YaleNUSCollege

Landscape Modification in Machine Learning Optimization

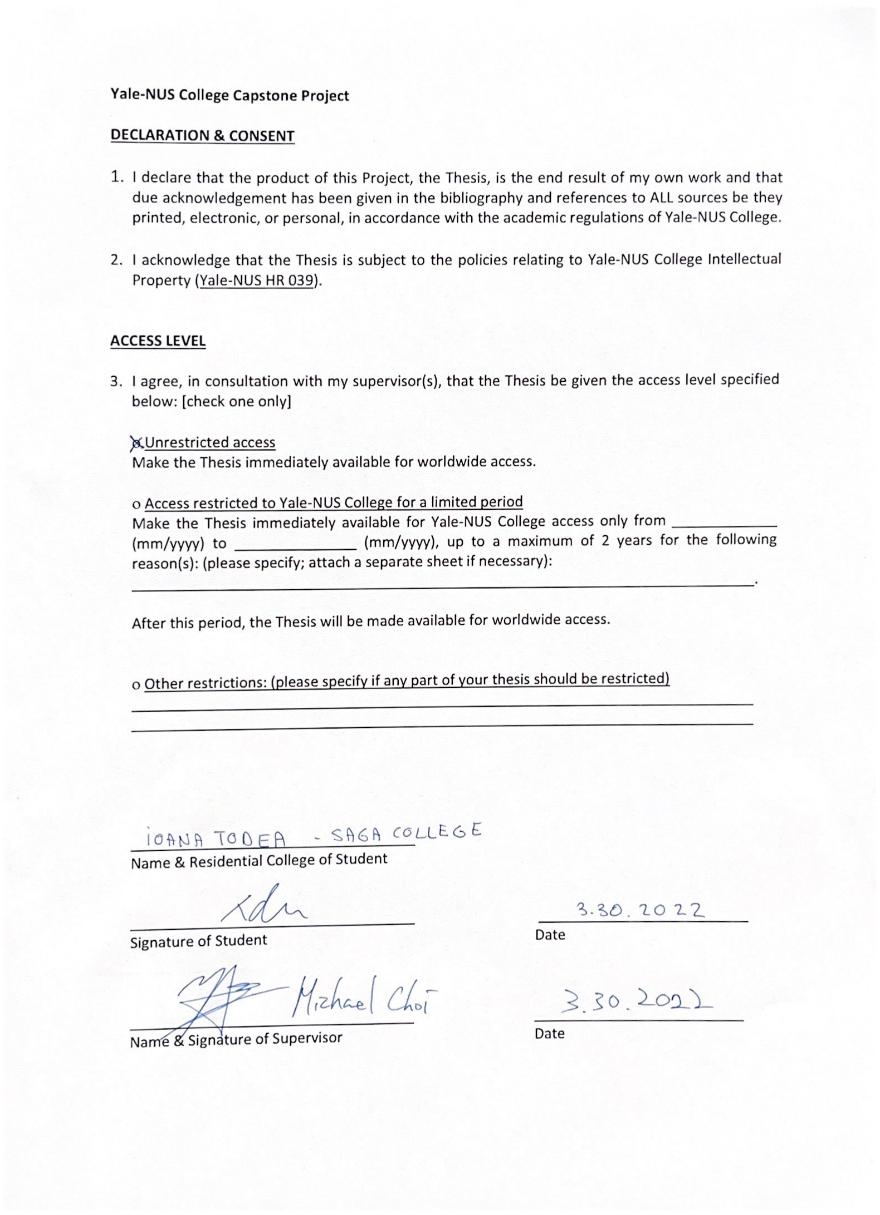
Ioana TODEA

Supervised by: Prof. Michael CHOI

Capstone Final Report for BSc (Honours) in

Mathematical, Computational and Statistical Sciences

AY 2021/2022



Yale-NUS College Capstone Project

DECLARATION & CONSENT

1. I declare that the product of this Project, the Thesis, is the end result of my own work and that due acknowledgement has been given in the bibliography and references to ALL sources be they printed, electronic, or personal, in accordance with the academic regulations of Yale-NUS College.

