we refer to the number of epochs, the management of the random seeds, and the loss function.

The number of epochs manages the number of times the dataset is fully analyzed and influences directly the complexity of the training. Empirical observations on the considered techniques suggest that the performances stop improving after about 80-100 epochs of training. Consequently, the value chosen for the number of epochs is 100: more epochs are likely to generate overfitting, while fewer would prematurely stop the training.

The non-deterministic nature of the training process implies that different executions may lead to different results, even if the same parameters are the same. Therefore, the results of a single test are not representative of the overall performance of the technique being tested. The influence of randomness on the results can be mitigated by repeating each test multiple times with different random seeds. For this reason, the tests are executed with 10 seeds: the integer numbers from 0 to 9.

The loss function chosen is the cross-entropy loss; which is one of the most commonly used loss functions for a classification task. This function evaluates both the correctness and the confidence of each prediction, penalizing more the wrong with high confidence predictions. A classification task with C classes can be solved by a network that estimates the sample likelihood of belonging to each class. Given a vector x of size C (the output of the network) and a class c (the correct class of the input), it is possible to define the cross-entropy loss as follows [15]:

$$Loss\ (x, c) = -\log\left(\frac{e^{x[c]}}{\sum_{i=1}^{C} e^{x[i]}}\right) \tag{6.1}$$

6.1.2. Datasets

The datasets chosen for the tests are MNIST [9], Fashion-MNIST [10], and CIFAR-10 [11]. These datasets are well known in the literature and are commonly used as benchmarks. Moreover, their size is quite large and the number of samples per class is balanced both in the training and test sets. The basic information about the chosen datasets is reported in Table 6.1.

	Training images		Test images			
	Total	Per class	Total	Per class	Image size	Classes
MNIST	60000	5421-6742	10000	892-1135	28x28x1	10
F-MNIST	60000	6000	10000	1000	28x28x1	10
CIFAR-10	50000	5000	10000	1000	32x32x3	10

Table 6.1.: Basic structure of MNIST, Fashion-MNIST, and CIFAR-10

Despite the peculiarities of the datasets, the pre-processing performed is the same for all of them. At each epoch, the training set is randomly shuffled, then partitioned into mini-batches composed of 500 samples. Each image undergoes a data augmentation process: firstly, it is scaled to size 32x32x3 for compatibility with the network used; secondly, a padding of 4 pixels is added to each plane of the image; finally, the image is randomly cropped, to get again the size 32x32x3, and normalized. Observe that the cropping is not unique, due to the added padding. The slightly different images that can be produced, starting from the same one, greatly reduce the overfitting.

6.1.3. Network

The tests are performed on ResNet34 [12]: a deep residual network composed by 34 layers and described in Figure 6.1. The residual network design requires a relatively small number of weights and its performance was state-of-art in 2015, when this architecture won important competitions. Machine learning libraries implement several versions of ResNet, differing by the number of layers, but all with the same structure.

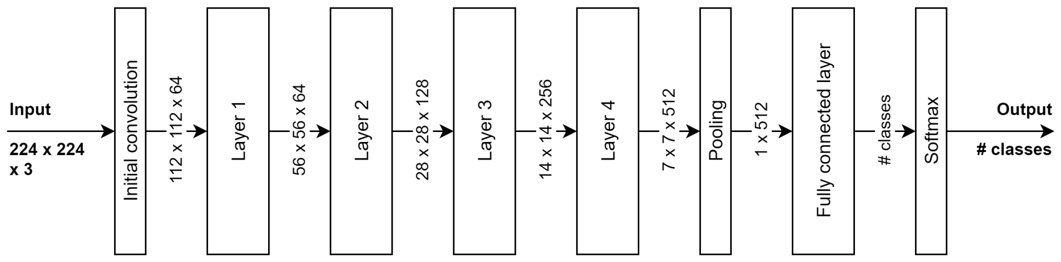


Figure 6.1.: Schematic structure of ResNet34.

Although the ResNet34 network was designed to receive as input RGB images of size 224x224, it can work with smaller images, provided that they are compatible with the network behavior. The constraints on the size of the input images are a consequence of the following characteristics of the network: the first convolution requires an image with 3 channels and, in the forward propagation, the dimensions of the image are halved five times. Therefore, the images compatible with ResNet34 must have exactly 3 channels and dimensions larger than 32x32.

6.2. Results management

The tests produce two different types of results: the performance of the trained network and a list of intermediate data, related to each epoch. The latter information describes how the performance of the network evolved through the training, and provides additional details on the metaheuristic-specific operations executed in each epoch. The performance is measured by the loss function and by the error function, defined as the ratio between misclassified and the total number of samples. These functions are computed on both the training and test sets in order to allow for an evaluation of the overfitting. An example of the results returned by a single test is provided in Table 6.2 and Table 6.3.

Optimizer	Dataset	Seed	Train loss	Test loss	Train error	Test error
NAG	MNIST	0	0.01466	0.03451	0.00538	0.0102

Table 6.2.: Performance measures of the NAG optimizer on a specific test.

Our tests have three main goals: evaluate the effectiveness of individual techniques, understand the reasons behind the techniques performance, and compare the algorithms. Therefore, the large amount of raw data produced by the tests is analyzed with different strategies to obtain as much information as possible on the techniques under study.

The analysis of individual techniques is performed by evaluating their results on each

Epoch	Time	Train loss	Test loss	Train error	Test error
0	65.8	1.9389	1.9137	0.7084	0.7125
1	126.5	0.9848	0.9368	0.3154	0.3011
2	186.7	0.2460	0.2271	0.0785	0.0753
...	...	...	...	...	...
99	4245.8	0.1959	0.7377	0.0698	0.2020

Table 6.3.: Training information (partial) for NAG on MNIST with seed 0.

dataset. Since every test is repeated multiple times, with different random seeds, the corresponding results are grouped by seed and then aggregated. This procedure consists in the application of three different functions to the data: the *min* function, the *arithmetic mean* function, and the *over* function. We define the over function as the average difference between the test and the training values, to evaluate the overfitting. These functions, applied to both loss and error, describe the overall performance of an algorithm for a specific dataset. As an example, the raw results of NAG on MNIST, displayed in Table 6.4, are aggregated as shown in Table 6.5.

Optimizer	Dataset	Seed	Training loss	Test loss	Training error	Test error
NAG	MNIST	0	0.01466	0.03451	0.00538	0.0102
NAG	MNIST	1	0.01469	0.03916	0.00513	0.0118
NAG	MNIST	2	0.01684	0.03202	0.00600	0.0100
NAG	MNIST	3	0.01441	0.03627	0.00542	0.0113
NAG	MNIST	4	0.01180	0.03588	0.00393	0.0099
NAG	MNIST	5	0.01099	0.04260	0.00368	0.0120
NAG	MNIST	6	0.01328	0.03998	0.00475	0.0117
NAG	MNIST	7	0.01892	0.04058	0.00635	0.0122
NAG	MNIST	8	0.01330	0.03601	0.00447	0.0105
NAG	MNIST	9	0.02603	0.04458	0.00840	0.0136

Table 6.4.: Test results for NAG on MNIST.

	Loss			Error		
Dataset	Min	Mean	Over	Min	Mean	Over
MNIST	0.03202	0.03816	0.02267	0.00990	0.01132	0.005969

Table 6.5.: Aggregated test-set results for NAG on MNIST.

The information on the epochs, returned by the tests, is useful to understand the behavior of the techniques during the training, since it helps to correlate the algorithm decisions with their effect on the performance. Due to its structure and size, such an information is clearer if presented by means of plots. The test results for a specific technique can generate many similar plots; to avoid redundancy, a single series of data is plotted for each technique. The chosen series is the one related to the test which best performed on CIFAR-10. Such a series should be the best expression of the algorithm on the hardest dataset considered, therefore the most meaningful.

Additional tests are performed whenever the previous results lead to multiple hypotheses on the behavior of a technique. Differently from the main tests, designed to get a complete view of the techniques, these ones are designed to verify specific claims. Thus, the analysis can be limited to a single dataset, CIFAR-10, for time reasons.

The comparison between techniques is achieved with performance profile, a tool introduced by E. D. Dolan and J. J. Moré [13] whose purpose is to compare the performance of optimizers in a fairly and unbiased way. The implementation of performance profiling we use is provided by D. Salvagnin [22].

6.3. Control-sample techniques

Three state-of-art techniques are tested to provide results that could be used as reference for the newly introduced techniques. The chosen techniques are: NAG, for the exploitation of the history of moves; SSGD, for the management of the learning rate; ADAM, for the automatic management of the learning rate.

All these algorithms are well known in the literature, so a deep analysis on their behavior is not required here. In the present work, the results of their tests are simply used as comparison for the newly introduced techniques. Below, key information is reported for each one: the hyper-parameters used in the tests, the aggregated results in tabular form, and a plot representing an example of training, obtained from the epochs information related to the best run on CIFAR-10.

6.3.1. Results

NAG has two hyper-parameters to tune: the learning rate η and the momentum coefficient μ. For the tests, these parameters were set to $\eta = 0.1$ and $\mu = 0.9$.

SSGD with multi-step scheduler defines the behavior of the learning rate with three hyper-parameters: the initial learning rate, the width, and the decay of the learning rate at each step. For the test, the initial learning rate is set to 0.1, the width is set to 35 epochs, and the decay is set to 0.1. Therefore the learning rate will assume the following values: 0.1 for epochs 0-34, 0.01 for epochs 35-69, and 0.001 for epochs 70-99.

ADAM has four hyper-parameters: the learning rate and three mathematical constants used for the moment estimation. There is no reason to change the default values of the constants, since their impact on performance is not trivial. Although the learning rate is an important parameter, it is auto-tuned by the algorithm and can then be left to its default value. For completeness, the hyper-parameter are set as follows: $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$, and $\eta = 0.1$.

The tests results for NAG, SSGD, and ADAM are reported, respectively, in Table 6.6, Table 6.7, and Table 6.8, while examples of their behavior are displayed in Figure 6.2, Figure 6.3, and Figure 6.4.

	Loss			Error		
Dataset	Min	Mean	Over	Min	Mean	Over
MNIST	0.03202	0.03816	0.02267	0.00990	0.01132	0.005969
F-MNIST	0.03498	0.03897	0.02419	0.00990	0.01150	0.006378
CIFAR-10	0.6590	0.7438	0.5672	0.1811	0.1973	0.1342

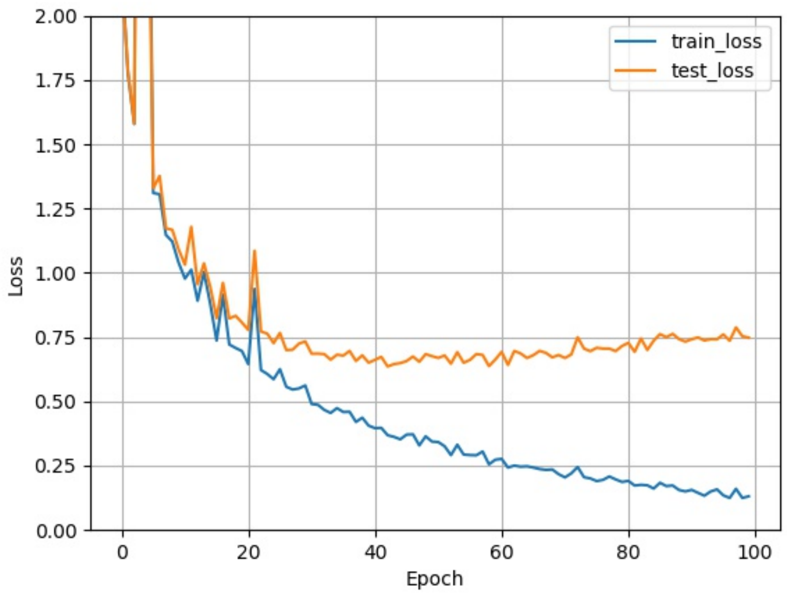
Table 6.6.: Aggregated test-set results for NAG.

	Loss			Error		
Dataset	Min	Mean	Over	Min	Mean	Over
MNIST	0.03020	0.03351	0.02122	0.00920	0.01051	0.006321
F-MNIST	0.02914	0.03449	0.02215	0.00910	0.01068	0.006500
CIFAR-10	0.6947	0.7219	0.4357	0.2137	0.2194	0.1178

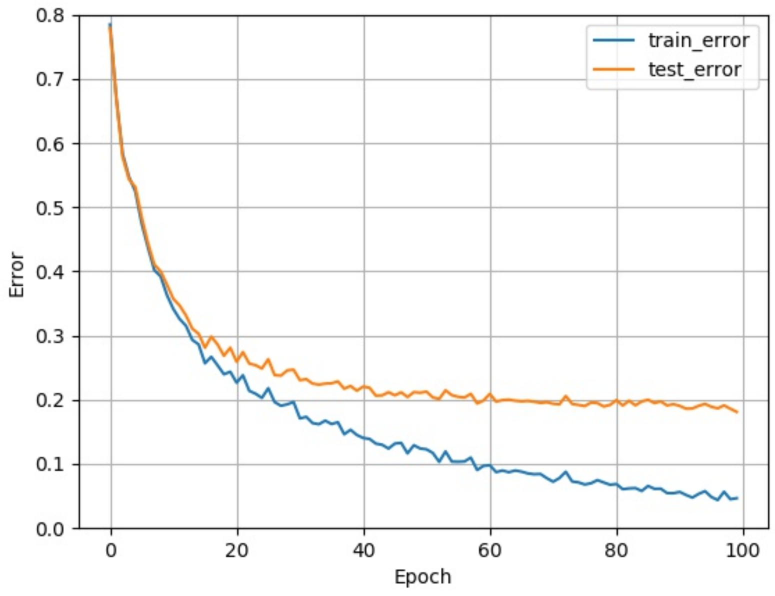
Table 6.7.: Aggregated test-set results for SSGD.

	Loss			Error		
Dataset	Min	Mean	Over	Min	Mean	Over
MNIST	0.03067	0.03658	0.02358	0.00860	0.01017	0.005774
F-MNIST	0.03154	0.03893	0.02576	0.00870	0.01050	0.005982
CIFAR-10	0.6761	0.7204	0.5994	0.1631	0.1704	0.1273

Table 6.8.: Aggregated test-set results for ADAM.

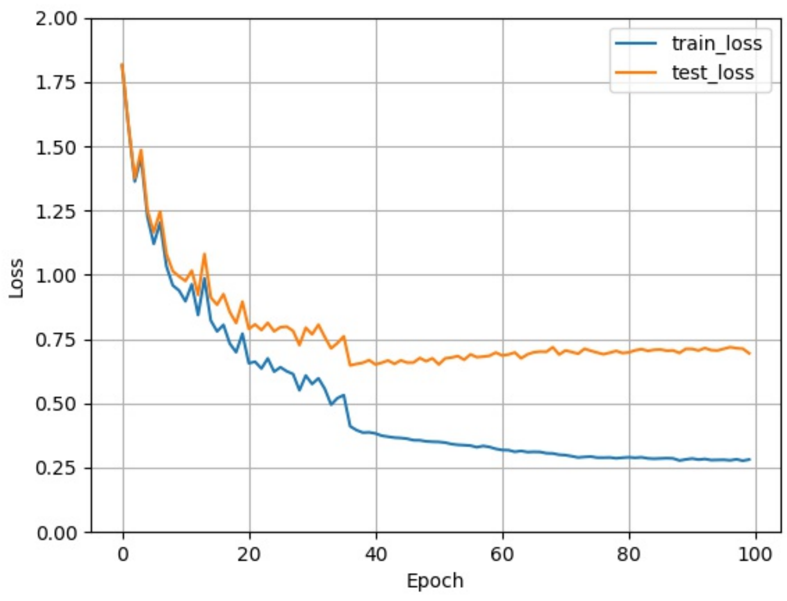


(a) Training and test loss.

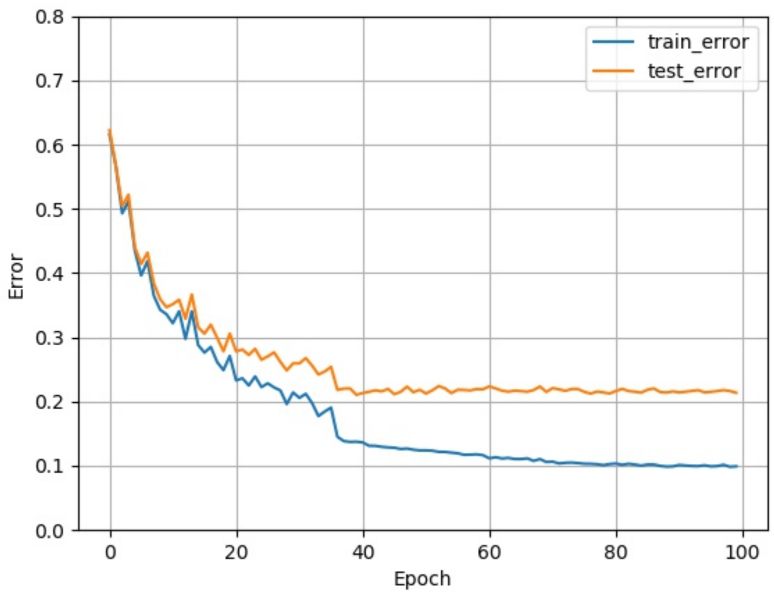


(b) Training and test error.

Figure 6.2.: Best performance of NAG on CIFAR-10 (seed 1).

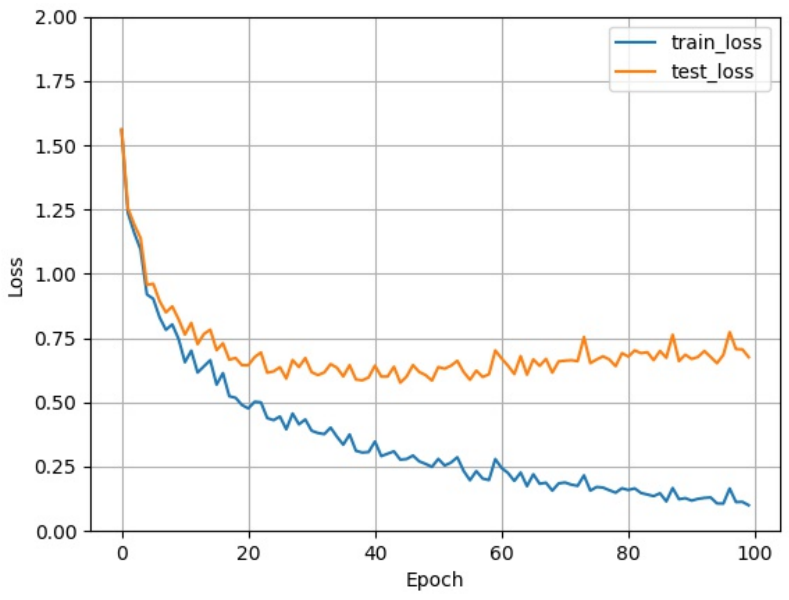


(a) Training and test loss.

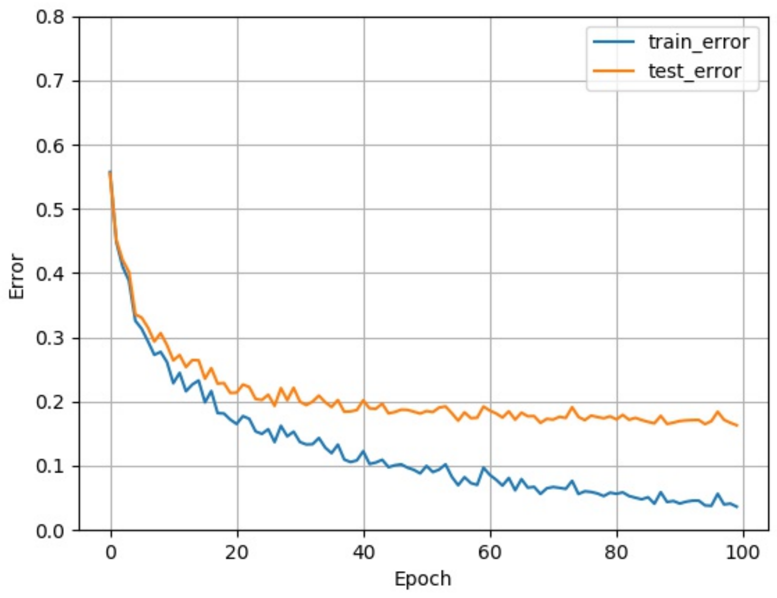


(b) Training and test error.

Figure 6.3.: Best performance of SSGD on CIFAR-10 (seed 2).



(a) Training and test loss.



(b) Training and test error.

Figure 6.4.: Best performance of ADAM on CIFAR-10 (seed 6).

6.4. Our new training techniques

The basic information reported for each new technique is the same as for state-of-art algorithms: hyper-parameter settings, aggregated tests results, and a training example. Moreover, observations on these data may lead to the formulation of hypotheses on the algorithms behavior. The validity of such hypotheses is verified with additional tests, whose results are reported.

6.4.1. SGD-SA results

SGD-SA requires two hyper-parameters: the cooling factor and the initial acceptance probability used in the temperature initialization procedure. For the tests, the cooling factor is set to 0.9, while the initial acceptance probability is set to 0.5. The role of these parameters is to manage the SA process, hence they are required for every technique based on SA: SGD-SA, SGD-SA-R, SGD-SA-LR, and SGD-SA-DB. If not stated differently, they are implicitly initialized with the same values mentioned above.

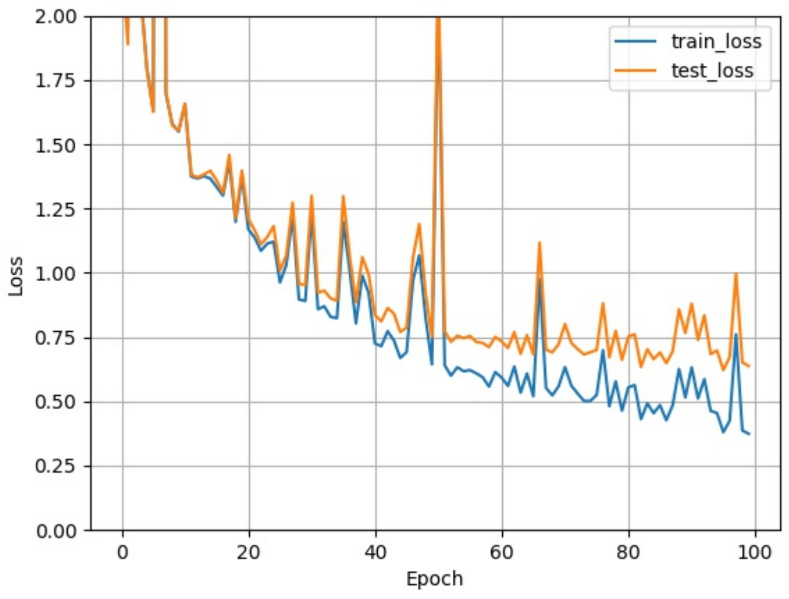
The test results for SGD-SA are reported in Table 6.9. The performance, in terms of error, is significatively worse than state-of-art techniques, whereas, in terms of loss, it is comparable.
The overfitting is however very low, indicating that the diversification procedures improved the generalizability of the solutions.

An example of training is displayed in Figure 6.5, while the information on the related SA process is displayed in Figure 6.6. The number of worsening moves is very low during all the training and the majority of them are accepted. This implies that almost all moves are accepted, hence the SA process has minimal influence on the performance of the algorithm.