2. I acknowledge that the Thesis is subject to the policies relating to Yale-NUS College Intellectual Property (Yale-NUS HR 039).

ACCESS LEVEL

3. I agree, in consultation with my supervisor(s), that the Thesis be given the access level specified below: [check one only]

[✓] Unrestricted access
Make the Thesis immediately available for worldwide access.

o Access restricted to Yale-NUS College for a limited period
Make the Thesis immediately available for Yale-NUS College access only from ____________ (mm/yyyy) to ______________ (mm/yyyy), up to a maximum of 2 years for the following reason(s): (please specify; attach a separate sheet if necessary):
____.

After this period, the Thesis will be made available for worldwide access.

o Other restrictions: (please specify if any part of your thesis should be restricted)
____
____

IOANA TODEA - SAGA COLLEGE
Name & Residential College of Student

[illegible] | 3.30.2022
Signature of Student | Date

[illegible] Michael Choi | 3.30.2022
Name & Signature of Supervisor | Date

Acknowledgements

Foremost, I would like to thank my supervisor, Professor Michael Choi, for his kind guidance throughout the year, and for inspiring me with his never-dying, contagious enthusiasm for mathematics.

To my friends, Giulia, Olesia, and YeonHee, thank you for being my family away from home, always there to force a laugh at my questionable jokes, and to spare a shoulder for my tears. To my dearest friend back in Romania, Diana, thank you for showing me an example of dignity and power in front of any adversity, for being a role model and a loyal friend. To Russell, thank you for reigniting in me a love of life that had slightly faded, and for responding to all my flaws with patience and love.

To my grandparents, thank you for gifting me the happiest of childhoods, and for awakening in me a life-affirming curiosity that I treasure to this day. To my parents, thank you for your selfless devotion, for pouring your lives and hearts into my education, happiness, and well-being. To my mom, thank you for teaching me to never give up. To my dad, thank you for treating me like the most important person alive (and for sending me poems).

Finally, to my brother, thank you for your endless effort to teach me, help me, understand me, and love me at all times. Most importantly, thank you for always lighting my path with your love of knowledge, wisdom, and goodness.

YALE-NUS COLLEGE

Abstract

B.Sc (Hons)

Landscape Modification in Machine Learning Optimization

by Ioana TODEA

In this thesis, we explore the effects of a new optimization technique introduced by Choi [4], that proposes modifying the loss landscape of a problem in order to obtain faster convergence. This technique, which we call landscape modification, was introduced in the context of simulated annealing. However, this thesis will focus on its application to machine learning optimization problems. Firstly, we will give a brief introduction of optimization algorithms that are most commonly used in machine learning applications. Subsequently, we will explore the underlying ideas behind landscape modification, and explain why it could potentially improve the performance of the aforementioned algorithms. Finally, we will present the results of numerical experiments performed to investigate the improvements in accuracy caused by using landscape modification.

Keywords: *Stochastic Optimization, Machine Learning, Deep Learning, Landscape Modification, Gradient Descent.*

Word count: 7053

Contents

Chapter 1

Review of Optimization Algorithms

1.1 Introduction

The essential role that machine learning plays in our daily lives is, during our current times, universally acknowledged. Its applications across different industries are varied, a few examples being Netflix's recommendation system, face recognition-based login, and, soon enough, Tesla's self-driving cars. What fueled the rapid evolution of machine learning in the past decades was, firstly, increasingly powerful computers, and, secondly, access to immense amounts of data [1]. Amongst machine learning methods, one that has proved most successful in recent years is *deep learning*, which makes use of deep neural networks. However, neural networks are decision systems with a large number of parameters, and thus the process of their optimization is both crucial and challenging [1, 5]. Many efforts have been made to develop algorithms that would perform this task successfully, and we review several of them in Chapter 1. Subsequently, we propose a potential improvement to these methods by the use of landscape modification in Chapter 2. In Chapter 3, we examine numerical results of experiments testing the performance of the improved optimization algorithms.

1.1.1 Goal of Optimization in Machine Learning

Briefly put, the goal of optimization is to minimize the training error of a model, that is, to minimize the number of wrong predictions it generates [1]. Training error is measured by a chosen loss function, and, as we are focusing on a *supervised learning* approach, this function is evaluated on a set of labeled training data. Eventually, the final goal is to reduce the expected error (generalization error) for unseen data [1]. To formalize the problem, we consider a set of labeled data $\{(x_i, y_i)\}_i^n \in \mathbf{R}^{d_x} \times \mathbf{R}^{d_y}$, a vector of parameters $w \in \mathbf{R}^d$ and a prediction function $h : \mathbf{R}^{d_x} \times \mathbf{R}^d \to \mathbf{R}^{d_y}$ [1]. Given this information, we can define a loss function $U(w) : \mathbf{R}^d \to \mathbf{R}$ as:

$$U(w) = \frac{1}{n} \sum_{i=1}^{n} U_i(w), \tag{1.1}$$

where $U_i(w)$ is actually $U_i(h(x_i, w), y_i)$, that is, the loss evaluated for one sample. The goal is, therefore, to find:

$$\min_{w \in \mathbf{R^d}} \frac{1}{n} \sum_{i=1}^{n} U_i(w).$$

1.2 Optimization Algorithms

1.2.1 Gradient Descent

Motivated by astronomic calculations, the French mathematician Augustin-Louis Cauchy invented the optimization method of gradient descent [15], which, to this day, is the backbone of all gradient-based algorithms, and, thusly, an appropriate point to start our review. The main idea of gradient descent is that, given an initial point on the loss function, the first order derivatives of all parameters (the gradients) are used to find the direction of the **steepest descent** [29]. The algorithm takes a step in this direction based on a **step size**, and this processes is repeated until the minimum point on the loss function is reached, or until a satisfactory result is achieved [23]. The update of such an algorithm at iteration t may look like:

$$w_t \leftarrow w_{t-1} - \eta \nabla U(w_{t-1}), \tag{1.2}$$

where η is the **step size**, and $\nabla U(w) = \left[\frac{\partial U(w)}{\partial w_1}, \frac{\partial U(w)}{\partial w_2}, \dots, \frac{\partial U(w)}{\partial w_d}\right]^\top$ the gradient of $U(w)$. For convex functions, gradient descent is intuitive and robust, and is proven to converge to the global minimum [23]. However, when the landscape is more complex and multi-modal, there are several problems the gradient descent method can encounter. Firstly, it is very likely that it will converge to a local minimum or get stuck in a saddle point [1] [23, 29], as can be noticed in Fig. 1.1. Secondly, an appropriate step size must be chosen manually in order to ensure convergence. A step size that is too large will "overshoot" and miss the minimum, while a

[1]A saddle point is a location on the loss function that is not a minimum (global or local) but where the gradient is zero.

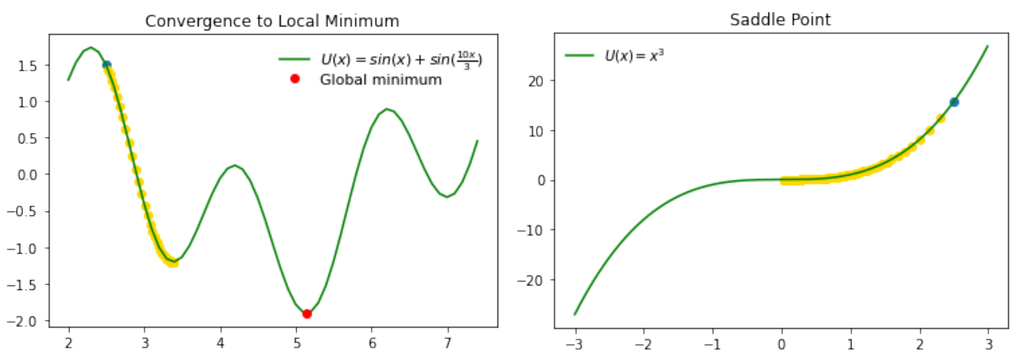


FIGURE 1.1: Left: Gradient descent converging to the local minimum. Right: Gradient descent being stuck in a saddle point.