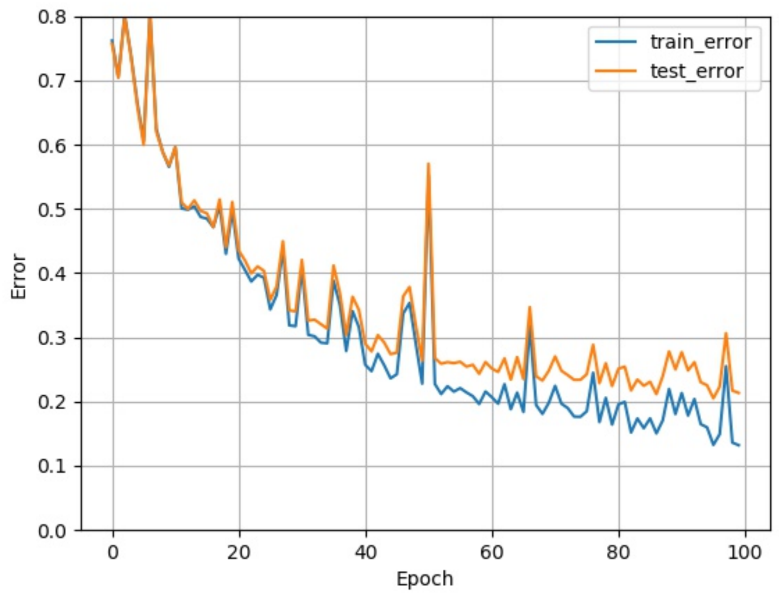
The algorithm performance does not depend on the SA process, so there must be another element that justifies it. We will analyze the randomness introduced in the learning rate choice and the ability of the algorithm to reject moves, aiming to discover which one leads to the observed behavior of SGD-SA.

	Loss			Error		
Dataset	Min	Mean	Over	Min	Mean	Over
MNIST	0.03328	0.03787	0.01528	0.01040	0.01245	0.004822
F-MNIST	0.03134	0.03732	0.01227	0.01090	0.01225	0.003859
CIFAR-10	0.6222	0.7767	0.2404	0.2135	0.2362	0.06733

Table 6.9.: Aggregated test-set results for SGD-SA.



(a) Training and test loss.



(b) Training and test error.

Figure 6.5.: Best performance of SGD-SA on CIFAR-10 (seed 4).

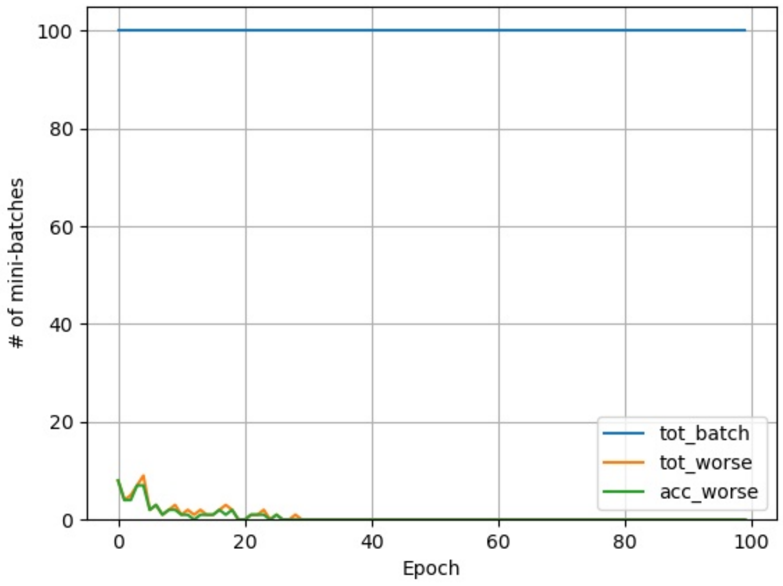


Figure 6.6.: Acceptance tests results for SGD-SA on CIFAR-10 (seed 4).

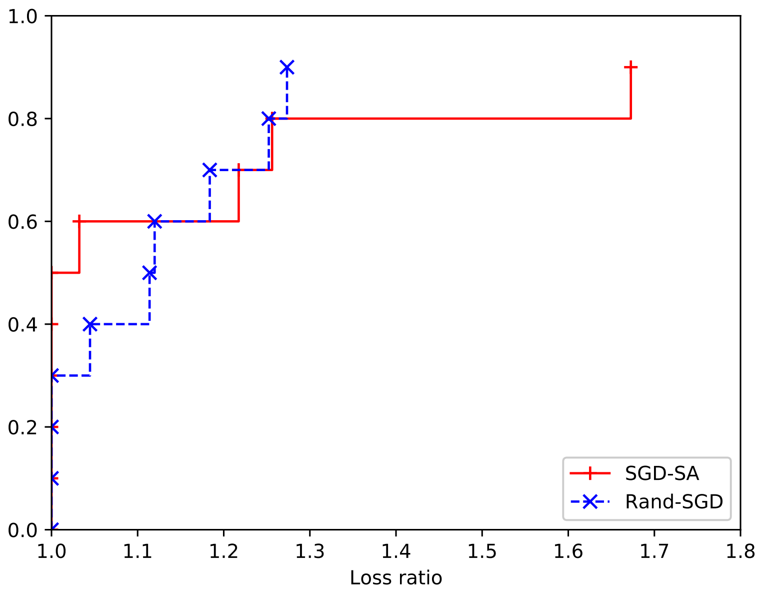
Randomness analysis

This test aims to verify if the performance observed in SGD-SA is due to the randomness introduced in the choice of the learning rate. This hypothesis is analyzed by comparing the performance of SGD-SA with a randomized version of SGD – a variant of SGD-SA that does not perform any SA related mechanics but only the randomization of the learning rate.

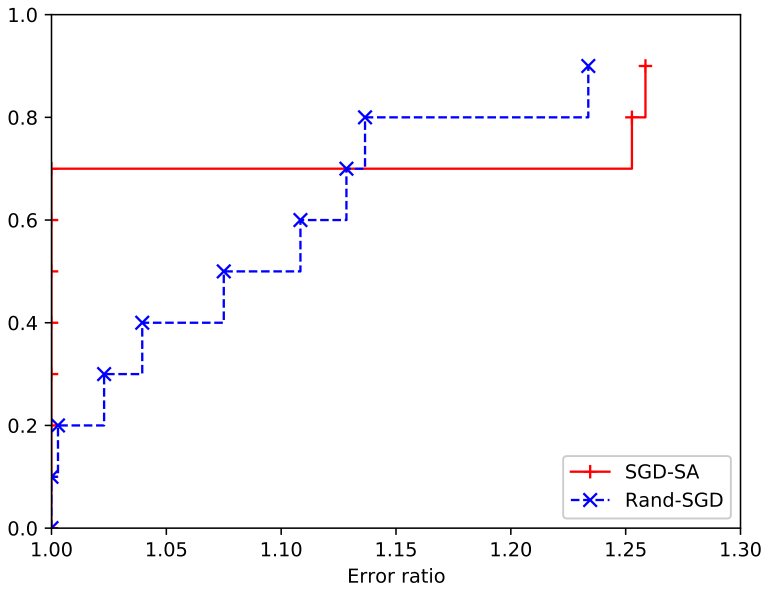
The test results for randomized SGD are displayed in Table 6.10, while the performance profile comparison with the original SGD-SA is reported in Figure 6.7. The results show that SGD-SA outperforms randomized SGD, however their performance is comparable. In conclusion, it is reasonable to assume that the randomness is not the main cause of the performance of SGD-SA, even if it has an important influence on SGD-SA.

	Loss			Error		
Optimizer	Min	Mean	Over	Min	Mean	Over
Rand-SGD	0.6026	0.7592	0.2526	0.2003	0.2420	0.06835
SGD-SA	0.6222	0.7767	0.2404	0.2135	0.2362	0.06733

Table 6.10.: Aggregated test-set results for randomized SGD and SGD-SA on CIFAR-10.



(a) Loss comparison.



(b) Error comparison.

Figure 6.7.: Performance profile comparison: SGD-SA vs. randomized SGD.

Temperature analysis

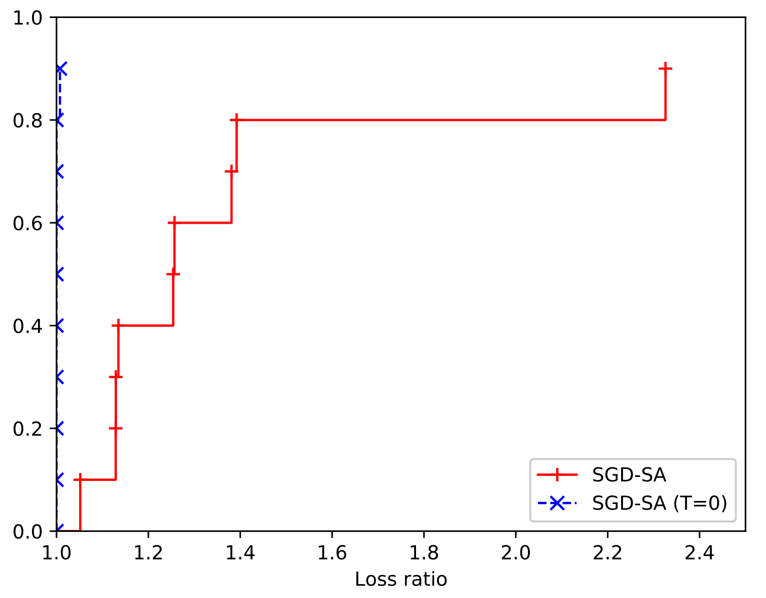
This test aims to verify if the performance observed in SGD-SA is due to the solution rejection feature, introduced by the SA acceptance test. This hypothesis is analyzed by comparing the performance of SGD-SA with a modified version of SGD-SA, which reject all worsening moves. Such algorithm can be easily obtained by setting the initial temperature of SGD-SA very close to 0 (we used 10^{-8}).

The test results for the variant of SGD-SA are reported in Table 6.11, while the performance profile comparison with the original SGD-SA is reported in Figure 6.8. The results show that the variant of SGD-SA with null temperature greatly outperforms SGD-SA. This confirms that the performance of SGD-SA is caused by its ability of rejecting moves, independently from the overall SA process implemented.

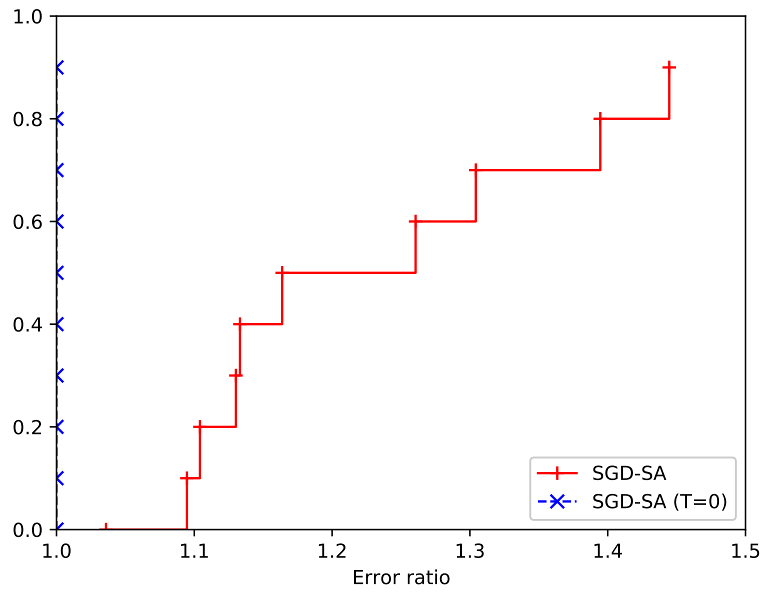
Due to the significance of these results, an example of training for SGD-SA with null temperature is reported in Figure 6.9, while the information on the related SA process is reported in Figure 6.10.

	Loss			Error		
Optimizer	Min	Mean	Over	Min	Mean	Over
SGD-SA (T=0)	0.5482	0.6010	0.2556	0.1819	0.1965	0.07461
SGD-SA	0.6222	0.7767	0.2404	0.2135	0.2362	0.06733

Table 6.11.: Aggregated test-set results for SGD-SA with and without null temperature on CIFAR-10.

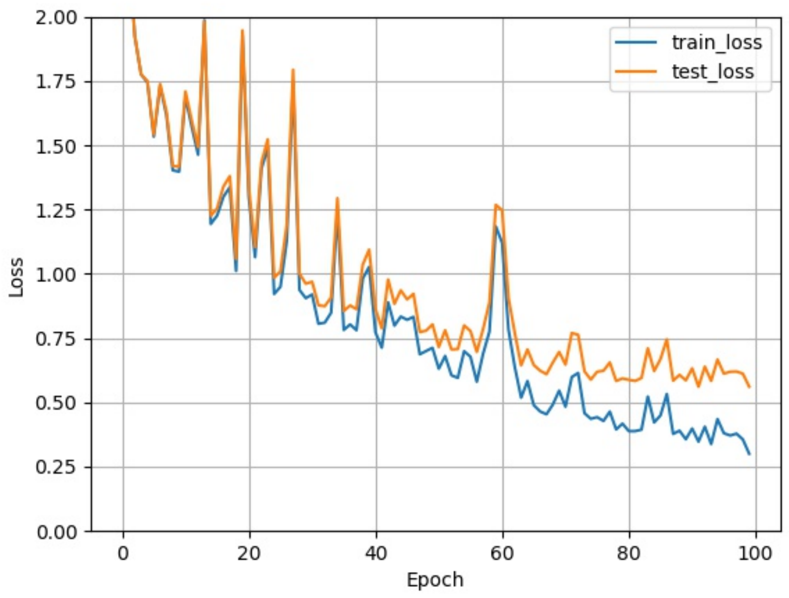


(a) Loss comparison.

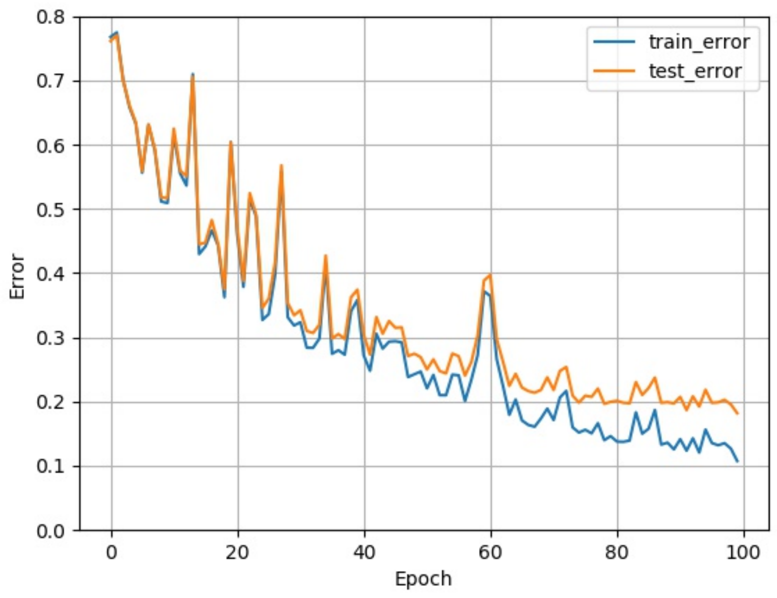


(b) Error comparison.

Figure 6.8.: Performance profile comparison: SGD-SA vs. SGD-SA with null temperature.



(a) Training and test loss.



(b) Training and test error.

Figure 6.9.: Best performance of SGD-SA with null temperature on CIFAR-10 (seed 4).

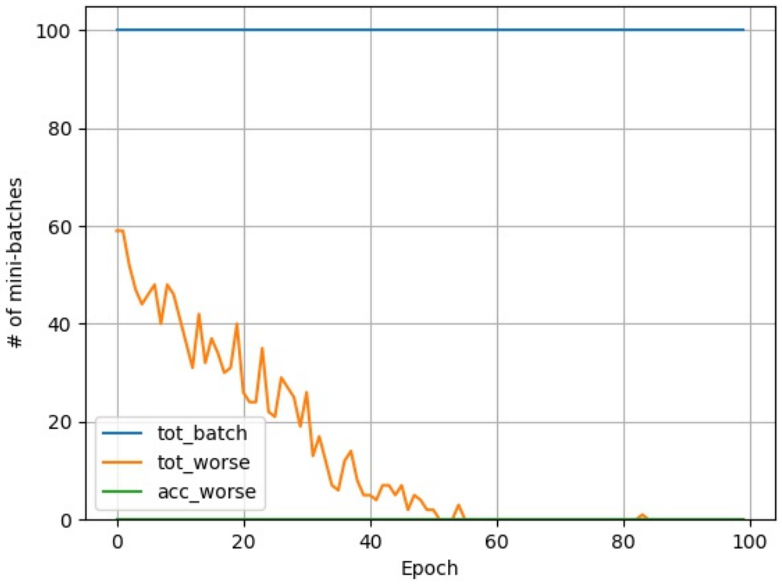


Figure 6.10.: Acceptance tests results for SGD-SA with null temperature on CIFAR-10 (seed 5).

6.4.2. SGD-SA-R results

SGD-SA-R requires only one hyper-parameter to be set: the norm function used as cost function. For the tests, both the L1 and the L2 norms are used, due to their effectiveness in the regularization.

The test results for SGD-SA-R with L1 and L2 norm are reported in Table 6.12 and Table 6.13, respectively. The performance, for both norms, is significantly worse than state-of-art techniques. On the other hand, the overfitting is very low, indicating that the independence between the solution generation and the cost function is an effective diversification strategy.

An example of training, for both norms, is shown in Figure 6.11 and 6.12, while the information on the related SA process is shown in Figure 6.13 and 6.14. Both examples suggest that the SA procedure is executed as desired: the number of worsening solutions is large and increases moderately during the training, indicating that the network is getting more complex to better fit the training set. At the same time, the fraction of worsening moves accepted is very high at the beginning, then it gradually decreases during the training.

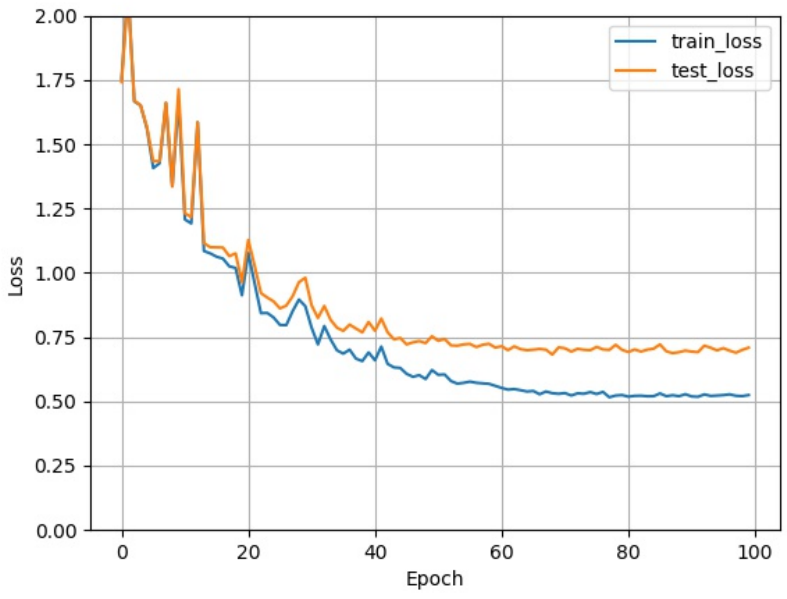
The large number of rejected moves, especially in the second half of the training, might have reduced the ability of the algorithm to reach further improving solutions, resulting in a decreased performance. Therefore, the algorithm could be improved by increasing the number of accepted solutions. This can be achieved by increasing the temperature of the system during the training either by setting a higher initial temperature or by setting a cooling factor closer to 1. Both these strategies are tested below.

	Loss			Error		
Dataset	Min	Mean	Over	Min	Mean	Over
MNIST	0.03218	0.03782	0.01195	0.01060	0.01236	0.003890
F-MNIST	0.03279	0.03790	0.01212	0.01060	0.01275	0.004166
CIFAR-10	0.7084	0.8254	0.1626	0.2370	0.2609	0.04798

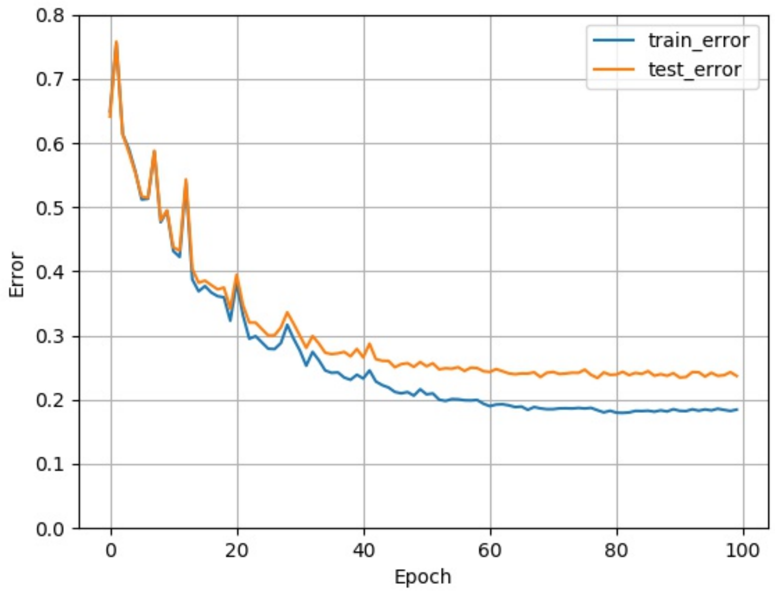
Table 6.12.: Aggregated test-set results for SGD-SA-R with L1 norm.

	Loss			Error		
Dataset	Min	Mean	Over	Min	Mean	Over
MNIST	0.03134	0.04476	0.009792	0.00960	0.01365	0.003418
F-MNIST	0.03186	0.03661	0.009729	0.00990	0.01202	0.003474
CIFAR-10	0.7029	0.8237	0.1857	0.2314	0.2548	0.06008

Table 6.13.: Aggregated test-set results for SGD-SA-R with L2 norm.

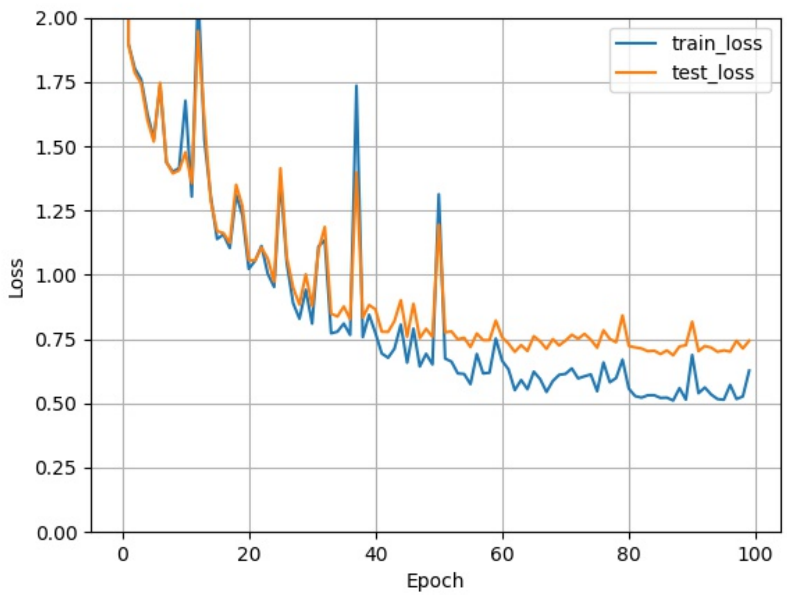


(a) Training and test loss.