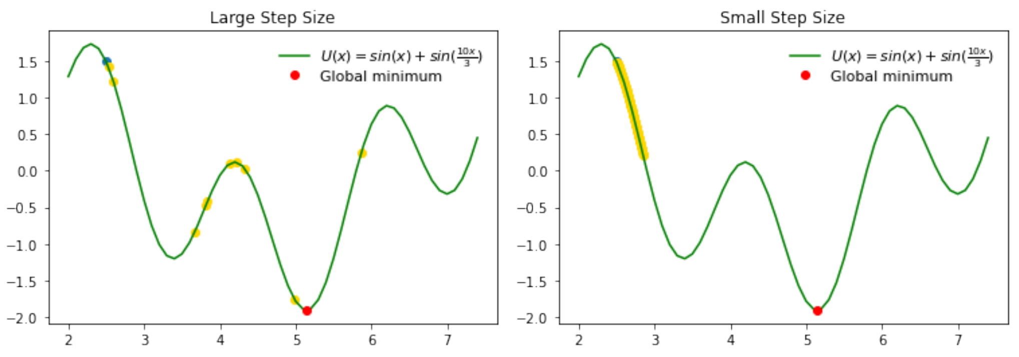


FIGURE 1.2: Left: Large step size causing gradient descent to diverge. Right: Small step size failing to converge.

very small step size might take too long to converge, as we can see in Fig. 1.2. Thirdly, gradient descent is computationally expensive in practice [23, 29]. As we can see from Eq. 1.1 and Eq. 1.2, at each iteration, all samples in the data are used to compute the error used for the gradient. Fortunately, optimization literature has developed potential solutions to overcome these two shortcomings of gradient descent.

1.2.2 Stochastic Gradient Descent

As mentioned above, in the best case scenario gradient descent will converge to the closest minimum, with no guarantee that this minimum will provide a satisfying generalization error. In a way, we are restricting our search to a very limited section of the loss function. Thus, in order to allow the algorithm to inspect the landscape for a better solution, randomness is added to its movement [5, 9]. **Stochastic Gradient Descent** picks a a batch of samples randomly at each iteration, and uses the sampled batch to calculate an "approximate gradient". The update looks as follows:

$$w_t \leftarrow w_{t-1} - \eta \frac{1}{|B|} \sum_{i \in \mathcal{B}} \nabla U_i(w_{t-1}), \tag{1.3}$$

where B is the **sampled batch**. When $1 > |B| > n$, this is called **Mini-batch Stochastic Gradient Descent** [29]. The new, approximated gradient might no longer indicate the "steepest" descent. In a sense, it allows the algorithm to potentially "climb up" on a hill and look for a better minimum. The smaller the batch size, the more noise is induced, and the more the computational cost is reduced. However, we must ensure that, overall, the algorithm is moving in the right direction (minimizing error), and thus a batch size must be chosen that provides a reasonable approximation to the full gradient [29]. Thus, mini-batch stochastic gradient descent is designed to speed up convergence and overcome local minima in search for a better minimum on the landscape. However, although introducing noise has the stated benefits, it can also pose some new issues to consider. As we introduce randomness, the algorithm becomes more sensitive to the choice of step size, and risks to converge slower or even diverge [29]. This is particularly true for ill-conditioned problems (landscapes for which different features would benefit from very different learning speeds), for example a landscape that is very flat in one direction and very steep in another [29]. A way to reduce noise and speed up convergence is using **momentum** [23, 29]. In the momentum method, the gradient is replaced with a weighted average of past gradients, as follows:

$$
\begin{aligned}
v_t &\leftarrow \beta v_{t-1} + \frac{1}{|B|} \sum_{i \in \mathcal{B}} \nabla U_i(w_{t-1}), \\
w_t &\leftarrow w_{t-1} - \eta v_{t-1},
\end{aligned}
$$

where v is the **weighted average of past gradients**, and β is a constant, $\beta \in [0, 1)$, that controls how "far into the past" we should be looking. The momentum variable v "remembers" past gradient values, and adjusts the direction of the next step based on them.