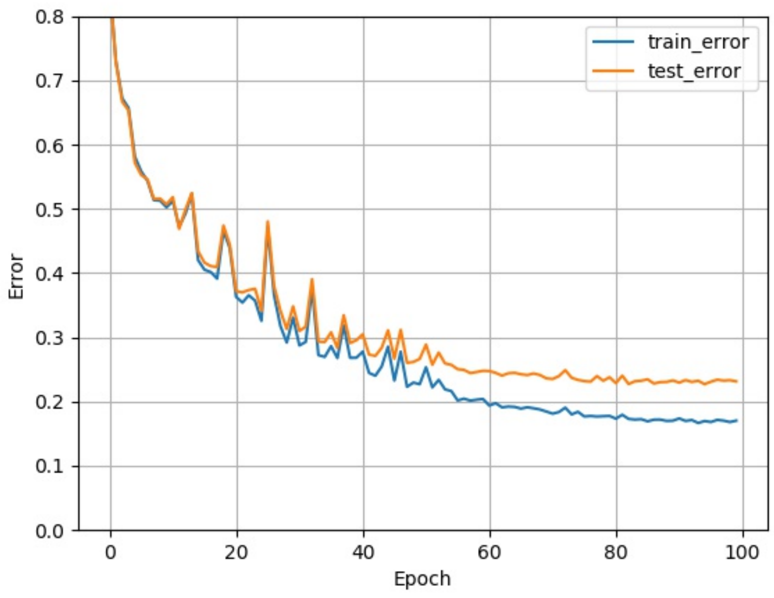


(b) Training and test error.

Figure 6.11.: Best performance of SGD-SA-R with L1 norm on CIFAR-10 (seed 1).



(a) Training and test loss.



(b) Training and test error.

Figure 6.12.: Best performance of SGD-SA-R with L2 norm on CIFAR-10 (seed 6).

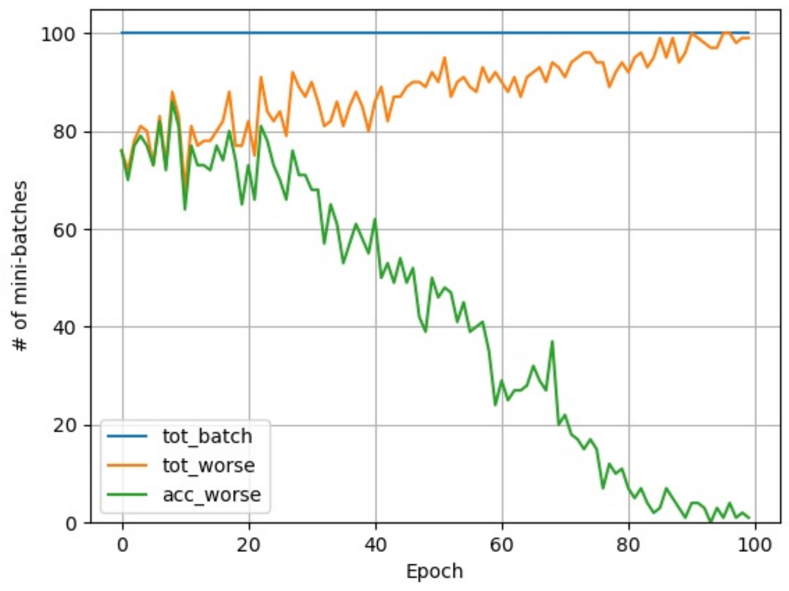


Figure 6.13.: Acceptance tests results for SGD-SA-R with L1 norm on CIFAR-10 (seed 1).

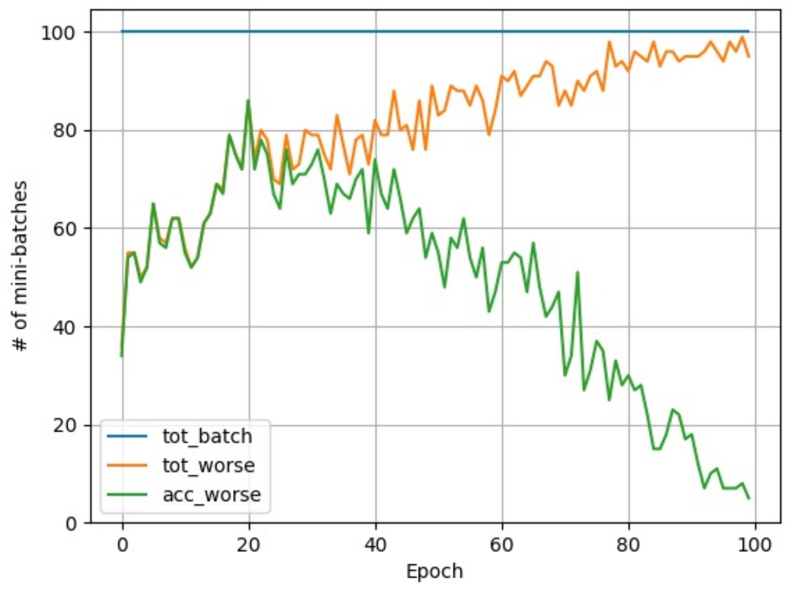


Figure 6.14.: Acceptance tests results for SGD-SA-R with L2 norm on CIFAR-10 (seed 6).

High temperature analysis

This test aims to verify if a higher temperature during the training, due to a higher initial temperature, can improve SGD-SA-R. This hypothesis is analyzed by comparing the performance of SGD-SA-R with the default temperature and with a higher temperature. The temperature is raised indirectly by modifying the initial acceptance probability from 0.5 to 0.9.

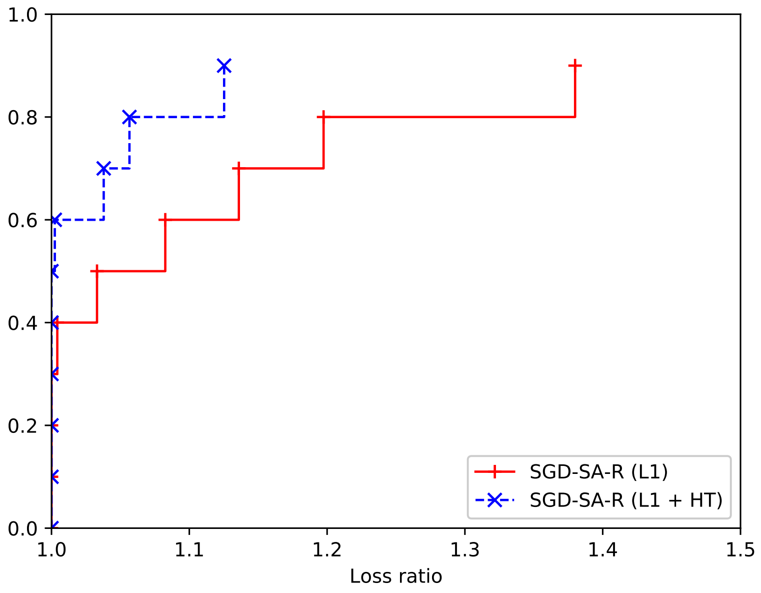
The test results for SGD-SA-R with high temperature are shown in Table 6.14 and Table 6.15, while the performance profile comparison with the SGD-SA-R with normal temperature is reported in Figure 6.15 and Figure 6.16. The results show that increasing the initial temperature improves the performance of SGD-SA-R, confirming our initial hypothesis.

		Loss			Error	
Optimizer	Min	Mean	Over	Min	Mean	Over
SGD-SA-R (L1+HT)	0.6688	0.7822	0.2196	0.2150	0.2493	0.06220
SGD-SA-R (L1)	0.7084	0.8254	0.1626	0.2370	0.2609	0.04798

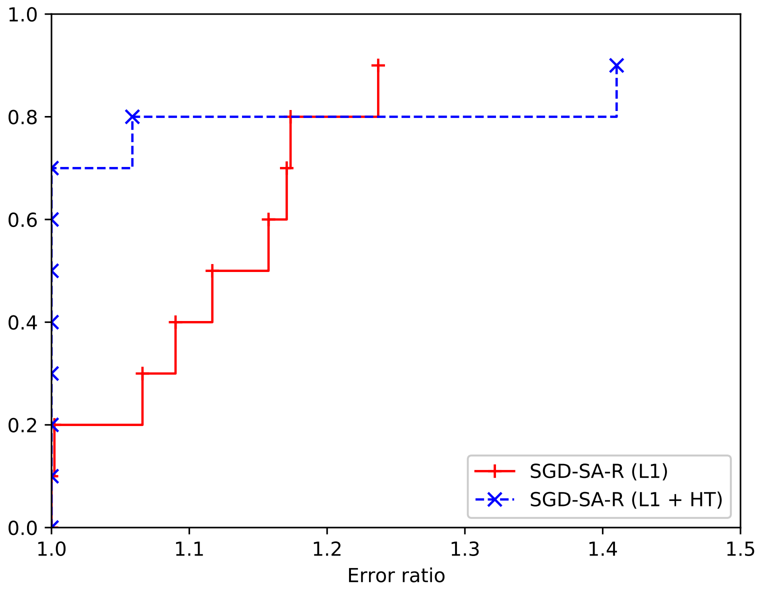
Table 6.14.: Aggregated test-set results for SGD-SA-R (L1 norm) with and without high temperature on CIFAR-10.

		Loss			Error	
Optimizer	Min	Mean	Over	Min	Mean	Over
SGD-SA-R (L2+HT)	0.6534	0.7899	0.2439	0.2182	0.2317	0.07266
SGD-SA-R (L2)	0.7029	0.8237	0.1857	0.2314	0.2548	0.06008

Table 6.15.: Aggregated test-set results for SGD-SA-R (L2 norm) with and without high temperature on CIFAR-10.

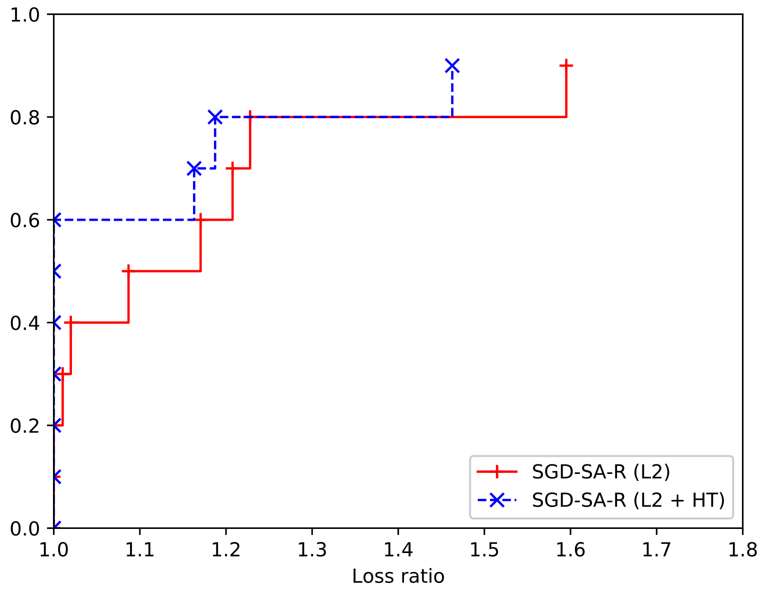


(a) Loss comparison.

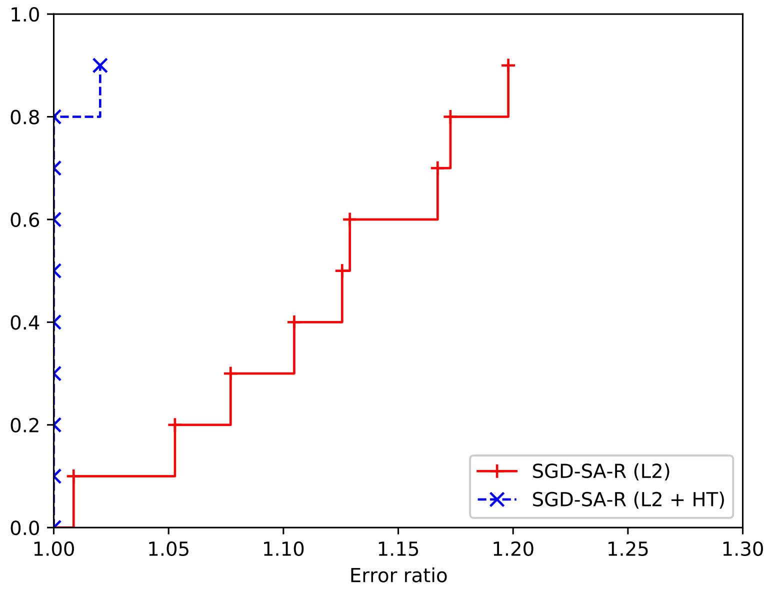


(b) Error comparison.

Figure 6.15.: Performance profile comparison: SGD-SA-R with L1 norm vs. SGD-SA-R with L1 norm and high temperature.



(a) Loss comparison.



(b) Error comparison.

Figure 6.16.: Performance profile comparison: SGD-SA-R with L2 norm vs. SGD-SA-R with L2 norm and high temperature.

Slow cooldown analysis

This test aims to verify if a higher temperature during the training, due to a slower cooling, can improve SGD-SA-R. This hypothesis is analyzed by comparing the performance of SGD-SA-R with the default temperature and with a higher temperature. The temperature is raised indirectly by modifying the cooling factor from 0.9 to 0.95.

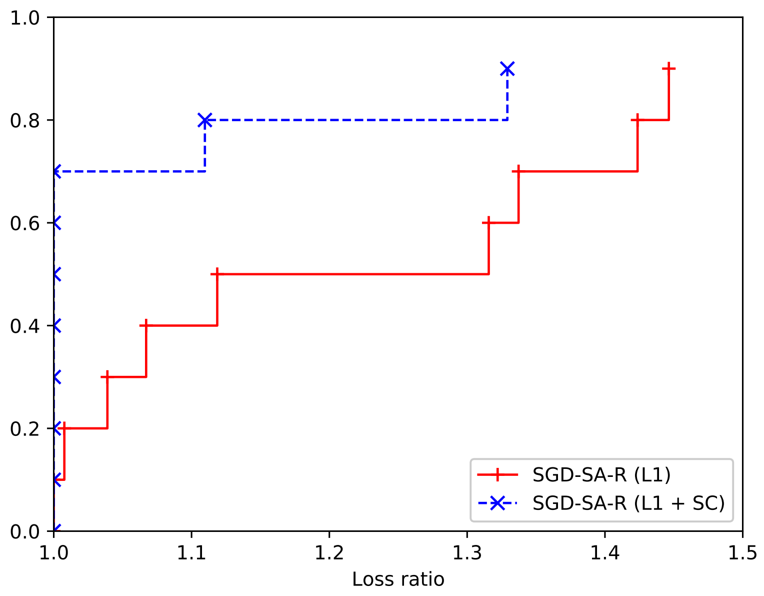
The test results for SGD-SA-R with slow cool down are displayed in Table 6.16 and Table 6.17, while the performance profile comparison with the SGD-SA-R with normal temperature is reported in Figure 6.17 and Figure 6.18. The results show that slowing down the cooling of the system improves the performance of SGD-SA-R, confirming the initial hypothesis.

	Loss			Error		
Optimizer	Min	Mean	Over	Min	Mean	Over
SGD-SA-R (L1+SC)	0.6454	0.7392	0.2656	0.2108	0.2257	0.07515
SGD-SA-R (L1)	0.7084	0.8254	0.1626	0.2370	0.2609	0.04798

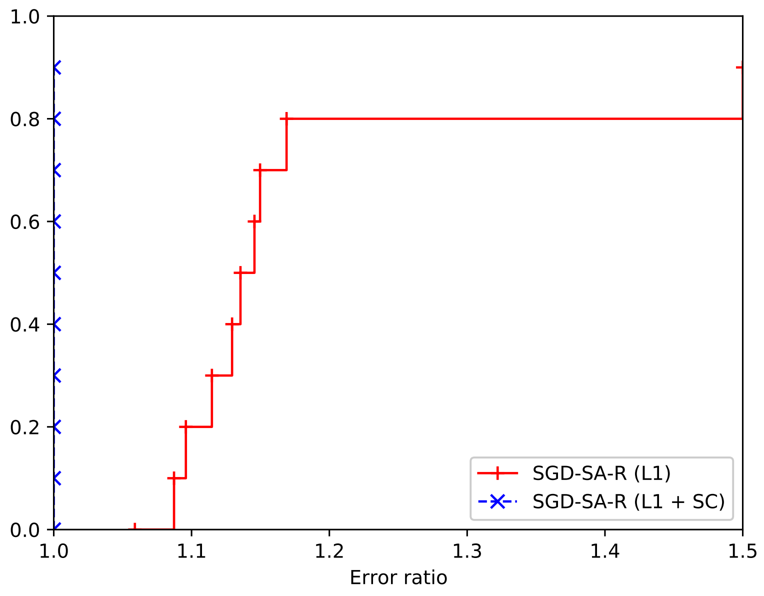
Table 6.16.: Aggregated test-set results for SGD-SA-R (L1 norm) with and without slow cool down on CIFAR-10.

	Loss			Error		
Optimizer	Min	Mean	Over	Min	Mean	Over
SGD-SA-R (L2+SC)	0.6315	0.6880	0.2516	0.2100	0.2268	0.07463
SGD-SA-R (L2)	0.7029	0.8237	0.1857	0.2314	0.2548	0.06008

Table 6.17.: Aggregated test-set results for SGD-SA-R (L2 norm) with and without slow cool down on CIFAR-10.

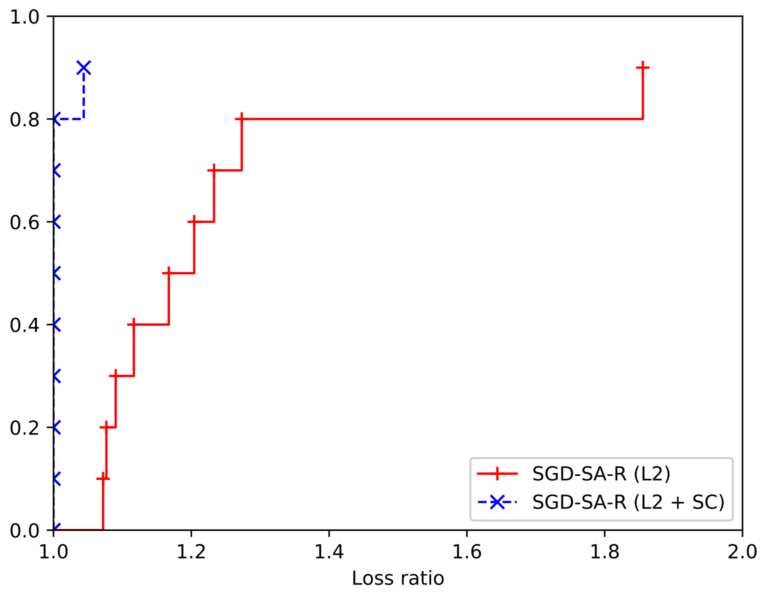


(a) Loss comparison.

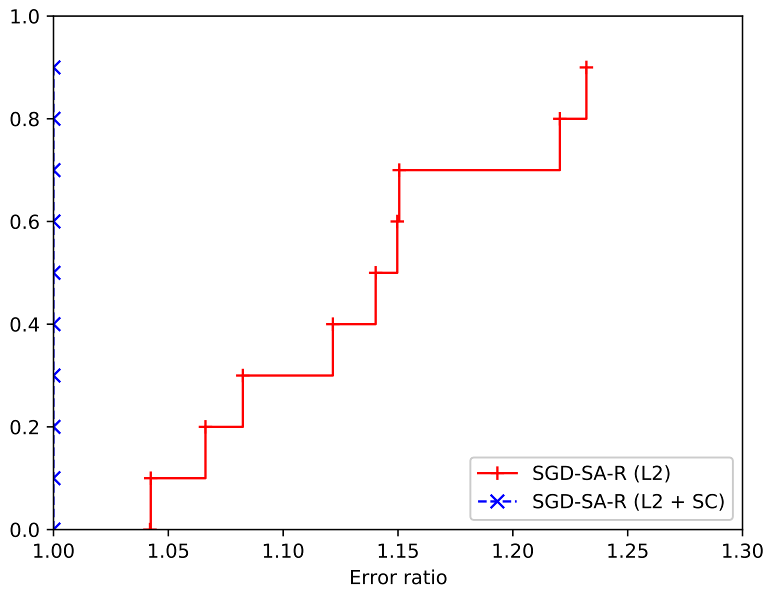


(b) Error comparison.

Figure 6.17.: Performance profile comparison: SGD-SA-R with L1 norm vs. SGD-SA-R with L1 norm and slow cooldown.



(a) Loss comparison.



(b) Error comparison.

Figure 6.18.: Performance profile comparison: SGD-SA-R with L2 norm vs. SGD-SA-R with L2 norm and slow cooldown.

6.4.3. SGD-SA-LR results

SGD-SA-LR requires two hyper-parameters to be set: the norm function used in the cost function and the lambda coefficient. For the tests, both the L1 and L2 norms are used, due to their effectiveness in the regularization, whereas lambda is set to 0.1, as empirically suggested by a simple parameter tuning.

The test results for SGD-SA-LR with L1 and L2 norms are reported in Table 6.18 and Table 6.19, respectively. The performance, in terms of loss, outperforms the state-of-art techniques, whereas, in terms of error, it is comparable. The overfitting is very low, indicating that the use of the loss both in the generation of solutions and in the computation of the cost does not significantly reduce the effectiveness of the diversification.

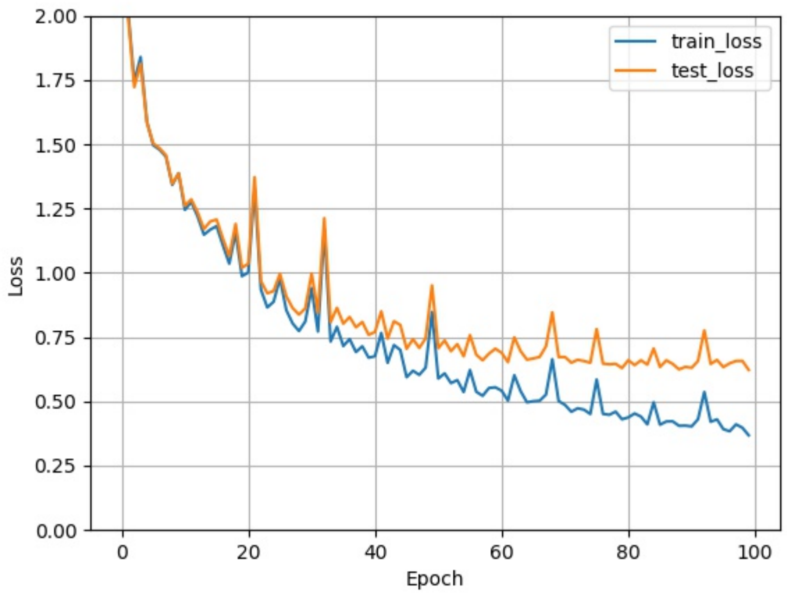
An example of training, for both norms, is displayed in Figure 6.19 and 6.20, while the information on the related SA process is displayed in Figure 6.21 and 6.22. Similarly to SGD-SA, the number of worsening moves is low, implying that the majority of moves is automatically accepted. On the other hand, the number of accepted solutions among the worsening moves decreases during the training. In conclusion, the observations above indicate that the SA mechanism has a secondary role on the performance of the algorithm; nevertheless, the moves rejection still has a crucial impact on the results.

	Loss			Error		
Dataset	Min	Mean	Over	Min	Mean	Over
MNIST	0.03185	0.03692	0.01425	0.01070	0.01183	0.004276
F-MNIST	0.03191	0.03674	0.01521	0.00980	0.01202	0.004769
CIFAR-10	0.6226	0.7056	0.2884	0.2026	0.2264	0.08464

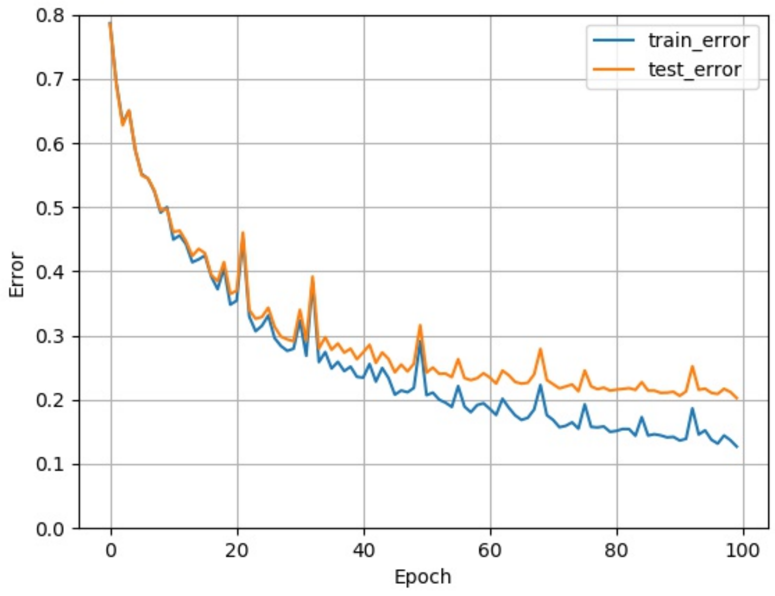
Table 6.18.: Aggregated test-set results for SGD-SA-LR with L1 norm.

	Loss			Error		
Dataset	Min	Mean	Over	Min	Mean	Over
MNIST	0.03219	0.03815	0.01361	0.01100	0.01249	0.004369
F-MNIST	0.03063	0.03718	0.01422	0.00940	0.01202	0.004317
CIFAR-10	0.6051	0.6718	0.2557	0.1972	0.2200	0.07434

Table 6.19.: Aggregated test-set results for SGD-SA-LR with L2 norm.

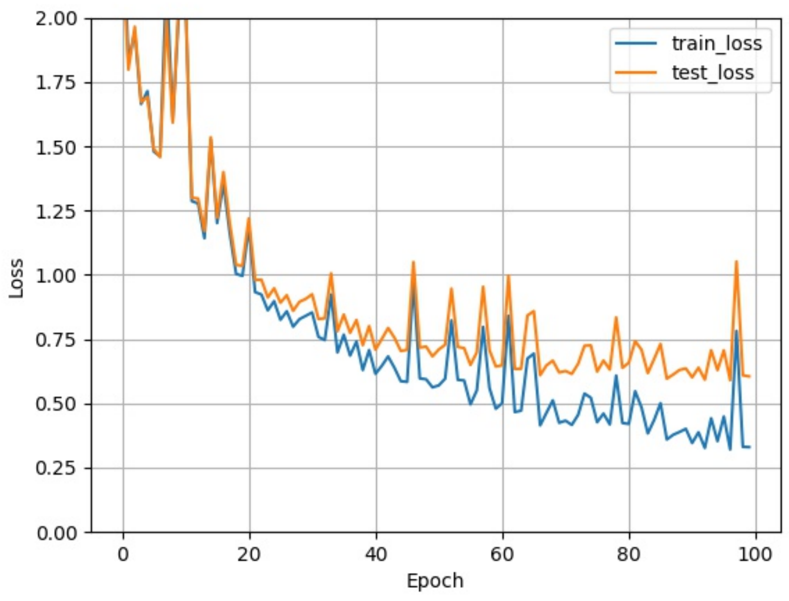


(a) Training and test loss.

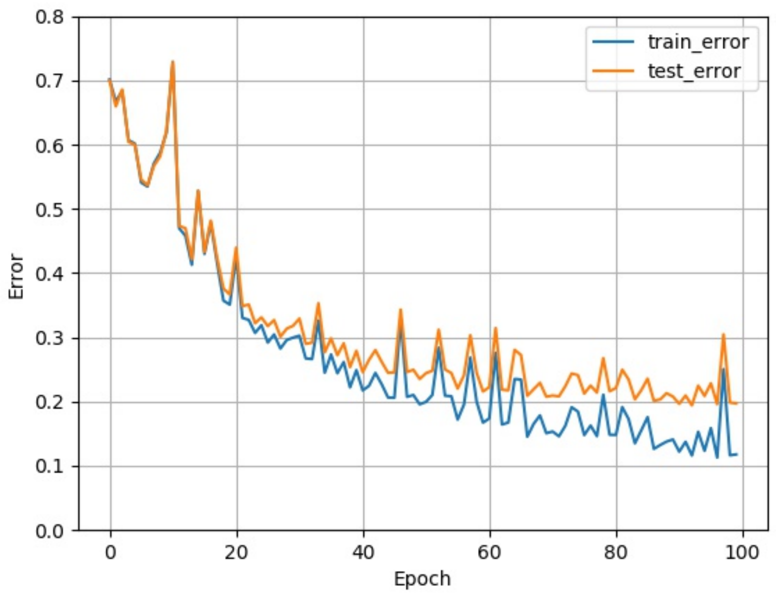


(b) Training and test error.

Figure 6.19.: Best performance of SGD-SA-LR with L1 norm on CIFAR-10 (seed 7).



(a) Training and test loss.



(b) Training and test error.

Figure 6.20.: Best performance of SGD-SA-LR with L2 norm on CIFAR-10 (seed 1).

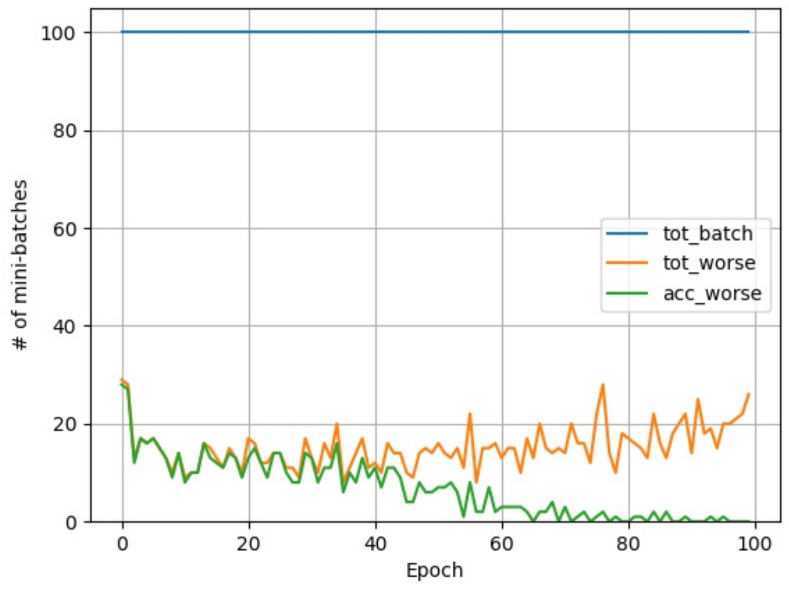


Figure 6.21.: Acceptance tests results for SGD-SA-LR with L1 norm on CIFAR-10 (seed 7).

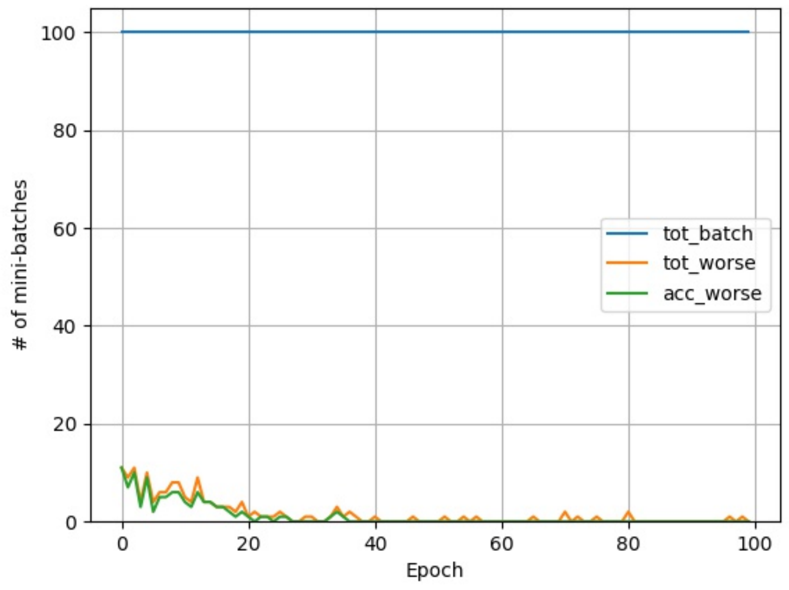


Figure 6.22.: Acceptance tests results for SGD-SA-LR with L2 norm on CIFAR-10 (seed 1).

6.4.4. SGD-SA-DB results

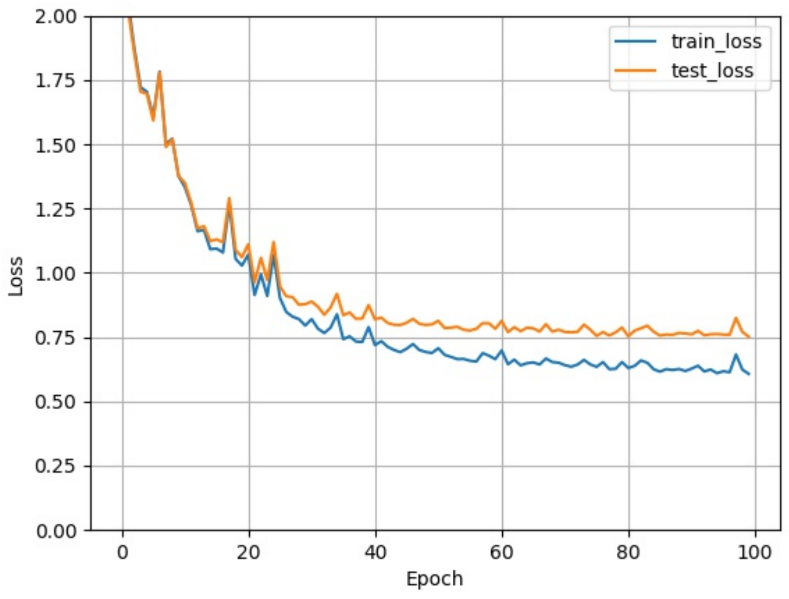
SGD-SA-DB does not require any hyper-parameter to be set, except for SA related ones, which are implicitly defined as for SGD-SA.

The test results for SGD-SA-DB are reported in Table 6.20. The performance is much worse than state-of-art techniques. On the other hand, the overfitting is very low, indicating that the idea of using two different mini-batches for each move is effective for the diversification.

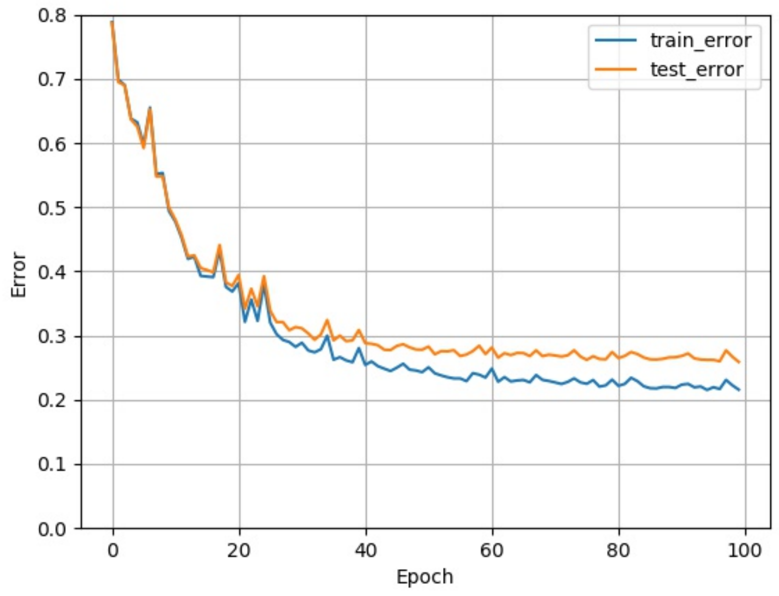
An example of training is displayed in Figure 6.23, while the information on the related SA process is displayed in Figure 6.24. The analysis of the SA process suggests an explanation for the poor performance of the technique. At the beginning of the training the number of worsening moves is low, but almost all of them are accepted. As the training proceeds, the number of worsening moves increases steeply, whereas the number of accepted ones gradually decreases. Consequently, the number of moves performed drops quickly, reducing the ability of the algorithm to find better solutions. The plot suggests that the large variation in the number of worsening moves cannot be captured effectively in the initialization of SA. Hence, after some epochs, the acceptance test loses effectiveness, since it is not suitable for the generated moves anymore.

	Loss			Error		
Dataset	Min	Mean	Over	Min	Mean	Over
MNIST	0.03490	0.03972	0.01461	0.01130	0.01297	0.004749
F-MNIST	0.03225	0.03662	0.01228	0.00980	0.01228	0.004169
CIFAR-10	0.7515	0.9099	0.1207	0.2589	0.2987	0.03864

Table 6.20.: Aggregated test-set results for SGD-SA-DB.



(a) Training and test loss.



(b) Training and test error.

Figure 6.23.: Best performance of SGD-SA-DB on CIFAR-10 (seed 2).

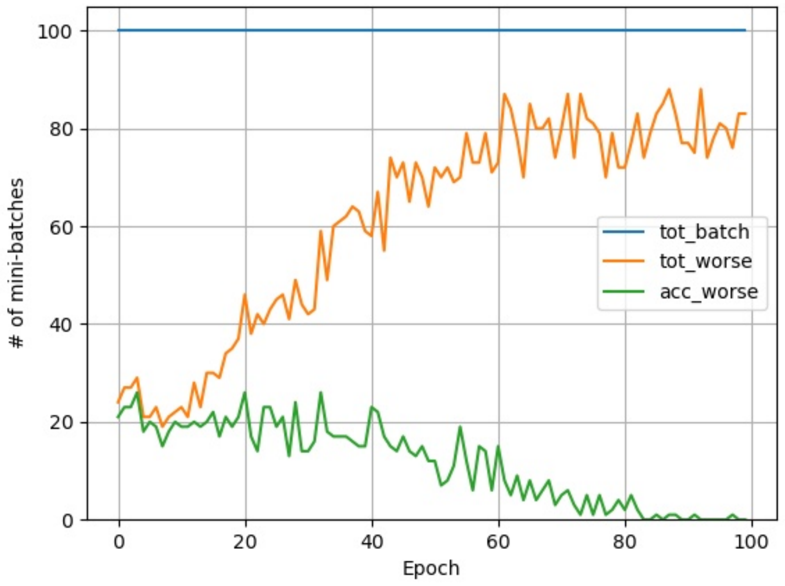


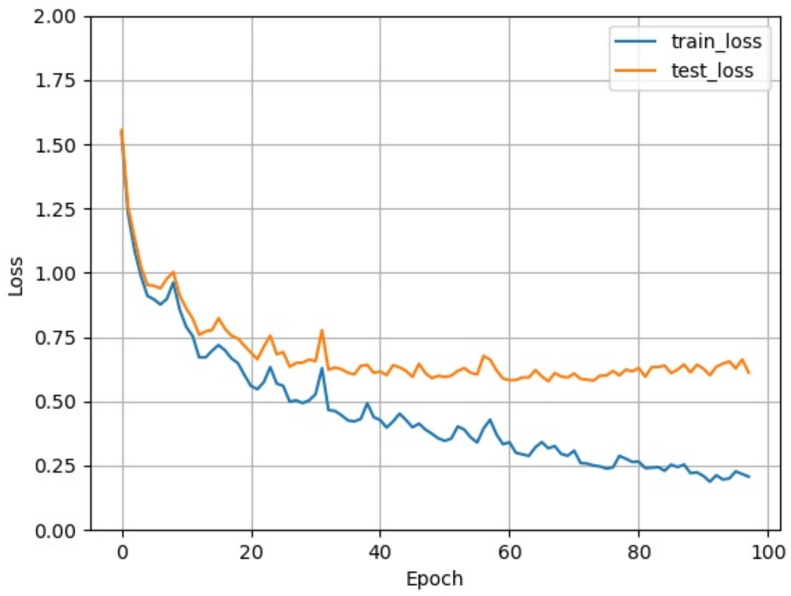
Figure 6.24.: Acceptance tests results for SGD-SA-DB on CIFAR-10 (seed 2).

6.4.5. ADAM-VNS results

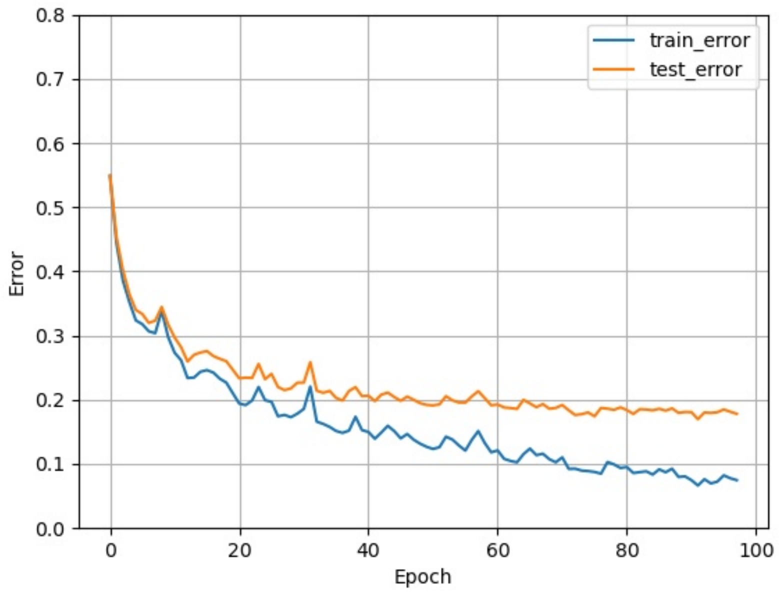
ADAM-VNS requres only two hyper-parameters to be set: the basic standard deviation of the noise and the threshold indicating the minimum valid improvement of a solution. For the tests, the standard deviation of the noise is set to 0.05, while the improvement threshold is set to 0.05, which means that a solution is considered a local optimum if in the last 5 epochs improved less than 5%. The values of these parameters were suggested empirically by the results of our parameter tuning.

The test results for ADAM-VNS are reported in Table 6.21. The performance is competitive with state-of-art techniques: in terms of loss, it outperforms them, whereas, in terms of error, it is comparable. The overfitting is low, indicating that the diversification procedures introduced by the VNS framework are effective in improving the generalizability.

An example of training is reported in Figure 6.25, while the information on the related noise applied is reported in Figure 6.26. The implementation of ADAM-VNS implicitly introduces a mechanism similar to early stopping, whose purpose is to reduce the overfitting. When a solution does not show any noticeable improvement, the intensification procedure stops and VNS diversification procedure is applied. The analysis of the noise suggests that this procedure is used many times during the ADAM-VNS training, hence it is likely to reduce significantly the overfitting.



(a) Training and test loss.



(b) Training and test error.

Figure 6.25.: Best performance of ADAM-VNS on CIFAR-10 (seed 9).

	Loss			Error		
Dataset	Min	Mean	Over	Min	Mean	Over
MNIST	0.02783	0.03442	0.01628	0.00770	0.00993	0.00378
F-MNIST	0.03220	0.03673	0.01765	0.00870	0.01060	0.00428
CIFAR-10	0.5731	0.6048	0.3716	0.1697	0.1789	0.09574

Table 6.21.: Aggregated test-set results for ADAM-VNS.

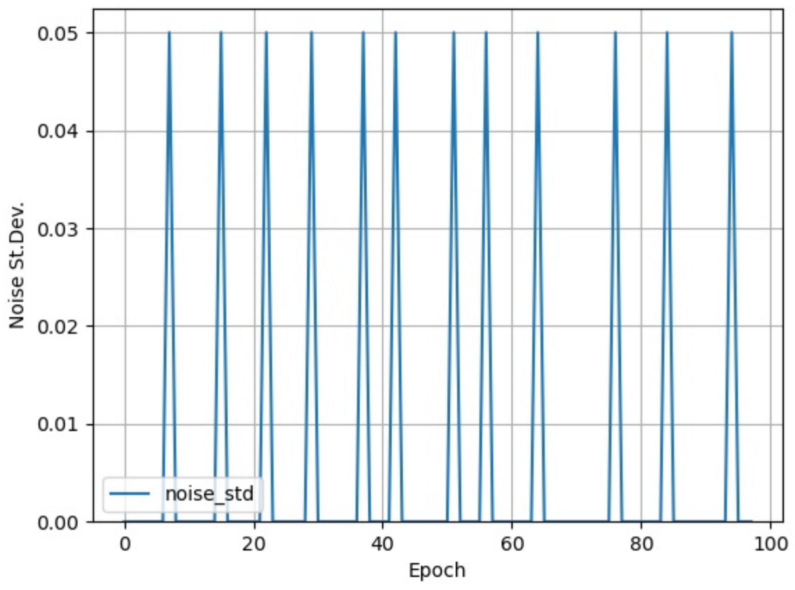


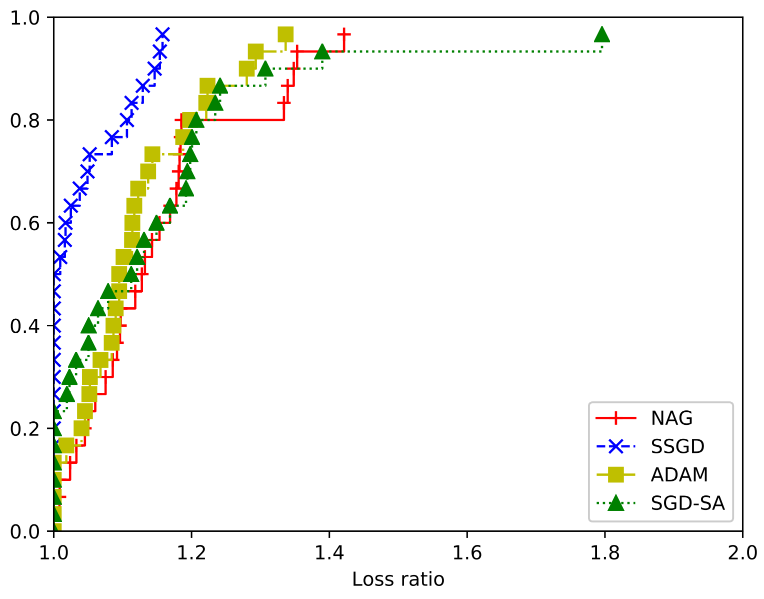
Figure 6.26.: Intensity of the perturbations for ADAM-VNS on CIFAR-10 (seed 9).