1.2.3 Adaptive Algorithms

Adding randomness to gradient descent has made it less prone to converge to suboptimal local minima, while momentum aids the noisy algorithm to move in the right direction faster. However, we have yet to address the issue of picking a proper step size. Adaptive algorithms are a class of optimization algorithms designed to address this concern, by introducing a step size that is reduced dynamically on a per-coordinate basis [29]. Parameters will thusly each be updated depending on their importance. Such an adaptive algorithm is **Adagrad** [7, 29]. The idea behind Adagrad is to perform larger updates for parameters with consistently steep gradients, and smaller updates for parameters with less steep gradients, as follows:

$$\begin{aligned} s_t &\leftarrow s_{t-1} + (\nabla U_i(w_{t-1}))^2, \\ w_t &\leftarrow w_{t-1} - \frac{\eta}{\sqrt{s_{t-1} + \epsilon}} \cdot \nabla U_i(w_{t-1}). \end{aligned}$$

where s is the sum of past gradient variances, ϵ is a small constant, and η is the step size. For simplicity, we use ∇U_i to refer to a mini-batch. We can notice that if a particular feature w has had large gradients in past iterations, then s **will be large** and thus **step size will be divided by a larger quantity**. Conversely, if w had small gradients in past iteration, then s **will be small** and **step size will be divided by a smaller quantity**. An issue that Adagrad faces is that s becomes increasingly large. **RMSProp** [26] is a version of Adagrad that "normalizes" s to avoid this issue [29]:

$$s_t \leftarrow \beta s_{t-1} + (1 - \beta)(\nabla U_i(w_{t-1}))^2, \tag{1.4}$$

where $\beta \in (0, 1)$. In order to further improve RMSProp, we can combine it with momentum, and the resulting algorithm is known as **Adam** [13, 29]:

$$\begin{aligned} v_t &\leftarrow \beta_1 v_{t-1} + (1 - \beta_1)\nabla U_i(w_{t-1}), \\ s_t &\leftarrow \beta_2 s_{t-1} + (1 - \beta_2)(\nabla U_i(w_{t-1}))^2, \\ w_t &\leftarrow w_{t-1} - \frac{\eta}{\sqrt{s_{t-1} + \epsilon}} \cdot v_{t-1}. \end{aligned}$$

Now, let us examine the transformation we will apply to the loss landscape in order to improve the state-of-art algorithms introduced above.

Chapter 2

Landscape Modification

2.1 Transformation of the Loss Landscape

Unlike all previously introduced algorithms that focus on adjusting the direction and/or step size of gradient-based methods, landscape modification proposes *transforming* the loss landscape itself, to one that is "easier" to optimize, and that would allow algorithms introduced in Chapter 1 to converge to better minima and potentially provide a lower generalization error. In [4], Choi introduces the following state-dependent transformation applied to a given loss function $U(x)$:

$$H(x) = \int_{U_{min}}^{U(x)} \frac{1}{f((u-c)_+) + \epsilon} du, \tag{2.1}$$

where U_{min} is the minimum point of $U(x)$, c is a constant and $c \geq U_{min}$, ϵ is a positive small constant, and $f : \mathbb{R}^d \to \mathbb{R}^+$ is a positive, twice differentiable and bounded function, with $f(0) = f^{'}(0) = f^{''}(0) = 0$.

2.1.1 Simulated Annealing with Langevin Diffusion

The transformation in Eq. 2.1 was introduced in the context of simulated annealing based on Langevin diffusion [4]. Simulated annealing [14] is an optimization method inspired by metallurgy, where, in order to increase ductility and reduce hardness, metal is heated and later gradually cooled down [6]. The heat gives particles freedom to move around and find the most "stable" spot before coming to a rest as the temperature is reduced. In optimization, the equivalent would be inducing noise to an algorithm, allowing it to examine the landscape for optima, and then gradually reducing this noise in order for the algorithm to settle in the minimum it found [5, 14]. Thus, **simulated annealing is very similar to stochastic gradient descent**, which is what inspires us to examine the findings of [4] in the context of gradient-based machine learning optimization. Let us formally explore this similarity, starting from the Langevin equation:

$$dX(t) = -\nabla U(X(t))dt + \sqrt{2T}dB_t. \tag{2.2}$$

Intuitively, this stochastic differential equation describes the behaviour of a particle influenced by a force, represented by $-\nabla U$, and by random noise, represented by the d-dimensional Brownian motion B_t, at temperature T [21]. In order to apply this equation to a machine learning context, we can consider a discrete approximation [21]:

$$X_{t+1} - X_t = -dt\nabla U(X(t)) + \sqrt{2T}\sqrt{dt}\mathcal{N}(0, 1). \tag{2.3}$$