6.5. Comparison between techniques

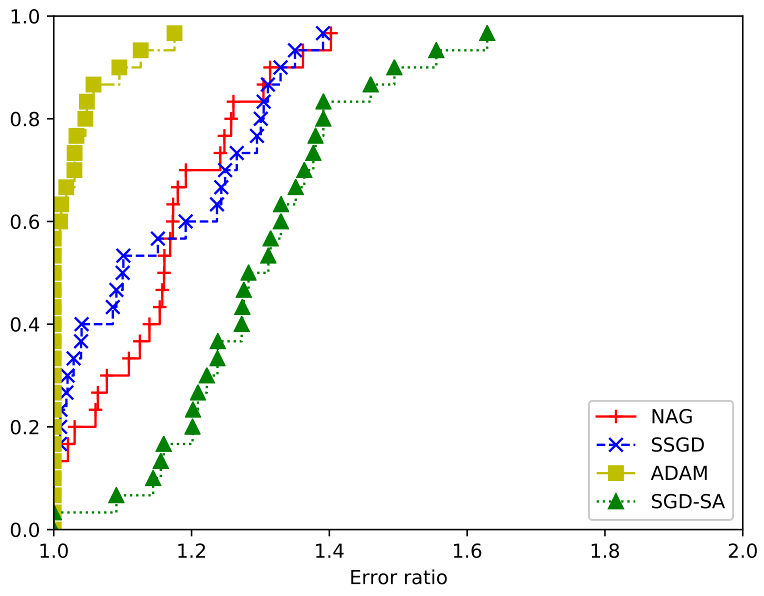
The performances of the techniques, separately described above, are compared here by means of performance profile. The data used to compute the performance profile is the information provided by the tests on all the datasets. The comparisons are performed between the state-of-art techniques and every other new technique that we introduced, one by one, then only between all the SA-based techniques. Specifically, the properties analyzed by the performance profile are: the loss, the error, the overfitting on the loss, and the overfitting on the error. The overfitting is here measured as the difference between the test and the training values, similarly to the over function, which in addition computes the mean. With a little abuse of notation, the overfitting is denoted as the over function in the figures, however no average operation is performed.

6.5.1. Comparison of state-of-art techniques versus SGD-SA

The comparison between SGD-SA and state-of-art techniques is shown in Figure 6.27 and Figure 6.28. In terms of loss, SGD-SA shows competitive results, performing similarly to ADAM and NAG; whereas, in terms of error, it performs significantly worse than state-of-art techniques. SGD-SA shows a much smaller overfitting with respect to state-of art techniques. This proves that the move rejection and the learning rate randomization do improve the generalizability of the results.

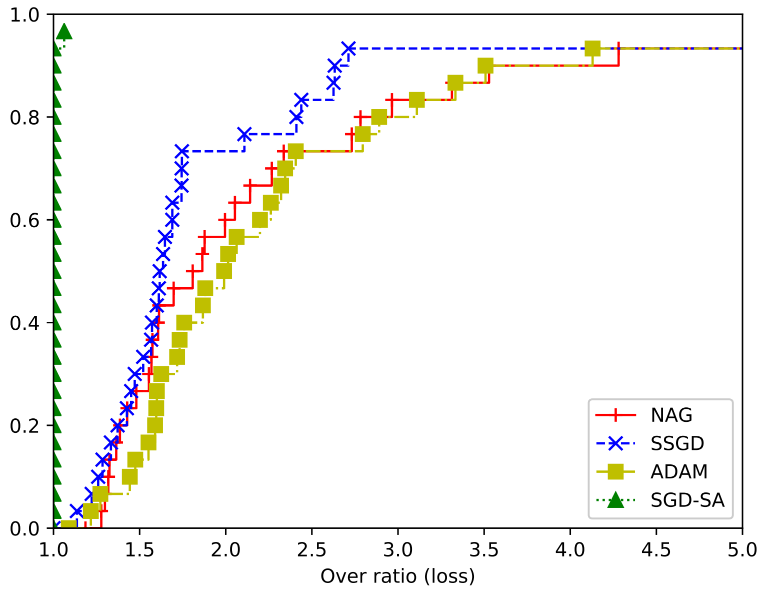


(a) Loss comparison.

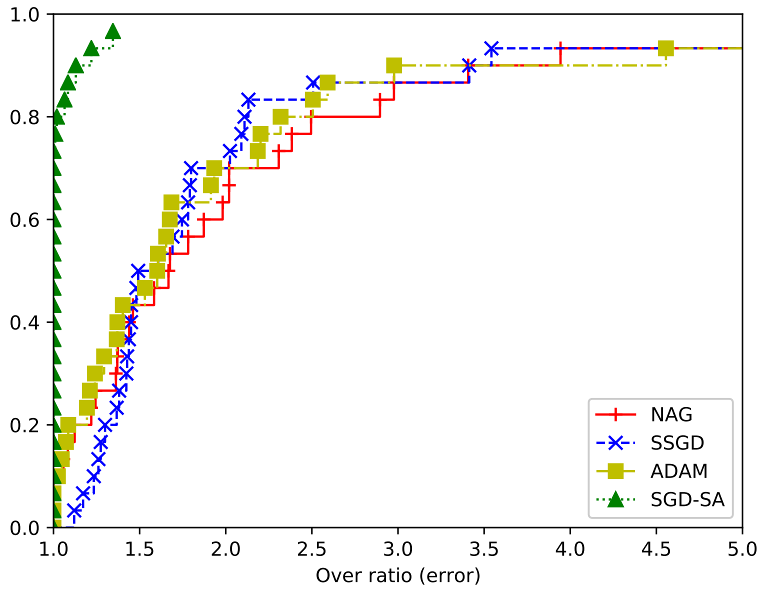


(b) Error comparison.

Figure 6.27.: Performance profile comparison over the test set: NAG vs. SSGD vs. ADAM vs. SGD-SA.



(a) Overfitting on loss comparison.



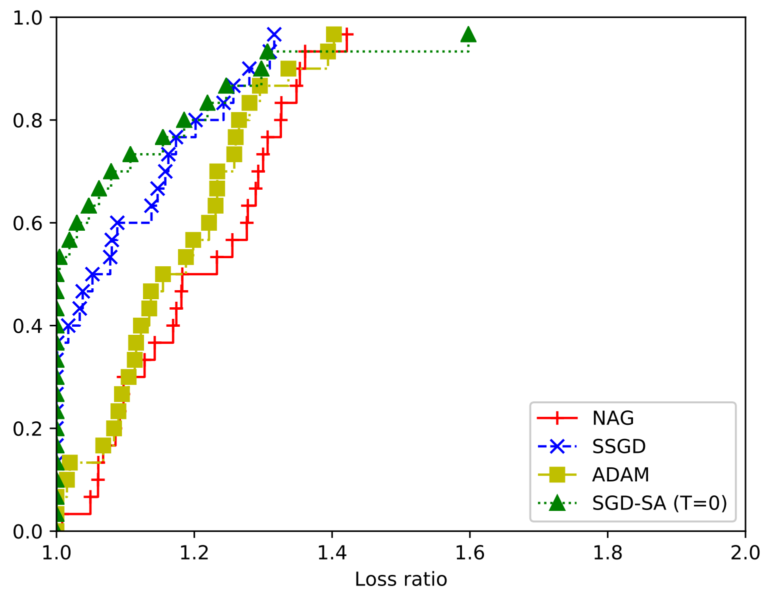
(b) Overfitting on error comparison.

Figure 6.28.: Performance profile comparison for overfitting: NAG vs. SSGD vs. ADAM vs. SGD-SA.

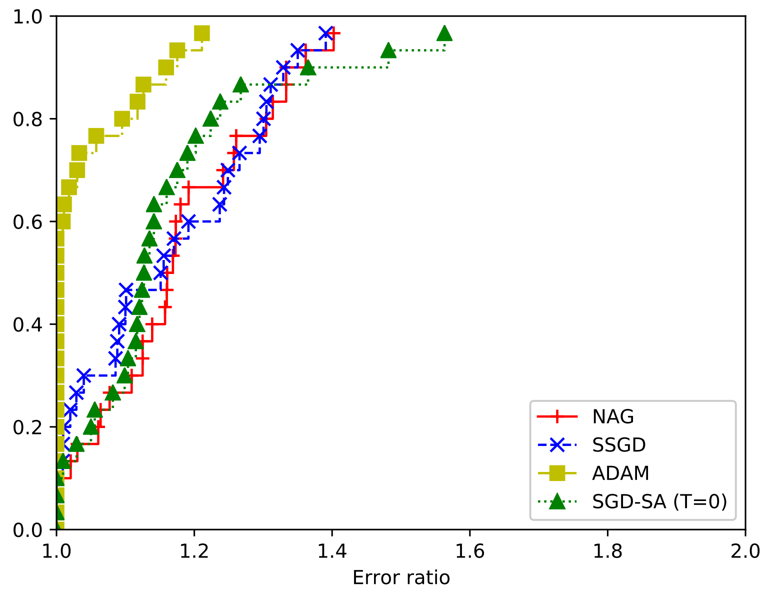
6.5.2. Comparison of state-of-art techniques versus SGD-SA (T=0)

SGD-SA with null temperature is an extreme version of SGD-SA which rejects all worsening moves. The null temperature breaks the SA mechanism, thus SGD-SA with null temperature cannot be properly classified as a metaheuristic algorithm. Nevertheless, the promising results of this algorithm, presented in Section 6.4.1, suggest that comparing its performance with state-of-art techniques and other SA-based ones may lead to meaningful observations.

The comparison between SGD-SA with null temperature and state-of-art techniques is shown in Figure 6.29 and Figure 6.30. In terms of loss, SGD-SA with null temperature outperforms all the state-of-art techniques, while, in terms of error, it performs similarly to NAG and SSGD. SGD-SA with null temperature shows a small overfitting with respect to state-of art techniques, however this difference is less pronounced if compared to other SA-based techniques. These results highlight the effectiveness of move rejection, even if its application is not managed by the SA acceptance test, but by a simpler policy which rejects all worsening moves.

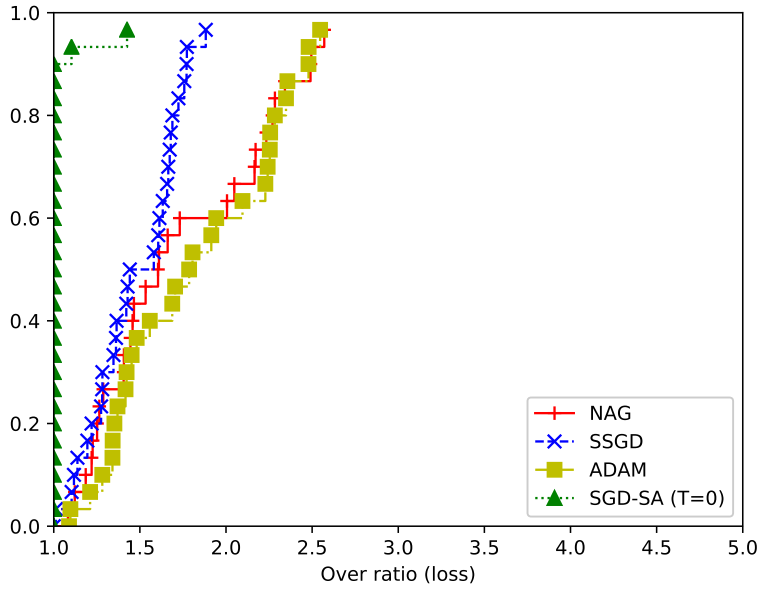


(a) Loss comparison.

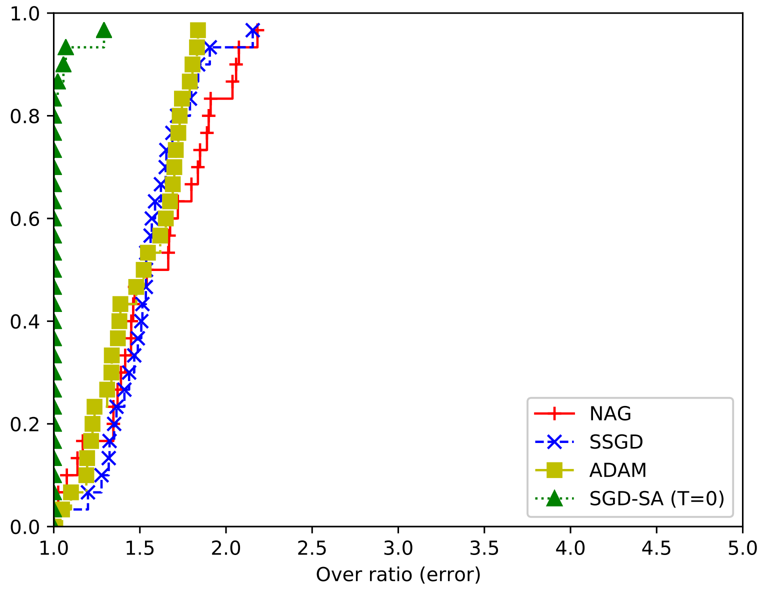


(b) Error comparison.

Figure 6.29.: Performance profile comparison over the test set: NAG vs. SSGD vs. ADAM vs. SGD-SA (T=0).



(a) Overfitting on loss comparison.

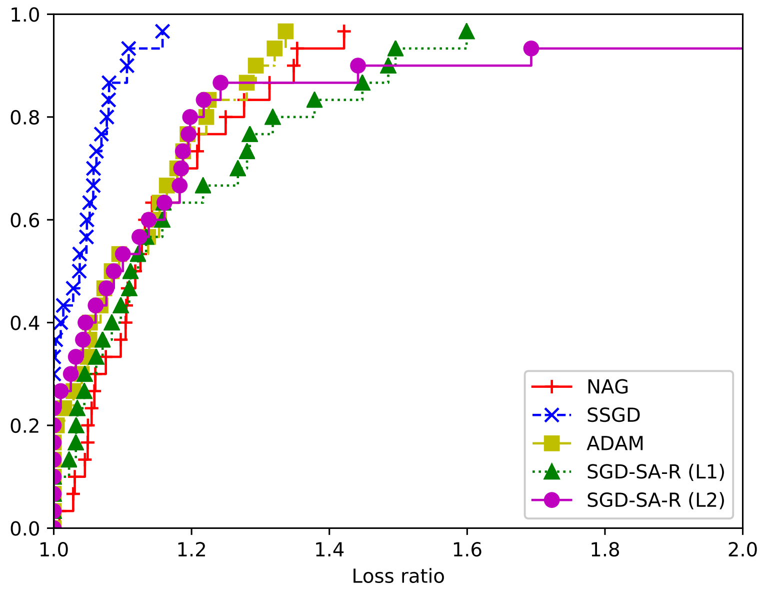


(b) Overfitting on error comparison.

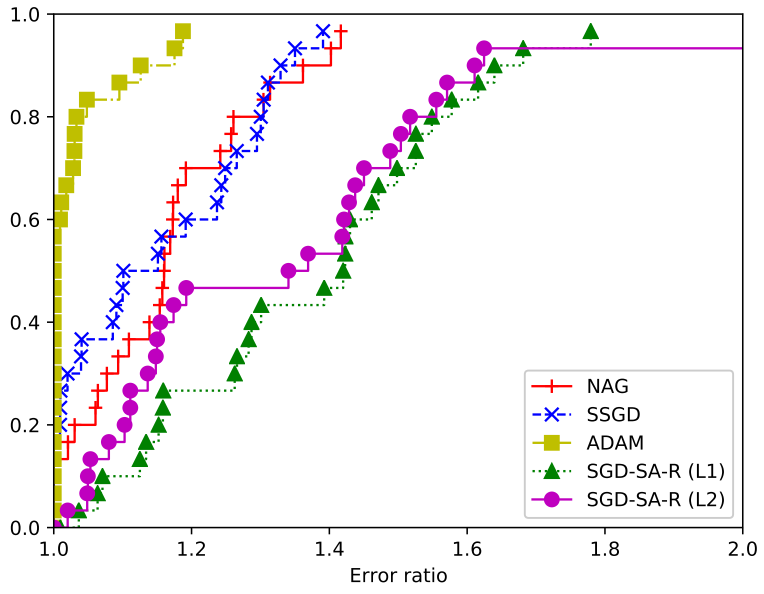
Figure 6.30.: Performance profile comparison for overfitting: NAG vs. SSGD vs. ADAM vs. SGD-SA (T=0).

6.5.3. Comparison of state-of-art techniques versus SGD-SA-R

The comparison between SGD-SA-R and state-of-art techniques is shown in Figure 6.31 and Figure 6.32. The results are similar to the ones obtained with SGD-SA technique, thus leading to the same observations. In terms of loss, SGD-SA-R shows competitive results, performing similarly to ADAM and NAG. In terms of error, it performs significantly worse than state-of-art techniques. SGD-SA-R shows a much smaller overfitting with respect to state-of art techniques. This proves that using different functions for generating and evaluating solutions improves the generalizability of the results. The results of SGD-SA-R with L1 and L2 norms are identical under the overfitting point of view; however, the L2 norm leads to slightly better performance in terms of loss and error.

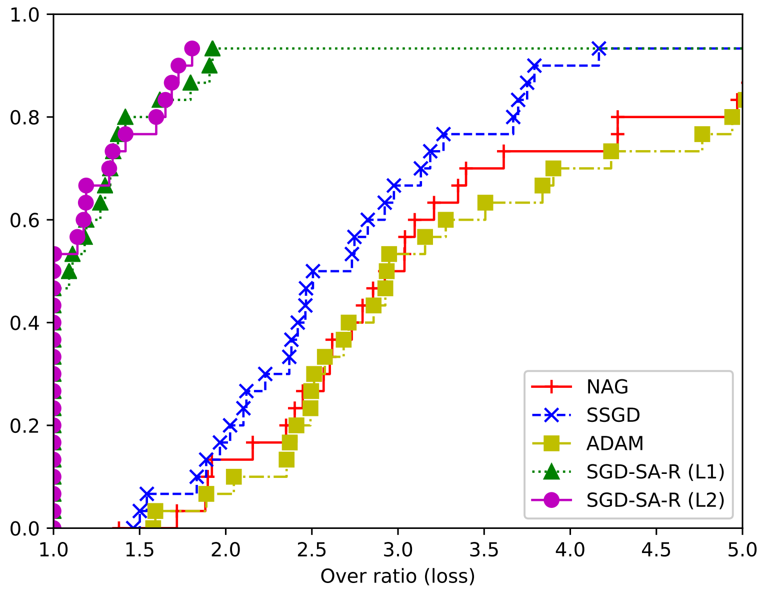


(a) Loss comparison.

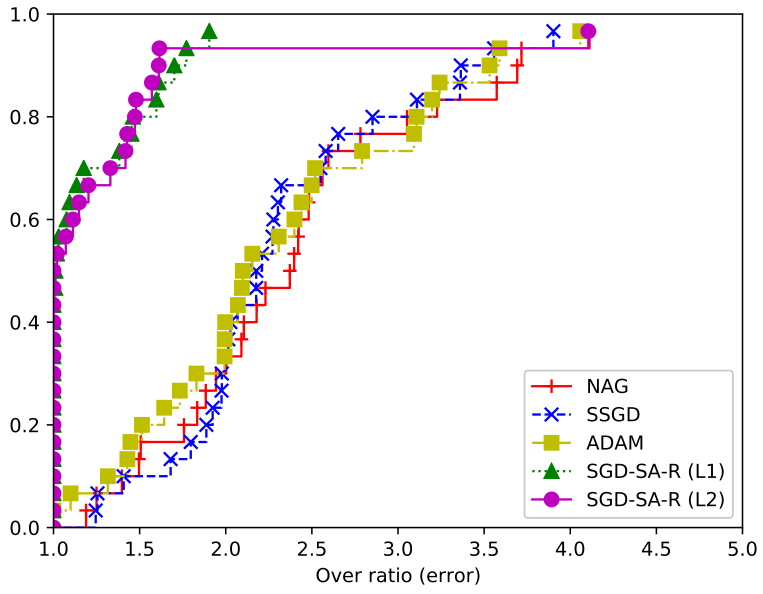


(b) Error comparison.

Figure 6.31.: Performance profile comparison over the test set: NAG vs. SSGD vs. ADAM vs. SGD-SA-R.



(a) Overfitting on loss comparison.

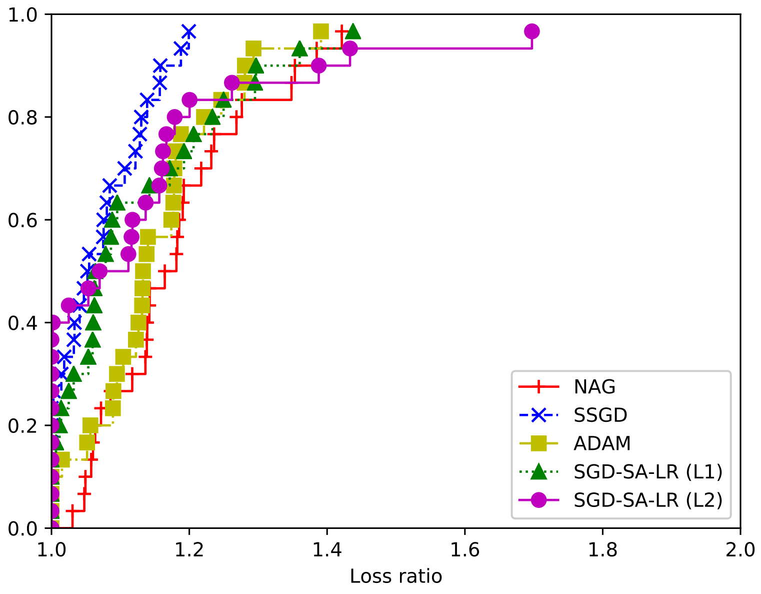


(b) Overfitting on error comparison.

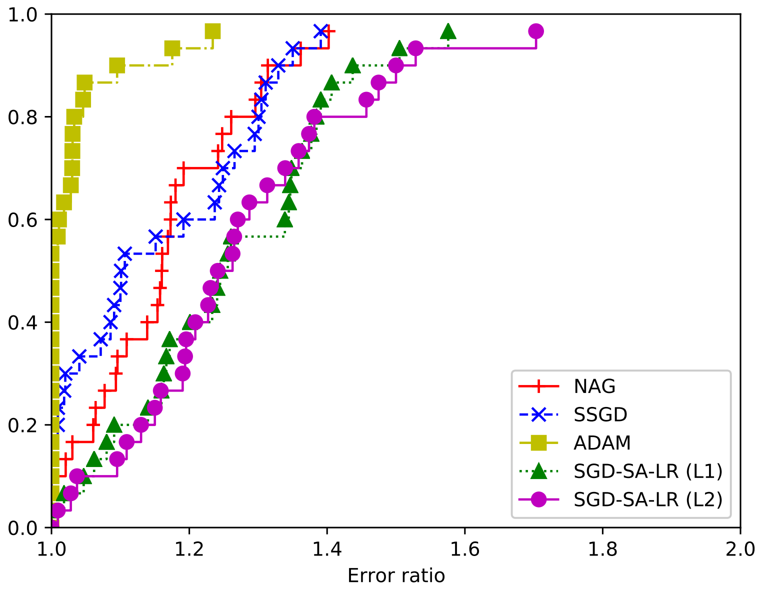
Figure 6.32.: Performance profile comparison for overfitting: NAG vs. SSGD vs. ADAM vs. SGD-SA-R.

6.5.4. Comparison of state-of-art techniques versus SGD-SA-LR

The comparison between SGD-SA-LR and state-of-art techniques is shown in Figure 6.33 and Figure 6.34. In terms of loss, SGD-SA-LR shows very good results, performing better than ADAM and NAG, and slightly worse than SSGD. In terms of error, it performs worse than every state-of-art technique. SGD-SA-LR shows a smaller overfitting with respect to state-of art techniques, however this difference is less pronounced in comparison with any other SA-based technique. The results of SGD-SA-LR with L1 and L2 norms are identical in terms of loss and error; however, the L2 norm leads to a slightly lower overfitting.

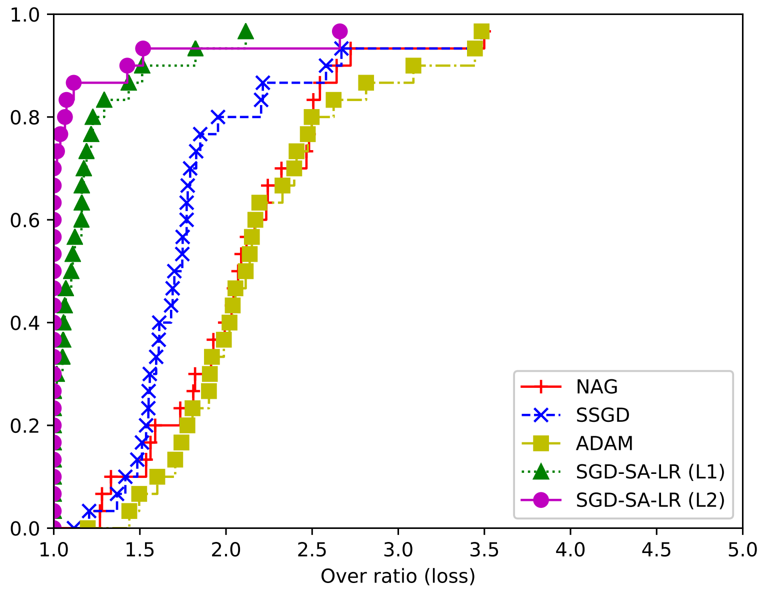


(a) Loss comparison.

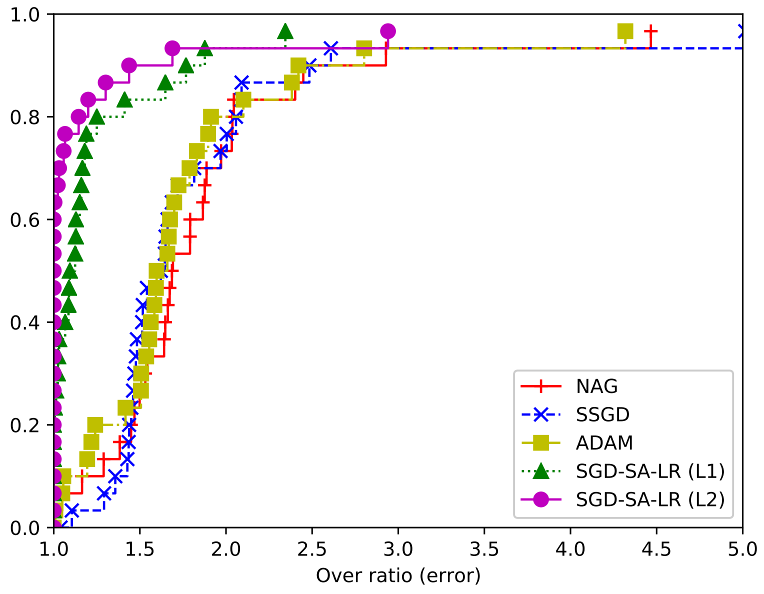


(b) Error comparison.

Figure 6.33.: Performance profile comparison over the test set: NAG vs. SSGD vs. ADAM vs. SGD-SA-LR.



(a) Overfitting on loss comparison.



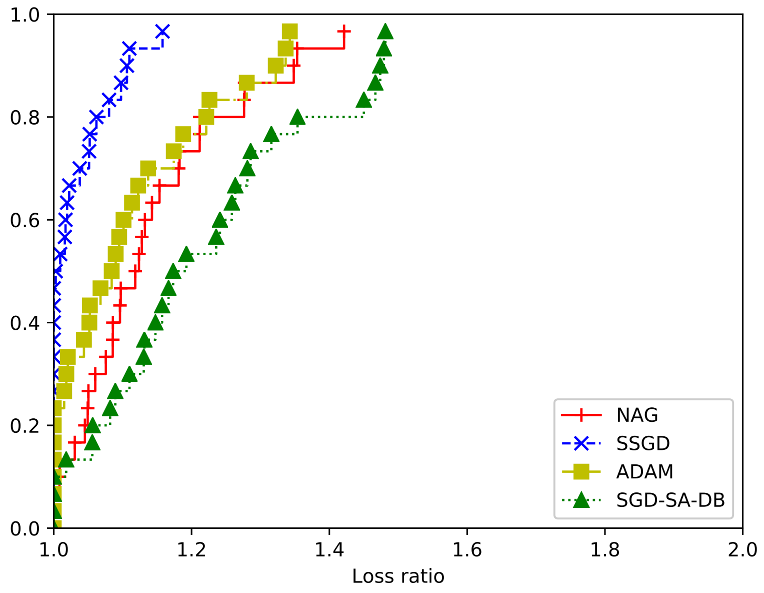
(b) Overfitting on error comparison.

Figure 6.34.: Performance profile comparison for overfitting: NAG vs. SSGD vs. ADAM vs. SGD-SA-LR.

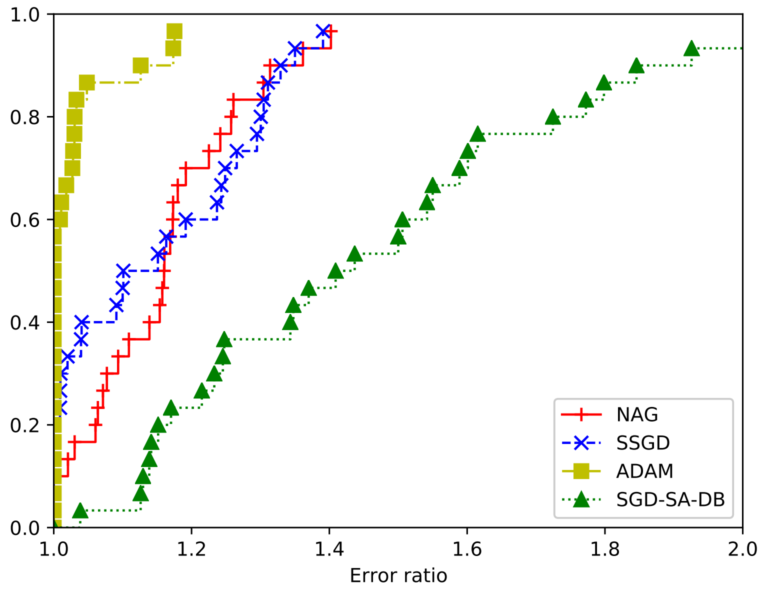
6.5.5. Comparison of state-of-art techniques versus SGD-SA-DB

The comparison between SGD-SA-DB and state-of-art techniques is shown in Figure 6.35 and Figure 6.36. In terms of loss, SGD-SA-DB performs worse than state-of-art techniques, but the results are still somehow comparable. In terms of error, the results are however poor. On the other hand, SGD-SA-DB shows an extremely small overfitting, if compared with state-of art techniques.

It is interesting to observe that these results – high error and low overfitting – are typical of intermediate solutions. Recalling the observation of Section 6.4.4, the small number of moves performed due to the ineffectiveness of SA limits the ability of the algorithm to find better solutions. The few moves accepted may cause the algorithm to perform as if the training was interrupted prematurely. As a consequence, the idea of using two batches per move may still be effective, even though the standard implementation of SA is not suitable for it.

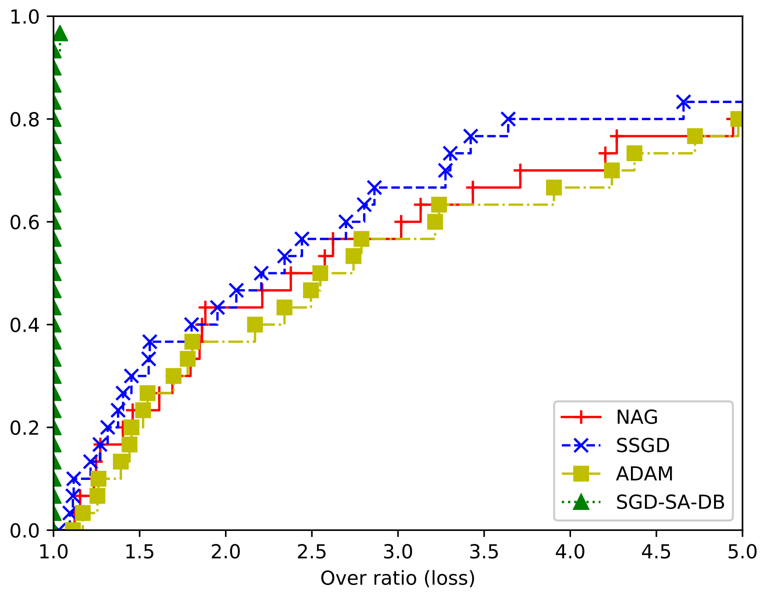


(a) Loss comparison.

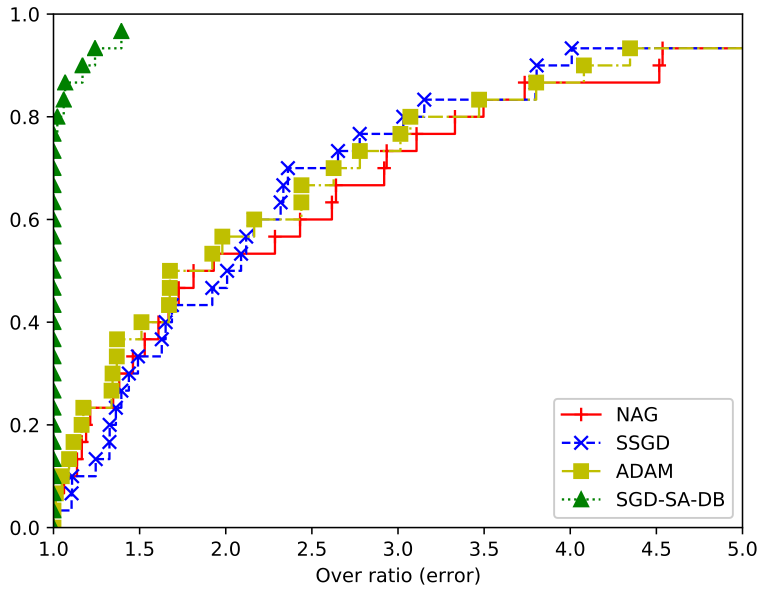


(b) Error comparison.

Figure 6.35.: Performance profile comparison over the test set: NAG vs. SSGD vs. ADAM vs. SGD-SA-DB.



(a) Overfitting on loss comparison.

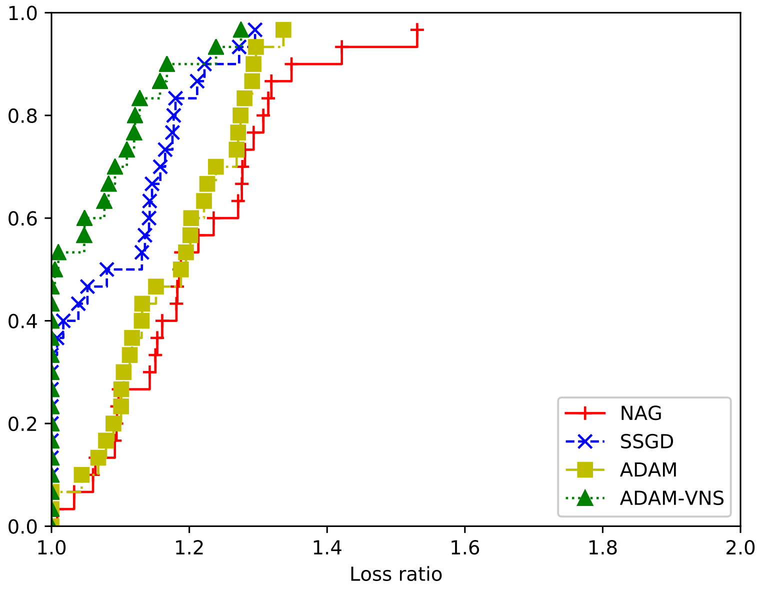


(b) Overfitting on error comparison.

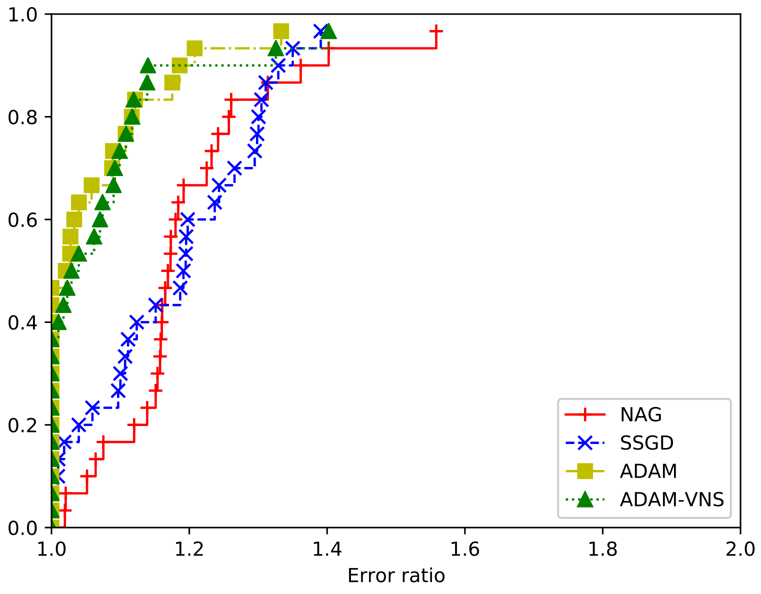
Figure 6.36.: Performance profile comparison for overfitting: NAG vs. SSGD vs. ADAM vs. SGD-SA-DB.

6.5.6. Comparison of state-of-art techniques versus ADAM-VNS

The comparison between ADAM-VNS and state-of-art techniques is shown in Figure 6.37 and Figure 6.38. In terms of loss, ADAM-VNS outperforms all the state-of-art techniques, while, in terms of error, it performs similarly to ADAM, and still outperforms NAG and SSGD. ADAM shows a small overfitting with respect to state-of art techniques, however this difference is less pronounced if compared to SA-based techniques. This observation suggests that the random Gaussian noise in combination with early stopping (if applied during the training) improves the generalizability of the solutions, without harming the intensification effectiveness of the algorithm.

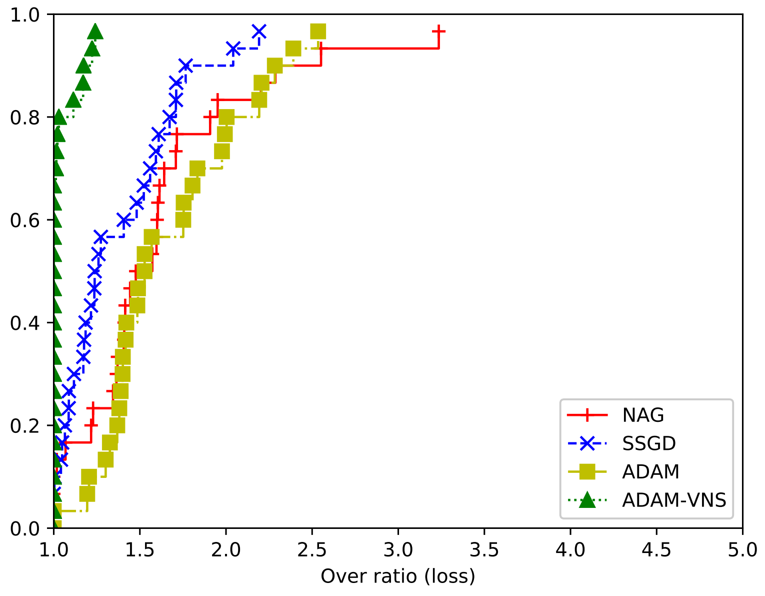


(a) Loss comparison.

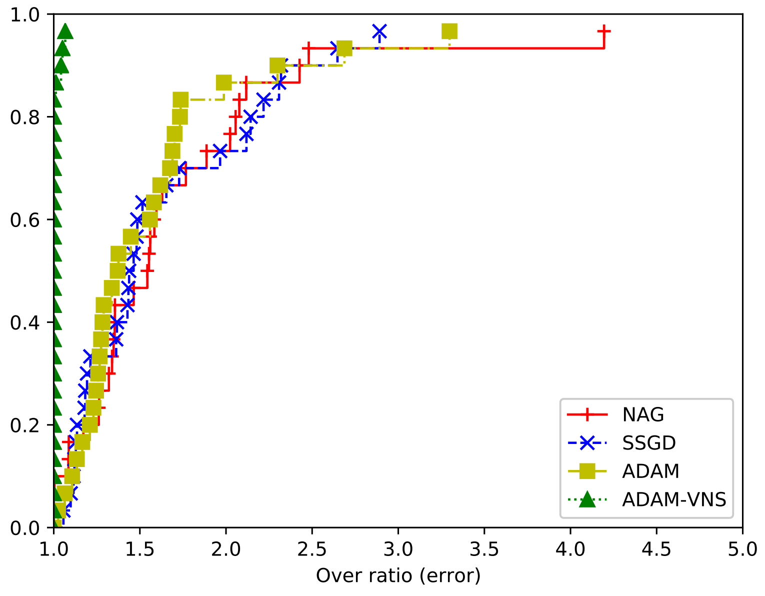


(b) Error comparison.

Figure 6.37.: Performance profile comparison over the test set: NAG vs. SSGD vs. ADAM vs. ADAM-VNS.



(a) Overfitting on loss comparison.



(b) Overfitting on error comparison.

Figure 6.38.: Performance profile comparison for overfitting: NAG vs. SSGD vs. ADAM vs. ADAM-VNS.

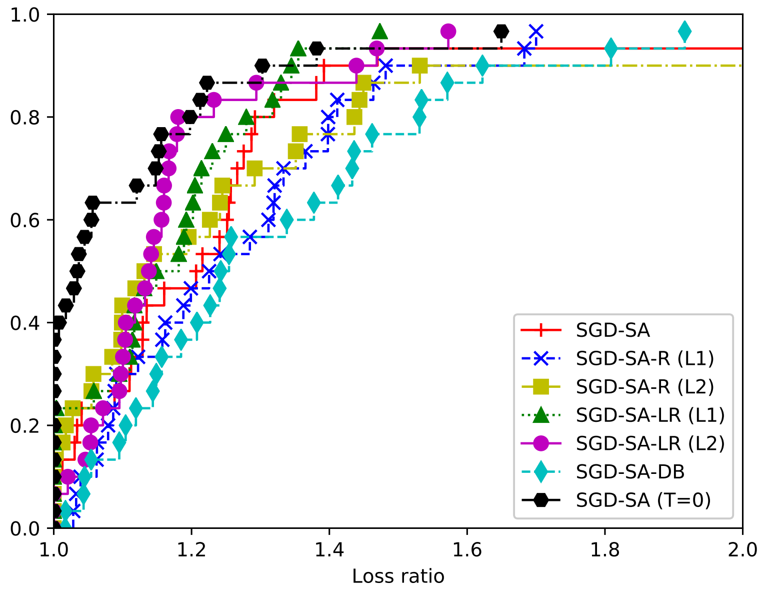
6.5.7. Comparison of SA-based techniques

The comparison between SA-based techniques is shown in Figure 6.39 and Figure 6.40. The results show that, in terms of loss and error, SGD-SA-LR is the best one, followed by SGD-SA-R with L2 norm, SGD-SA, SGD-SA-R with L1 norm, and SGD-SA-DB. As to overfitting, the best method is SGD-SA-DB, followed by SGD-SA-R, SGD-SA, and SGD-SA-LR.

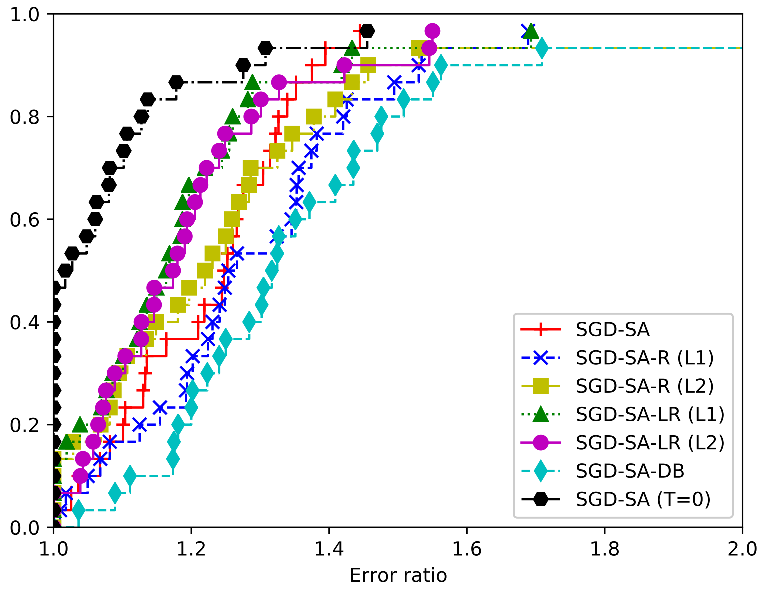
The results show a trade-off between effectiveness and overfitting: techniques with low loss and error tend to have a large overfitting, whereas techniques with low overfitting tend to have a poor performance. The complete independence between the solution generation and the cost function, as in SGD-SA-R and SGD-SA-DB, is a very effective diversification strategy but weakens the intensification of the algorithm. Techniques where the solution generation and the cost function have common elements, as SGD-SA and SGD-SA-LR, tend to perform better, at the cost of a slightly higher overfitting.

All the SA-based techniques presented were designed as improvements of SGD-SA. In particular, SGD-SA-R introduces a different cost function, independent from the solution generation and from the loss. As the results show, this modification reduces overfitting, maintaining a similar performance. This technique is further modified by introducing a partial dependency from the loss in the cost function, obtaining SGD-SA-LR. The increased influence of the loss improved the intensification, resulting in a better performance. Overall the results indicate SGD-SA-LR as the best SA-based technique.

SGD-SA with null temperature is not technically a SA-based algorithm, hence it was not mentioned in the considerations above. However, it is interesting to observe that it outperforms all SA-based techniques. Whereas, following the aforementioned effectiveness-overfitting trade-off, it shows a larger overfitting with respect to SA-based techniques. These results highlight that the effectiveness of SA-based algorithms is due to the move rejection feature. Moreover, they suggest that the sophistication of this feature does not lead to any improvement in the performance, but only in the overfitting.

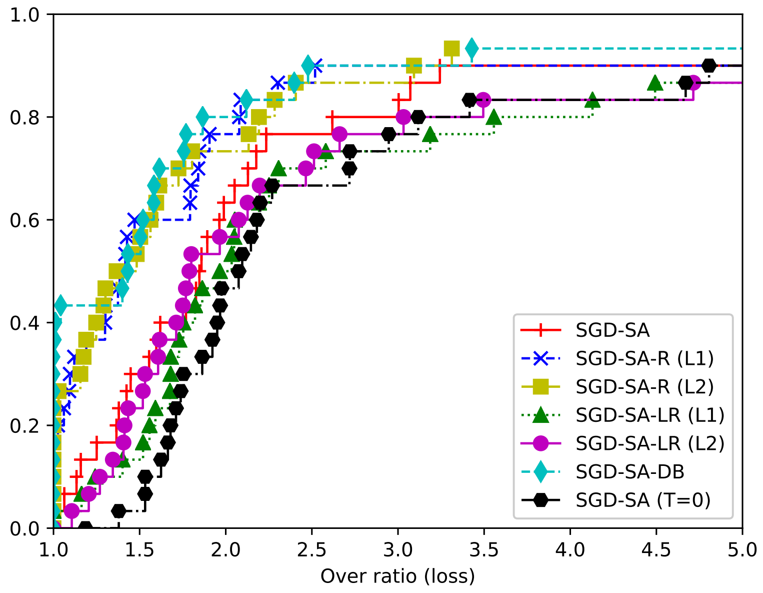


(a) Loss comparison.