A simple substitution [21] would give us:

$$\underline{w_{t+1} - w_t = -\eta \nabla U(w_t)} + \sqrt{2T}\sqrt{\eta}\mathcal{N}(0,1). \tag{2.4}$$

The underlined section of Eq. 2.4 is exactly the definition of gradient descent introduced in Eq. 1.2, to which some noise is added [21]. If, instead of T, we consider $T(t)$ in the original Eq. 2.2 (i.e we introduce a *cooling schedule* by gradually reducing the temperature), we obtain the Langevin diffusion-based simulated annealing. Both stochastic gradient descent and this version of simulated annealing introduce noise, and, furthermore, a different amount of noise for each iteration. However, in Langevin diffusion the noise is additive in nature, and it gets gradually reduced, while in stochastic gradient descent it is contained in the gradient itself, and controlled by the choice of mini-batch (both by the size of the batch and by the choice of component samples). A further parallel can be drawn if we consider **kinetic simulated annealing**, which would be **the continuous equivalent of stochastic gradient descent with added momentum** [4]. We have:

$$\begin{aligned} dX(t) &= Y(t)dt, \\ dY(t) &= -\frac{1}{T(t)}Y(t)dt - \nabla U(X(t))dt + \sqrt{2}dB_t, \end{aligned} \tag{2.5}$$

where $Y(t)$ represents the momentum variable. We can notice that, much like the effect of momentum on stochastic gradient descent, the induced noise is reduced. Furthermore, under certain conditions, convergence is proved for both the simulated annealing described by Eq. 2.2 [2], as well as for its kinetic variant described in Eq. 2.5 [19]. For large enough t, we must consider the following logarithmic cooling schedule:

$$T(t) = \frac{E}{\ln t}, \tag{2.6}$$

where $E > E*$ [4]. As we can see, convergence is dependent on what we call the critical height, $E*$. The critical height can be understood as the highest hill the algorithms needs to climb from a local minimum to reach the global minimum [3], as visualised in Fig. 2.1.

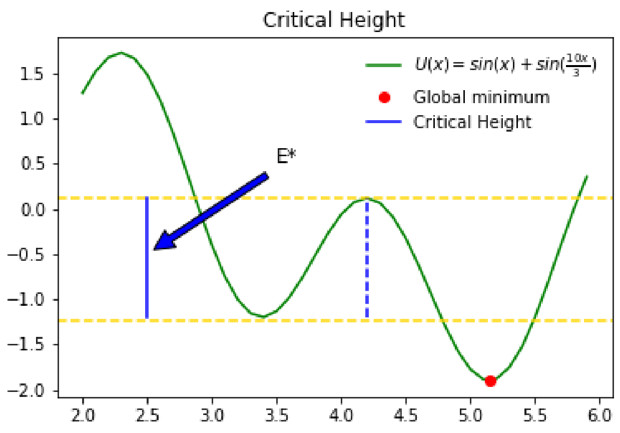


FIGURE 2.1: Visualizing the critical height.