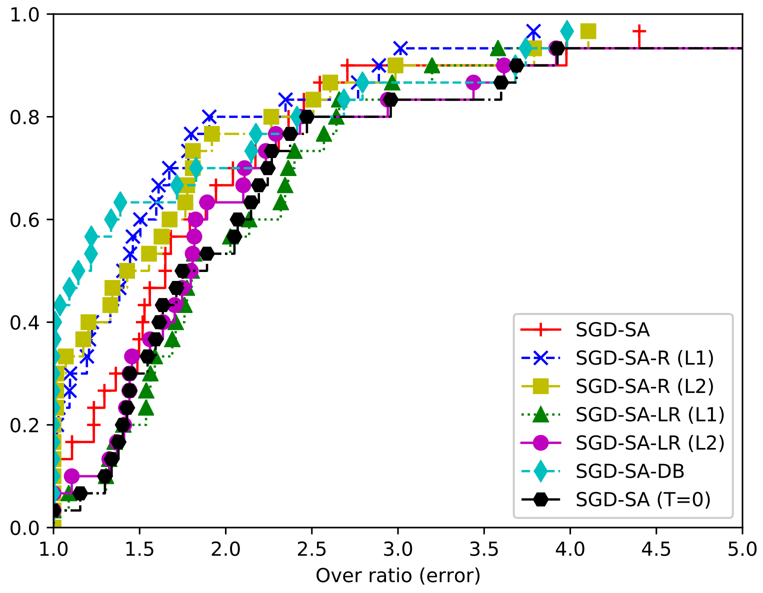


(b) Error comparison.

Figure 6.39.: Performance profile comparison over the test set: SGD-SA vs. SGD-SA-R vs. SGD-SA-LR vs. SGD-SA-DB vs. SGD-SA (T=0).



(a) Overfitting on loss comparison.



(b) Overfitting on error comparison.

Figure 6.40.: Performance profile comparison for overfitting: SGD-SA vs. SGD-SA-R vs. SGD-SA-LR vs. SGD-SA-DB vs. SGD-SA (T=0).

7. Conclusions

In the present work, we have proposed new techniques for the training of Convolutional Neural Networks (CNN) based on the embedding of Stochastic Gradient Descent (SGD) within metaheuristic frameworks. The performances of our algorithms have been tested by training a deep neural network to solve the image classification problem for different benchmark datasets. Finally, the results have been used to compare the effectiveness of our techniques with the state-of-art of SGD-based training algorithms.

The application of metaheuristic frameworks to the training of neural network has been shown to profitably improve the generalization property of the solutions. On the other hand, the additional structure introduced to SGD reduces the ability of the algorithms to find better solutions: only few operations influence the training, whereas the others increase the overhead.

The Simulated Annealing (SA) framework introduces the possibility of rejecting moves that do not satisfy a specific criterion, allowing the algorithm to avoid poor solutions. The test has proven the importance of this feature, attesting it as the most significant element introduced by SA. The standard behavior of SA implies that the number of accepted moves gradually decreases as the system cools down. However, in the context of the neural network training, a reduced dynamic has a dramatic impact on the results. Moreover, the properties of the generated solutions may change during the training, whereas the parameters of SA are set at the beginning of the execution. As a consequence, the SA criterion may become inadequate. The test results have shown that the SA-based techniques can achieve performances comparable, in terms of loss, to the state-of-art ones. In terms of error, instead, the performances are noticeably worse than the state-of-art ones.

The Variable Neighborhood Search (VNS) framework introduces the concept of early stopping in combination with the addition of random Gaussian noise. The test has proven the usefulness of this feature in improving the generalizability of the solutions, without a significant drop in the performances. The VNS framework normally requires local optima to be found iteratively, but in the context of neural network training proving a solution to be a local optimum is not feasible. This peculiarity of the problem requires the formulation of an alternative stopping condition, different from local optimality. Unfortunately, such condition is rather arbitrary. Hence the performance depends heavily on the implementation quality of the condition and on parameter tuning. The test results have shown that the VNS-based technique, with our implementation, achieved better results than all state-of-art techniques considered, both in terms of loss and error.

Future works may focus on the extensive study of VNS-based techniques, with the aim of better understanding the reasons behind their promising results. In our experience, it would be significant trying to define a cleaner stopping condition and to design new VNS-based algorithms. A different research path may consist in the development of techniques that exploit the specific metaheuristic features that have been proven effective for the neural network training, yet without implementing the full frameworks.

A. Detailed tests results

The above sections analyzed the test results aggregated in tables or summarized by means of plots. The purpose of this appendix is to provide more detailed information on test results. The data related to each training epoch is too large to be fully reported (slightly less than 40000 table rows). Whereas, the unaggregated test results related to the final performance of the trained networks are more manageable (less than 400 table rows) and are useful to better understand the variability in the performance of the optimizers. The data reported below is raw and with 5 decimal places (no rounding has been performed).

				Loss		Error	
Optimizer	Note	Dataset	Seed	Train	Test	Train	Test
NAG		MNIST	0	0.01466	0.03451	0.00538	0.01020
NAG		MNIST	1	0.01469	0.03916	0.00513	0.01180
NAG		MNIST	2	0.01684	0.03202	0.00600	0.01000
NAG		MNIST	3	0.01441	0.03627	0.00542	0.01130
NAG		MNIST	4	0.01180	0.03588	0.00393	0.00990
NAG		MNIST	5	0.01099	0.04260	0.00368	0.01200
NAG		MNIST	6	0.01328	0.03998	0.00475	0.01170
NAG		MNIST	7	0.01892	0.04058	0.00635	0.01220
NAG		MNIST	8	0.01330	0.03601	0.00447	0.01050
NAG		MNIST	9	0.02603	0.04458	0.00840	0.01360
NAG		F-MNIST	0	0.01210	0.03522	0.00438	0.00990
NAG		F-MNIST	1	0.01703	0.04181	0.00628	0.01220
NAG		F-MNIST	2	0.01534	0.03624	0.00535	0.01050
NAG		F-MNIST	3	0.01474	0.03998	0.00502	0.01200
NAG		F-MNIST	4	0.01493	0.03498	0.00505	0.01030
NAG		F-MNIST	5	0.01212	0.03911	0.00407	0.01100
NAG		F-MNIST	6	0.01360	0.04483	0.00477	0.01280
NAG		F-MNIST	7	0.02048	0.03931	0.00687	0.01270
NAG		F-MNIST	8	0.01226	0.03929	0.00433	0.01130
NAG		F-MNIST	9	0.01527	0.03896	0.00510	0.01230
NAG		CIFAR-10	0	0.19592	0.73775	0.06976	0.20200
NAG		CIFAR-10	1	0.13104	0.74786	0.04662	0.18110
NAG		CIFAR-10	2	0.16121	0.77685	0.05778	0.20160
NAG		CIFAR-10	3	0.19951	0.72836	0.07070	0.20000
NAG		CIFAR-10	4	0.15372	0.75124	0.05496	0.19140
NAG		CIFAR-10	5	0.20041	0.72344	0.07214	0.19640
NAG		CIFAR-10	6	0.14807	0.72719	0.05296	0.18930
NAG		CIFAR-10	7	0.19237	0.78975	0.06834	0.21270
NAG		CIFAR-10	8	0.14550	0.79635	0.05230	0.20130
NAG		CIFAR-10	9	0.23778	0.65899	0.08494	0.19710

Table A.1.: Test results for NAG.

				Loss		Error	
Optimizer	Note	Dataset	Seed	Train	Test	Train	Test
SSGD		MNIST	0	0.01158	0.03510	0.00402	0.01090
SSGD		MNIST	1	0.01251	0.03315	0.00425	0.01100
SSGD		MNIST	2	0.01324	0.03020	0.00445	0.01020
SSGD		MNIST	3	0.01249	0.03227	0.00432	0.00990
SSGD		MNIST	4	0.01199	0.03305	0.00405	0.01030
SSGD		MNIST	5	0.01153	0.03148	0.00395	0.00920
SSGD		MNIST	6	0.01263	0.03466	0.00403	0.01070
SSGD		MNIST	7	0.01291	0.03598	0.00452	0.01100
SSGD		MNIST	8	0.01172	0.03432	0.00403	0.01080
SSGD		MNIST	9	0.01235	0.03491	0.00427	0.01110
SSGD		F-MNIST	0	0.01239	0.03493	0.00417	0.00970
SSGD		F-MNIST	1	0.01237	0.03539	0.00433	0.01210
SSGD		F-MNIST	2	0.01207	0.03540	0.00393	0.01140
SSGD		F-MNIST	3	0.01146	0.03373	0.00413	0.01040
SSGD		F-MNIST	4	0.01234	0.03188	0.00428	0.01040
SSGD		F-MNIST	5	0.01239	0.03423	0.00407	0.01110
SSGD		F-MNIST	6	0.01261	0.03652	0.00427	0.01120
SSGD		F-MNIST	7	0.01228	0.03600	0.00397	0.01050
SSGD		F-MNIST	8	0.01267	0.02914	0.00433	0.00910
SSGD		F-MNIST	9	0.01290	0.03772	0.00432	0.01090
SSGD		CIFAR-10	0	0.27822	0.74318	0.09916	0.21870
SSGD		CIFAR-10	1	0.28013	0.71874	0.10014	0.22040
SSGD		CIFAR-10	2	0.28186	0.69467	0.09920	0.21370
SSGD		CIFAR-10	3	0.28401	0.74246	0.10090	0.22420
SSGD		CIFAR-10	4	0.28578	0.73109	0.10132	0.22190
SSGD		CIFAR-10	5	0.29249	0.71826	0.10428	0.22250
SSGD		CIFAR-10	6	0.28703	0.71795	0.10318	0.21380
SSGD		CIFAR-10	7	0.28080	0.72048	0.09850	0.22010
SSGD		CIFAR-10	8	0.30304	0.70331	0.10752	0.21510
SSGD		CIFAR-10	9	0.28841	0.72900	0.10146	0.22330

Table A.2.: Test results for SSGD.

				Loss		Error	
Optimizer	Note	Dataset	Seed	Train	Test	Train	Test
ADAM		MNIST	0	0.01126	0.03843	0.00392	0.00990
ADAM		MNIST	1	0.01366	0.03613	0.00453	0.01000
ADAM		MNIST	2	0.01125	0.03866	0.00360	0.01030
ADAM		MNIST	3	0.01102	0.03067	0.00355	0.00860
ADAM		MNIST	4	0.01495	0.03710	0.00518	0.01020
ADAM		MNIST	5	0.01111	0.03447	0.00397	0.00930
ADAM		MNIST	6	0.01623	0.03817	0.00515	0.01100
ADAM		MNIST	7	0.01423	0.04091	0.00517	0.01110
ADAM		MNIST	8	0.01223	0.03484	0.00407	0.00990
ADAM		MNIST	9	0.01409	0.03644	0.00482	0.01140
ADAM		F-MNIST	0	0.01146	0.03730	0.00402	0.01140
ADAM		F-MNIST	1	0.01228	0.03276	0.00438	0.00870
ADAM		F-MNIST	2	0.01520	0.04330	0.00535	0.01150
ADAM		F-MNIST	3	0.01691	0.04362	0.00563	0.01090
ADAM		F-MNIST	4	0.01400	0.04261	0.00462	0.01160
ADAM		F-MNIST	5	0.01278	0.04066	0.00435	0.01120
ADAM		F-MNIST	6	0.01219	0.03154	0.00433	0.00940
ADAM		F-MNIST	7	0.01173	0.03977	0.00412	0.01010
ADAM		F-MNIST	8	0.01254	0.03559	0.00440	0.00940
ADAM		F-MNIST	9	0.01255	0.04213	0.00398	0.01080
ADAM		CIFAR-10	0	0.12738	0.73142	0.04510	0.17280
ADAM		CIFAR-10	1	0.13048	0.71204	0.04596	0.17020
ADAM		CIFAR-10	2	0.14150	0.75308	0.05192	0.17190
ADAM		CIFAR-10	3	0.12314	0.74323	0.04266	0.17240
ADAM		CIFAR-10	4	0.13297	0.72900	0.04694	0.17770
ADAM		CIFAR-10	5	0.10435	0.69214	0.03652	0.16480
ADAM		CIFAR-10	6	0.10033	0.67608	0.03642	0.16310
ADAM		CIFAR-10	7	0.13260	0.73364	0.04718	0.16870
ADAM		CIFAR-10	8	0.11231	0.74020	0.04054	0.17390
ADAM		CIFAR-10	9	0.10485	0.69303	0.03714	0.16800

Table A.3.: Test results for ADAM.

				Loss		Error	
Optimizer	Note	Dataset	Seed	Train	Test	Train	Test
SGD-SA		MNIST	0	0.02170	0.03625	0.00747	0.01260
SGD-SA		MNIST	1	0.02415	0.04607	0.00842	0.01460
SGD-SA		MNIST	2	0.02580	0.03749	0.00842	0.01380
SGD-SA		MNIST	3	0.02229	0.03438	0.00710	0.01040
SGD-SA		MNIST	4	0.02454	0.03989	0.00852	0.01210
SGD-SA		MNIST	5	0.02531	0.04115	0.00868	0.01280
SGD-SA		MNIST	6	0.01663	0.03466	0.00557	0.01040
SGD-SA		MNIST	7	0.02688	0.04205	0.00947	0.01500
SGD-SA		MNIST	8	0.01936	0.03349	0.00653	0.01190
SGD-SA		MNIST	9	0.01928	0.03328	0.00610	0.01090
SGD-SA		F-MNIST	0	0.03363	0.04194	0.01128	0.01290
SGD-SA		F-MNIST	1	0.01814	0.03134	0.00583	0.01110
SGD-SA		F-MNIST	2	0.02302	0.03938	0.00775	0.01300
SGD-SA		F-MNIST	3	0.03103	0.04027	0.01013	0.01190
SGD-SA		F-MNIST	4	0.02279	0.03800	0.00768	0.01190
SGD-SA		F-MNIST	5	0.03294	0.03534	0.01078	0.01200
SGD-SA		F-MNIST	6	0.01705	0.03226	0.00578	0.01090
SGD-SA		F-MNIST	7	0.01961	0.03320	0.00667	0.01250
SGD-SA		F-MNIST	8	0.02041	0.03491	0.00693	0.01210
SGD-SA		F-MNIST	9	0.03187	0.04656	0.01108	0.01420
SGD-SA		CIFAR-10	0	0.96473	1.31343	0.17118	0.23790
SGD-SA		CIFAR-10	1	0.59729	0.80533	0.20828	0.26470
SGD-SA		CIFAR-10	2	0.48602	0.74949	0.14928	0.22050
SGD-SA		CIFAR-10	3	0.35557	0.66711	0.12470	0.21960
SGD-SA		CIFAR-10	4	0.37418	0.63771	0.13210	0.21350
SGD-SA		CIFAR-10	5	0.46108	0.70533	0.16294	0.22930
SGD-SA		CIFAR-10	6	0.45804	0.62216	0.15926	0.21390
SGD-SA		CIFAR-10	7	0.57668	0.75699	0.20360	0.25210
SGD-SA		CIFAR-10	8	0.65614	0.80818	0.23184	0.28330
SGD-SA		CIFAR-10	9	0.43382	0.70134	0.14532	0.22700

Table A.4.: Test results for SGD-SA.

				Loss		Error	
Optimizer	Note	Dataset	Seed	Train	Test	Train	Test
Rand-SGD		CIFAR-10	0	0.57808	0.78522	0.18800	0.24730
Rand-SGD		CIFAR-10	1	0.39584	0.66153	0.13728	0.21130
Rand-SGD		CIFAR-10	2	0.49987	0.78300	0.17724	0.24440
Rand-SGD		CIFAR-10	3	0.48945	0.74700	0.17328	0.24780
Rand-SGD		CIFAR-10	4	0.53145	0.81209	0.19038	0.26340
Rand-SGD		CIFAR-10	5	0.54389	0.78565	0.17994	0.24650
Rand-SGD		CIFAR-10	6	0.49052	0.77912	0.17008	0.24310
Rand-SGD		CIFAR-10	7	0.33725	0.60262	0.11834	0.20030
Rand-SGD		CIFAR-10	8	0.73848	0.95668	0.23794	0.28410
Rand-SGD		CIFAR-10	9	0.46162	0.67940	0.16440	0.23220

Table A.5.: Test results for Randomized-SGD.

				Loss		Error	
Optimizer	Note	Dataset	Seed	Train	Test	Train	Test
SGD-SA	T=0	MNIST	0	0.02150	0.03984	0.00742	0.01190
SGD-SA	T=0	MNIST	1	0.02053	0.04330	0.00730	0.01190
SGD-SA	T=0	MNIST	2	0.01754	0.03918	0.00622	0.01050
SGD-SA	T=0	MNIST	3	0.01947	0.03739	0.00667	0.01090
SGD-SA	T=0	MNIST	4	0.01715	0.03367	0.00590	0.00880
SGD-SA	T=0	MNIST	5	0.01857	0.03241	0.00612	0.01050
SGD-SA	T=0	MNIST	6	0.01643	0.03185	0.00567	0.01040
SGD-SA	T=0	MNIST	7	0.04036	0.05749	0.01290	0.01630
SGD-SA	T=0	MNIST	8	0.01683	0.03447	0.00600	0.01130
SGD-SA	T=0	MNIST	9	0.01710	0.03363	0.00575	0.01020
SGD-SA	T=0	F-MNIST	0	0.01607	0.03493	0.00557	0.01140
SGD-SA	T=0	F-MNIST	1	0.02392	0.04083	0.00802	0.01360
SGD-SA	T=0	F-MNIST	2	0.01641	0.03087	0.00558	0.01060
SGD-SA	T=0	F-MNIST	3	0.01562	0.03129	0.00520	0.00900
SGD-SA	T=0	F-MNIST	4	0.01928	0.03530	0.00670	0.01060
SGD-SA	T=0	F-MNIST	5	0.02137	0.04057	0.00740	0.01190
SGD-SA	T=0	F-MNIST	6	0.01860	0.03302	0.00663	0.01050
SGD-SA	T=0	F-MNIST	7	0.02484	0.03822	0.00817	0.01250
SGD-SA	T=0	F-MNIST	8	0.01518	0.03145	0.00512	0.01000
SGD-SA	T=0	F-MNIST	9	0.02104	0.03649	0.00713	0.01210
SGD-SA	T=0	CIFAR-10	0	0.30019	0.56471	0.10348	0.18240
SGD-SA	T=0	CIFAR-10	1	0.33083	0.57848	0.11708	0.18980
SGD-SA	T=0	CIFAR-10	2	0.35092	0.59764	0.12452	0.19510
SGD-SA	T=0	CIFAR-10	3	0.34727	0.59079	0.12194	0.19380
SGD-SA	T=0	CIFAR-10	4	0.37504	0.64251	0.13470	0.20610
SGD-SA	T=0	CIFAR-10	5	0.30043	0.56118	0.10746	0.18190
SGD-SA	T=0	CIFAR-10	6	0.29307	0.54824	0.10190	0.18380
SGD-SA	T=0	CIFAR-10	7	0.46465	0.71972	0.16416	0.23030
SGD-SA	T=0	CIFAR-10	8	0.33199	0.58520	0.11724	0.19610
SGD-SA	T=0	CIFAR-10	9	0.35889	0.62126	0.12632	0.20560

Table A.6.: Test results for SGD-SA (T=0).

				Loss		Error	
Optimizer	Note	Dataset	Seed	Train	Test	Train	Test
SGD-SA-R	L1	MNIST	0	0.02109	0.03365	0.00708	0.01060
SGD-SA-R	L1	MNIST	1	0.02523	0.04026	0.00840	0.01430
SGD-SA-R	L1	MNIST	2	0.03418	0.04518	0.01100	0.01420
SGD-SA-R	L1	MNIST	3	0.03721	0.04556	0.01267	0.01410
SGD-SA-R	L1	MNIST	4	0.02174	0.03577	0.00733	0.01250
SGD-SA-R	L1	MNIST	5	0.01965	0.03218	0.00637	0.01060
SGD-SA-R	L1	MNIST	6	0.02101	0.03309	0.00682	0.01240
SGD-SA-R	L1	MNIST	7	0.02326	0.03760	0.00763	0.01140
SGD-SA-R	L1	MNIST	8	0.02781	0.03979	0.00863	0.01270
SGD-SA-R	L1	MNIST	9	0.02753	0.03511	0.00877	0.01080
SGD-SA-R	L1	F-MNIST	0	0.01871	0.03605	0.00632	0.01100
SGD-SA-R	L1	F-MNIST	1	0.02486	0.03641	0.00818	0.01280
SGD-SA-R	L1	F-MNIST	2	0.01884	0.03279	0.00632	0.01060
SGD-SA-R	L1	F-MNIST	3	0.03045	0.04104	0.01030	0.01520
SGD-SA-R	L1	F-MNIST	4	0.03571	0.04040	0.01152	0.01340
SGD-SA-R	L1	F-MNIST	5	0.02105	0.03634	0.00708	0.01170
SGD-SA-R	L1	F-MNIST	6	0.02135	0.03461	0.00727	0.01190
SGD-SA-R	L1	F-MNIST	7	0.02070	0.03391	0.00687	0.01170
SGD-SA-R	L1	F-MNIST	8	0.03534	0.04660	0.01150	0.01530
SGD-SA-R	L1	F-MNIST	9	0.03076	0.04084	0.01048	0.01390
SGD-SA-R	L1	CIFAR-10	0	0.67034	0.79707	0.23702	0.27260
SGD-SA-R	L1	CIFAR-10	1	0.52471	0.70891	0.18460	0.23700
SGD-SA-R	L1	CIFAR-10	2	0.70618	1.00581	0.18222	0.24460
SGD-SA-R	L1	CIFAR-10	3	0.56203	0.70837	0.19892	0.24540
SGD-SA-R	L1	CIFAR-10	4	0.81473	0.93349	0.27392	0.31620
SGD-SA-R	L1	CIFAR-10	5	0.73101	0.95406	0.20006	0.25140
SGD-SA-R	L1	CIFAR-10	6	0.58637	0.72399	0.19770	0.25260
SGD-SA-R	L1	CIFAR-10	7	0.65738	0.77333	0.23350	0.27260
SGD-SA-R	L1	CIFAR-10	8	0.65771	0.78034	0.22360	0.26530
SGD-SA-R	L1	CIFAR-10	9	0.71770	0.86844	0.19806	0.25170

Table A.7.: Test results for SGD-SA-R (L1).