2.1.2 Improved Adaptive Kinetic Simulated Annealing

In [8], a method is proposed to accelerate convergence of simulated annealing by modifying the Langevin diffusion equation in 2.2 in a manner that will reduce the critical height. The modified equation is as follows:

$$dX(t) = -\nabla U(X(t))dt + \sqrt{2(f((U(X(t)-c)_{+}) + T(t))}dB_t, \tag{2.7}$$

where f and c are as explained in Eq. 2.1. Convergence is proved in the same paper, and it follows the conditions of Eq. 2.6, with one significant modification, that the critical height E^* is replaced by a *clipped* critical height $c*$, where $c \leq E*$ [4]. An intuitive explanation for why this is an improvement can be obtained looking at the last term of Eq. 2.7, and from Fig. 2.2. When the loss value is less than c, we induce the same amount of noise as the original algorithm. However, when the loss value is greater than c, more noise is induced [3]. Namely, we accelerate the algorithm around local minima (with a faster cooling schedule), but we allow it to settle around the global minimum. Noise becomes, in this case, *state-dependent*. In [4], this idea is applied to *kinetic simulated annealing*, in-

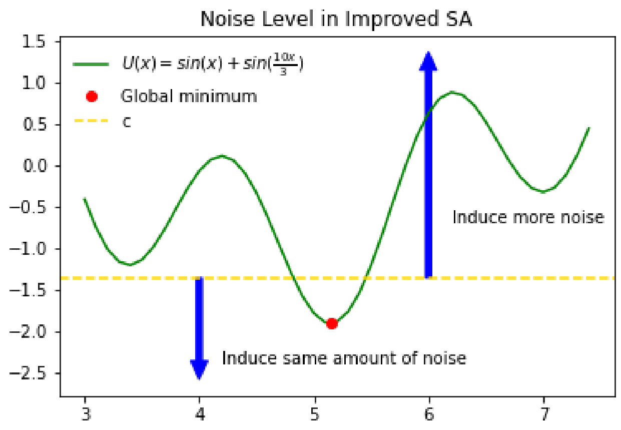


FIGURE 2.2: When $U(x) < c$, the noise induced is the same as the original simulated annealing algorithm. When $U(x) > c$, more noise is introduced than the original algorithm.

troduced in Eq. 2.5, in order to further accelerate convergence by adding the benefit of momentum. The innovative idea introduced in the paper is that the state-dependent noise can be embedded in the landscape [3], by *transforming* the objective function itself, and thus obtaining:

$$\begin{aligned} dX(t) &= Y(t)dt, \\ dY(t) &= -\frac{1}{T(t)}Y(t)dt - T(t)\nabla H(X(t))dt + \sqrt{2}dB_t, \end{aligned} \tag{2.8}$$

where

$$H(x) = \int_{U_{min}}^{U(x)} \frac{1}{f((u-c)_+) + \epsilon} du + \ln\left(f((U(x) - c)_+) + \epsilon\right). \tag{2.9}$$

We notice that Eq. 2.9 has an additional term that we omitted when we introduced the transformation in Eq. 2.1. This term is necessary in order to preserve some mathematical properties relevant to the proof of convergence offered in the paper. However, it is not relevant in a machine learning context, where proof of converge is intractable. Since the noise is captured in the first term, we can omit the second term for simplicity, and arrive at the final form of $H(x)$ in Eq. 2.1. Finally, the idea of **adaptive transformation** is introduced by allowing c in Eq. 2.1 to be the running minimum of the loss function $U(x)$, instead of a constant. Intuitively, we can understand this as the landscape being modified adaptively as the algorithm progresses [4].

2.2 Landscape Modification through $H(X)$

In order to understand the effect of this transformation, we consider a simple example. Let us take:

$$U(x) = sin(x) + sin\left(\frac{10x}{3}\right)$$

Next, let us apply $H(x)$, as defined in 2.1, taking $f(x) = x$:

$$H(x) = \int_{U_{min}}^{U(x)} \frac{1}{(u-c)_+ + \epsilon} du$$
$$H(x) = \int_{U_{min}}^{c} \frac{1}{\epsilon} du + \int_{c}^{U(x)} \frac{1}{(u-c)_+ + \epsilon} du$$

When $U(x) \leq c$ we have:

$$H(x) = \int_{U_{min}}^{U(x)} \frac{1}{\epsilon} du = \frac{1}{\epsilon}(U(x) - U_{min}) \tag{2.10}$$

When $U(x) > c$ we have:

$$H(x) = \frac{1}{\epsilon}(c - U_{min}) + \ln\left(\frac{(U(x) - c) + \epsilon}{\epsilon}\right) \tag{2.11}$$

If we take $c = U_{min}$ and $\epsilon = 1$, we can visualize the transformation in Fig. 2.3.