				Loss		Error	
Optimizer	Note	Dataset	Seed	Train	Test	Train	Test
SGD-SA-R	L2	MNIST	0	0.02204	0.03259	0.00665	0.01010
SGD-SA-R	L2	MNIST	1	0.02297	0.03134	0.00747	0.01080
SGD-SA-R	L2	MNIST	2	0.02405	0.03505	0.00765	0.01150
SGD-SA-R	L2	MNIST	3	0.02160	0.03667	0.00693	0.01280
SGD-SA-R	L2	MNIST	4	0.01784	0.03187	0.00588	0.01100
SGD-SA-R	L2	MNIST	5	0.02241	0.03226	0.00707	0.01080
SGD-SA-R	L2	MNIST	6	0.02683	0.03562	0.00887	0.01180
SGD-SA-R	L2	MNIST	7	0.14992	0.14287	0.03793	0.03710
SGD-SA-R	L2	MNIST	8	0.02119	0.03542	0.00697	0.01100
SGD-SA-R	L2	MNIST	9	0.02087	0.03395	0.00690	0.00960
SGD-SA-R	L2	F-MNIST	0	0.02569	0.03641	0.00827	0.01120
SGD-SA-R	L2	F-MNIST	1	0.03075	0.03890	0.00927	0.01320
SGD-SA-R	L2	F-MNIST	2	0.03995	0.04727	0.01207	0.01500
SGD-SA-R	L2	F-MNIST	3	0.03023	0.03838	0.00952	0.01240
SGD-SA-R	L2	F-MNIST	4	0.02908	0.03466	0.00908	0.01080
SGD-SA-R	L2	F-MNIST	5	0.02836	0.03631	0.00932	0.01250
SGD-SA-R	L2	F-MNIST	6	0.01970	0.03186	0.00655	0.00990
SGD-SA-R	L2	F-MNIST	7	0.02266	0.03247	0.00760	0.01060
SGD-SA-R	L2	F-MNIST	8	0.01856	0.03453	0.00623	0.01220
SGD-SA-R	L2	F-MNIST	9	0.02380	0.03528	0.00755	0.01240
SGD-SA-R	L2	CIFAR-10	0	0.53265	0.70287	0.18814	0.24560
SGD-SA-R	L2	CIFAR-10	1	0.65353	0.83845	0.22470	0.27650
SGD-SA-R	L2	CIFAR-10	2	0.63748	0.86286	0.20054	0.25850
SGD-SA-R	L2	CIFAR-10	3	0.67668	0.84859	0.22790	0.27770
SGD-SA-R	L2	CIFAR-10	4	0.56662	0.76257	0.17518	0.24330
SGD-SA-R	L2	CIFAR-10	5	0.97093	1.17178	0.18070	0.25630
SGD-SA-R	L2	CIFAR-10	6	0.62738	0.74392	0.17030	0.23140
SGD-SA-R	L2	CIFAR-10	7	0.52814	0.71325	0.18468	0.24250
SGD-SA-R	L2	CIFAR-10	8	0.58394	0.79076	0.19268	0.25220
SGD-SA-R	L2	CIFAR-10	9	0.60293	0.80242	0.20232	0.26390

Table A.8.: Test results for SGD-SA-R (L2).

				Loss		Error	
Optimizer	Note	Dataset	Seed	Train	Test	Train	Test
SGD-SA-R	L1 + HT	CIFAR-10	0	0.64238	0.82727	0.22736	0.28860
SGD-SA-R	L1 + HT	CIFAR-10	1	0.47473	0.71061	0.16062	0.23650
SGD-SA-R	L1 + HT	CIFAR-10	2	0.48689	0.72886	0.15720	0.22440
SGD-SA-R	L1 + HT	CIFAR-10	3	0.49002	0.68572	0.16994	0.23020
SGD-SA-R	L1 + HT	CIFAR-10	4	0.54545	0.77957	0.18434	0.25560
SGD-SA-R	L1 + HT	CIFAR-10	5	0.93401	1.00803	0.33590	0.35450
SGD-SA-R	L1 + HT	CIFAR-10	6	0.48979	0.66877	0.17248	0.22620
SGD-SA-R	L1 + HT	CIFAR-10	7	0.50489	0.77009	0.16372	0.23230
SGD-SA-R	L1 + HT	CIFAR-10	8	0.57647	0.87810	0.15514	0.22920
SGD-SA-R	L1 + HT	CIFAR-10	9	0.48078	0.76450	0.14380	0.21500

Table A.9.: Test results for SGD-SA-R (L1 + HT).

				Loss		Error	
Optimizer	Note	Dataset	Seed	Train	Test	Train	Test
SGD-SA-R	L2 + HT	CIFAR-10	0	0.55778	0.81745	0.13598	0.21820
SGD-SA-R	L2 + HT	CIFAR-10	1	0.72207	0.99553	0.16398	0.23690
SGD-SA-R	L2 + HT	CIFAR-10	2	0.50061	0.71451	0.16414	0.23400
SGD-SA-R	L2 + HT	CIFAR-10	3	0.50824	0.78080	0.15916	0.23680
SGD-SA-R	L2 + HT	CIFAR-10	4	0.45050	0.75474	0.13764	0.22590
SGD-SA-R	L2 + HT	CIFAR-10	5	0.58785	0.73461	0.20562	0.25410
SGD-SA-R	L2 + HT	CIFAR-10	6	0.44872	0.72946	0.14986	0.21980
SGD-SA-R	L2 + HT	CIFAR-10	7	0.81878	1.04329	0.17196	0.24740
SGD-SA-R	L2 + HT	CIFAR-10	8	0.43324	0.67556	0.15144	0.22340
SGD-SA-R	L2 + HT	CIFAR-10	9	0.43303	0.65344	0.15046	0.22030

Table A.10.: Test results for SGD-SA-R (L2 + HT).

				Loss		Error	
Optimizer	Note	Dataset	Seed	Train	Test	Train	Test
SGD-SA-R	L1 + SC	CIFAR-10	0	0.49496	0.71253	0.17600	0.23710
SGD-SA-R	L1 + SC	CIFAR-10	1	0.36588	0.66436	0.12690	0.21260
SGD-SA-R	L1 + SC	CIFAR-10	2	0.50895	0.75211	0.15668	0.23100
SGD-SA-R	L1 + SC	CIFAR-10	3	0.53934	0.78606	0.15986	0.22570
SGD-SA-R	L1 + SC	CIFAR-10	4	0.34744	0.64540	0.12308	0.21080
SGD-SA-R	L1 + SC	CIFAR-10	5	0.39337	0.67014	0.13794	0.22140
SGD-SA-R	L1 + SC	CIFAR-10	6	0.49937	0.69686	0.17648	0.23050
SGD-SA-R	L1 + SC	CIFAR-10	7	0.48885	0.76743	0.15336	0.23320
SGD-SA-R	L1 + SC	CIFAR-10	8	0.71261	1.03721	0.15982	0.23490
SGD-SA-R	L1 + SC	CIFAR-10	9	0.38583	0.66007	0.13526	0.21970

Table A.11.: Test results for SGD-SA-R (L1 + SC).

				Loss		Error	
Optimizer	Note	Dataset	Seed	Train	Test	Train	Test
SGD-SA-R	L2 + SC	CIFAR-10	0	0.49899	0.73370	0.16228	0.23570
SGD-SA-R	L2 + SC	CIFAR-10	1	0.53139	0.69626	0.18802	0.24050
SGD-SA-R	L2 + SC	CIFAR-10	2	0.42572	0.67770	0.15048	0.22670
SGD-SA-R	L2 + SC	CIFAR-10	3	0.44130	0.68821	0.15102	0.22540
SGD-SA-R	L2 + SC	CIFAR-10	4	0.48910	0.68292	0.17168	0.22820
SGD-SA-R	L2 + SC	CIFAR-10	5	0.38360	0.63154	0.13530	0.21000
SGD-SA-R	L2 + SC	CIFAR-10	6	0.34579	0.69087	0.12314	0.22200
SGD-SA-R	L2 + SC	CIFAR-10	7	0.37397	0.66535	0.13186	0.21620
SGD-SA-R	L2 + SC	CIFAR-10	8	0.41603	0.67738	0.14470	0.21920
SGD-SA-R	L2 + SC	CIFAR-10	9	0.45847	0.73598	0.16290	0.24380

Table A.12.: Test results for SGD-SA-R (L2 + SC).

				Loss		Error	
Optimizer	Note	Dataset	Seed	Train	Test	Train	Test
SGD-SA-LR	L1	MNIST	0	0.01785	0.03263	0.00595	0.01080
SGD-SA-LR	L1	MNIST	1	0.02085	0.03492	0.00723	0.01080
SGD-SA-LR	L1	MNIST	2	0.01866	0.03538	0.00640	0.01140
SGD-SA-LR	L1	MNIST	3	0.01696	0.03337	0.00568	0.01080
SGD-SA-LR	L1	MNIST	4	0.03208	0.04286	0.01000	0.01490
SGD-SA-LR	L1	MNIST	5	0.01633	0.03185	0.00565	0.01070
SGD-SA-LR	L1	MNIST	6	0.02158	0.03557	0.00723	0.01120
SGD-SA-LR	L1	MNIST	7	0.01658	0.03485	0.00572	0.01120
SGD-SA-LR	L1	MNIST	8	0.04285	0.04934	0.01425	0.01560
SGD-SA-LR	L1	MNIST	9	0.02296	0.03838	0.00743	0.01090
SGD-SA-LR	L1	F-MNIST	0	0.01855	0.03312	0.00602	0.01030
SGD-SA-LR	L1	F-MNIST	1	0.02111	0.03480	0.00705	0.01210
SGD-SA-LR	L1	F-MNIST	2	0.01999	0.03647	0.00702	0.01230
SGD-SA-LR	L1	F-MNIST	3	0.02281	0.04161	0.00765	0.01290
SGD-SA-LR	L1	F-MNIST	4	0.01721	0.03828	0.00620	0.01170
SGD-SA-LR	L1	F-MNIST	5	0.03665	0.04656	0.01217	0.01500
SGD-SA-LR	L1	F-MNIST	6	0.01688	0.03198	0.00575	0.01090
SGD-SA-LR	L1	F-MNIST	7	0.01720	0.03191	0.00583	0.00980
SGD-SA-LR	L1	F-MNIST	8	0.02093	0.03328	0.00702	0.01260
SGD-SA-LR	L1	F-MNIST	9	0.02399	0.03941	0.00780	0.01260
SGD-SA-LR	L1	CIFAR-10	0	0.34333	0.67880	0.11908	0.21310
SGD-SA-LR	L1	CIFAR-10	1	0.44908	0.78377	0.15492	0.24460
SGD-SA-LR	L1	CIFAR-10	2	0.43722	0.68655	0.15446	0.23110
SGD-SA-LR	L1	CIFAR-10	3	0.36792	0.66829	0.13288	0.21730
SGD-SA-LR	L1	CIFAR-10	4	0.46924	0.77440	0.16474	0.24470
SGD-SA-LR	L1	CIFAR-10	5	0.36925	0.69033	0.13100	0.22190
SGD-SA-LR	L1	CIFAR-10	6	0.36010	0.66057	0.12610	0.21990
SGD-SA-LR	L1	CIFAR-10	7	0.36829	0.62262	0.12708	0.20260
SGD-SA-LR	L1	CIFAR-10	8	0.42191	0.69579	0.14830	0.23280
SGD-SA-LR	L1	CIFAR-10	9	0.58542	0.79490	0.15934	0.23630

Table A.13.: Test results for SGD-SA-LR (L1).

				Loss		Error	
Optimizer	Note	Dataset	Seed	Train	Test	Train	Test
SGD-SA-LR	L2	MNIST	0	0.02438	0.03709	0.00803	0.01300
SGD-SA-LR	L2	MNIST	1	0.02518	0.03700	0.00822	0.01150
SGD-SA-LR	L2	MNIST	2	0.01684	0.03231	0.00555	0.01130
SGD-SA-LR	L2	MNIST	3	0.02229	0.03870	0.00773	0.01290
SGD-SA-LR	L2	MNIST	4	0.02950	0.04587	0.00948	0.01360
SGD-SA-LR	L2	MNIST	5	0.02059	0.03518	0.00672	0.01250
SGD-SA-LR	L2	MNIST	6	0.02094	0.03354	0.00712	0.01100
SGD-SA-LR	L2	MNIST	7	0.03179	0.04043	0.01040	0.01350
SGD-SA-LR	L2	MNIST	8	0.03192	0.04919	0.01063	0.01460
SGD-SA-LR	L2	MNIST	9	0.02200	0.03219	0.00733	0.01100
SGD-SA-LR	L2	F-MNIST	0	0.02375	0.03830	0.00783	0.01340
SGD-SA-LR	L2	F-MNIST	1	0.01900	0.03358	0.00612	0.01040
SGD-SA-LR	L2	F-MNIST	2	0.01542	0.03381	0.00532	0.01150
SGD-SA-LR	L2	F-MNIST	3	0.02739	0.04049	0.00908	0.01280
SGD-SA-LR	L2	F-MNIST	4	0.01907	0.03063	0.00647	0.00940
SGD-SA-LR	L2	F-MNIST	5	0.02192	0.03606	0.00742	0.01220
SGD-SA-LR	L2	F-MNIST	6	0.02161	0.03506	0.00727	0.01210
SGD-SA-LR	L2	F-MNIST	7	0.02196	0.03723	0.00747	0.01170
SGD-SA-LR	L2	F-MNIST	8	0.03884	0.04946	0.01312	0.01550
SGD-SA-LR	L2	F-MNIST	9	0.02063	0.03719	0.00693	0.01120
SGD-SA-LR	L2	CIFAR-10	0	0.36022	0.61970	0.12510	0.20570
SGD-SA-LR	L2	CIFAR-10	1	0.33002	0.60507	0.11772	0.19720
SGD-SA-LR	L2	CIFAR-10	2	0.44909	0.68209	0.15960	0.23020
SGD-SA-LR	L2	CIFAR-10	3	0.39318	0.65197	0.12918	0.21400
SGD-SA-LR	L2	CIFAR-10	4	0.43049	0.73036	0.15128	0.22440
SGD-SA-LR	L2	CIFAR-10	5	0.36213	0.63549	0.12798	0.20850
SGD-SA-LR	L2	CIFAR-10	6	0.35499	0.63996	0.12570	0.20720
SGD-SA-LR	L2	CIFAR-10	7	0.49327	0.73394	0.17258	0.24580
SGD-SA-LR	L2	CIFAR-10	8	0.39363	0.65421	0.13736	0.21020
SGD-SA-LR	L2	CIFAR-10	9	0.59487	0.76560	0.21010	0.25680

Table A.14.: Test results for SGD-SA-LR (L2).

				Loss		Error	
Optimizer	Note	Dataset	Seed	Train	Test	Train	Test
SGD-SA-DB		MNIST	0	0.02983	0.04049	0.00972	0.01330
SGD-SA-DB		MNIST	1	0.03483	0.04807	0.01150	0.01550
SGD-SA-DB		MNIST	2	0.02876	0.04451	0.00943	0.01370
SGD-SA-DB		MNIST	3	0.02302	0.03860	0.00755	0.01290
SGD-SA-DB		MNIST	4	0.01597	0.03490	0.00507	0.01130
SGD-SA-DB		MNIST	5	0.02791	0.03643	0.00922	0.01240
SGD-SA-DB		MNIST	6	0.02428	0.03847	0.00797	0.01300
SGD-SA-DB		MNIST	7	0.02374	0.04129	0.00772	0.01370
SGD-SA-DB		MNIST	8	0.01938	0.03494	0.00643	0.01140
SGD-SA-DB		MNIST	9	0.02343	0.03950	0.00760	0.01250
SGD-SA-DB		F-MNIST	0	0.01746	0.03805	0.00565	0.01210
SGD-SA-DB		F-MNIST	1	0.02640	0.03461	0.00863	0.01250
SGD-SA-DB		F-MNIST	2	0.01930	0.03225	0.00660	0.00980
SGD-SA-DB		F-MNIST	3	0.02158	0.03299	0.00695	0.01080
SGD-SA-DB		F-MNIST	4	0.02491	0.04099	0.00808	0.01270
SGD-SA-DB		F-MNIST	5	0.04005	0.04385	0.01327	0.01550
SGD-SA-DB		F-MNIST	6	0.01827	0.03566	0.00635	0.01100
SGD-SA-DB		F-MNIST	7	0.02526	0.03244	0.00848	0.01150
SGD-SA-DB		F-MNIST	8	0.02346	0.03945	0.00805	0.01470
SGD-SA-DB		F-MNIST	9	0.02668	0.03588	0.00905	0.01220
SGD-SA-DB		CIFAR-10	0	1.02687	1.08191	0.37154	0.38670
SGD-SA-DB		CIFAR-10	1	0.65107	0.83066	0.22156	0.27250
SGD-SA-DB		CIFAR-10	2	0.60731	0.75149	0.21572	0.25890
SGD-SA-DB		CIFAR-10	3	0.73597	0.95839	0.21262	0.26580
SGD-SA-DB		CIFAR-10	4	0.81505	0.90085	0.28952	0.31960
SGD-SA-DB		CIFAR-10	5	0.73467	0.85909	0.24164	0.28420
SGD-SA-DB		CIFAR-10	6	0.93703	0.99167	0.27760	0.31410
SGD-SA-DB		CIFAR-10	7	0.78952	0.91032	0.26704	0.29900
SGD-SA-DB		CIFAR-10	8	0.75277	0.83871	0.22552	0.27630
SGD-SA-DB		CIFAR-10	9	0.84167	0.97618	0.27802	0.31010

Table A.15.: Test results for SGD-SA-DB.

				Loss		Error	
Optimizer	Note	Dataset	Seed	Train	Test	Train	Test
ADAM-VNS		MNIST	0	0.01392	0.03716	0.00490	0.00970
ADAM-VNS		MNIST	1	0.02457	0.03740	0.00825	0.01140
ADAM-VNS		MNIST	2	0.01517	0.03298	0.00503	0.00930
ADAM-VNS		MNIST	3	0.02319	0.03439	0.00772	0.01140
ADAM-VNS		MNIST	4	0.01577	0.03279	0.00565	0.00860
ADAM-VNS		MNIST	5	0.01806	0.02783	0.00572	0.00770
ADAM-VNS		MNIST	6	0.01309	0.03753	0.00460	0.01010
ADAM-VNS		MNIST	7	0.01819	0.03168	0.00615	0.00990
ADAM-VNS		MNIST	8	0.01884	0.02915	0.00617	0.00910
ADAM-VNS		MNIST	9	0.02057	0.04325	0.00732	0.01210
ADAM-VNS		F-MNIST	0	0.01347	0.03510	0.00450	0.01030
ADAM-VNS		F-MNIST	1	0.02732	0.04177	0.00885	0.01220
ADAM-VNS		F-MNIST	2	0.01875	0.03407	0.00650	0.01030
ADAM-VNS		F-MNIST	3	0.01784	0.03533	0.00602	0.01030
ADAM-VNS		F-MNIST	4	0.01967	0.03220	0.00658	0.00870
ADAM-VNS		F-MNIST	5	0.01789	0.03797	0.00617	0.01090
ADAM-VNS		F-MNIST	6	0.01540	0.03532	0.00522	0.01050
ADAM-VNS		F-MNIST	7	0.01436	0.03773	0.00460	0.01050
ADAM-VNS		F-MNIST	8	0.01691	0.03373	0.00548	0.01000
ADAM-VNS		F-MNIST	9	0.02920	0.04404	0.00933	0.01230
ADAM-VNS		CIFAR-10	0	0.21276	0.60816	0.07562	0.17580
ADAM-VNS		CIFAR-10	1	0.18565	0.62962	0.06750	0.17410
ADAM-VNS		CIFAR-10	2	0.34968	0.59094	0.12316	0.19240
ADAM-VNS		CIFAR-10	3	0.26366	0.57313	0.09414	0.17740
ADAM-VNS		CIFAR-10	4	0.19459	0.57460	0.06896	0.17090
ADAM-VNS		CIFAR-10	5	0.19635	0.62866	0.07036	0.17640
ADAM-VNS		CIFAR-10	6	0.21558	0.62653	0.07738	0.17810
ADAM-VNS		CIFAR-10	7	0.20596	0.61061	0.07406	0.18120
ADAM-VNS		CIFAR-10	8	0.31912	0.60372	0.11374	0.19270
ADAM-VNS		CIFAR-10	9	0.18794	0.60177	0.06636	0.16970

Table A.16.: Test results for ADAM-VNS.

References

[1] Sebastian Ruder, *An overview of gradient descent optimization algorithms.*
ArXiv e-prints, Version 2 (2017).
ArXiv: `https://arxiv.org/abs/1609.04747`

[2] Diederik P. Kingma, Jimmy Ba, *ADAM: a method for stochastic optimization.*
ArXiv e-prints, Version 9 (2017).
ArXiv: `https://arxiv.org/abs/1412.6980`

[3] Kenneth Sörensen, Fred W. Glover, *Metaheuristics.*
In: Saul I. Gass, Michael C. Fu, Encyclopedia of Operations Research and Management Science, Springer (2013).
DOI: `https://doi.org/10.1007/978-1-4419-1153-7_1167`

[4] Scott Kirkpatrick, C. Daniel Gelatt, Mario P. Vecchi, *Optimization by Simulated Annealing.*
Science, Volume 220, Issue 4598, Pages 671-680, AAAS (1983).
DOI: `https://doi.org/10.1126/science.220.4598.671`

[5] Moon-Won Park, Yeong-Dae Kim, *A systematic procedure for setting parameters in simulated annealing algorithms.*
Computers & Operations Research, Volume 25, Issue 3, Pages 207-217, Elsevier (1998).
DOI: `https://doi.org/10.1016/S0305-0548(97)00054-3`

[6] Nenad Mladenović, Pierre Hansen, *Variable neighborhood search.*
Computers & Operations Research, Volume 24, Issue 11, Pages 1097-1100, Elsevier (1997).
DOI: `https://doi.org/10.1016/S0305-0548(97)00031-2`

[7] Enrique Alba, Rafael Martí, *Metaheuristic procedures for training neural networks.*
Operations Research/Computer Science Interfaces Series, Volume 35, Springer (2006).
DOI: `https://doi.org/10.1007/0-387-33416-5`

[8] Matteo Fischetti, Matteo Stringher, *Embedded hyper-parameter tuning by Simulated Annealing.*
ArXiv e-prints, Version 1 (2019).
ArXiv: `https://arxiv.org/abs/1906.01504`

[9] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner, *Gradient-based learning applied to document recognition.*
Proceedings of the IEEE, Volume 86, Issue 11, Pages 2278-2324, IEEE (1998).
DOI: `https://doi.org/10.1109/5.726791`

[10] Han Xiao, Kashif Rasul, Roland Vollgraf, *Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms.*
ArXiv e-prints, Version 2 (2017).
ArXiv: `https://arxiv.org/abs/1708.07747`

[11] Alex Krizhevsky, *Learning Multiple Layers of Features from Tiny Images.*
MSc thesis, unpublished (2009).
URL: `https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf`

[12] Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun, *Deep Residual Learning for Image Recognition.*
ArXiv e-prints, Version 1 (2015).
ArXiv: `https://arxiv.org/abs/1512.03385`

[13] Elizabeth D. Dolan, Jorge J. Moré, *Benchmarking optimization software with performance profiles.*
Mathematical Programming, Volume 91, Pages 201–213, Springer (2002).
DOI: `https://doi.org/10.1007/s101070100263`

Websites

[14] PyTorch website.
URL: `https://pytorch.org`
Visited: 20/10/2020

[15] PyTorch definition of cross-entropy loss.
URL:
`https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html`
Visited: 10/11/2020

[16] Matplotlib website.
URL: `https://matplotlib.org`
Visited: 14/11/2020

[17] Visual Studio Code website.
URL: `https://code.visualstudio.com`
Visited: 13/11/2020

[18] Git website.
URL: `https://git-scm.com`
Visited: 13/11/2020

[19] GitHub website.
URL: `https://github.com`
Visited: 13/11/2020

[20] NVIDIA website.
URL: `https://www.nvidia.com`
Visited: 14/11/2020

[21] NVIDIA CUDA website.
URL: `https://developer.nvidia.com/cuda-zone`
Visited: 14/11/2020

[22] Performance profiling Python script.
URL: `http://www.dei.unipd.it/~fisch/ricop/RO2/PerfProf/perfprof.py`
Visited: 10/11/2020.