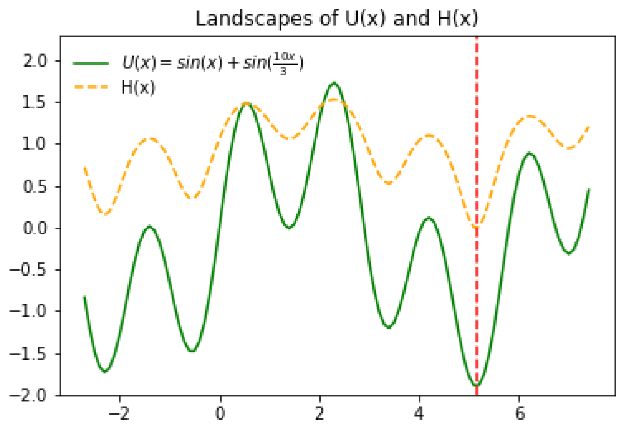


FIGURE 2.3: Applying $H(x)$ to $U(x)$, with $f(x) = x$, c the minimum of $U(x)$, and $\epsilon = 1$.

From 2.3, we can make several observations:

- $U(x)$ and $H(x)$ have the same stationary points. Hence, minimizing $H(x)$ is equivalent to minimizing $U(x)$.
- $H(x)$ "flattens" the loss landscape, which correspond to the idea of *clipped critical height* introduced in Section 2.1.2. Thus, the height of the hills is reduced, which should help algorithms climb them more easily in search of the global minimum.

2.2.1 H(x) Parameter Significance

Next, let us summarize the significance of each parameter:

- The choice $f(x)$ will determine the scale of the transformation. We can notice in 2.11 from the example above that $f(x) = x$ causes a log-scale transformation, when $U(x) > c$. When $U(x) \leq c$, the transformation will always be linear, irregardless of the choice of f, as we can see in Eq. 2.10. Another simple example we could examine is $f(x) = x^2$. In that case, when $U(x) > c$, we will have:

$$H(x) = \frac{1}{\epsilon}(c - U_{min}) + \frac{1}{\sqrt{\epsilon}} arctan\left(\frac{(U(x) - c)}{\sqrt{\epsilon}}\right)$$

  Thus, $f(x) = x^2$ causes a arctan-scale transformation. We can visualise the two transformations in Fig. 2.4. Intuitively, we expect $f(x) = x^2$ to cause more "flattening" than $f(x) = x$, as we are dividing the gradient by a larger value. This is confirmed in graph as well.

- The choice of c will determine by how much the height of the hills will be cut. As shown in Fig. 2.5, we have the following observations:

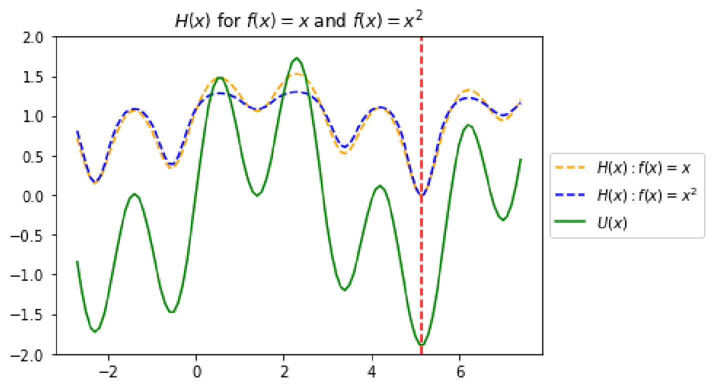


FIGURE 2.4: Examining the effect of f(x) on H(x), when $\epsilon = 1$ and $c = U_{min}$.

  - When $c \geq U_{max}$, the function is not "flattened" at all, and the height of the hills remains the same ($H(x)$ is identical for all $c \geq U_{max}$ in this case, which is why the pink and yellow lines overlap in the Fig. 2.5). Given Eq. 2.10, $U(x)$ will just be shifted by a constant (more specifically, by U_{min}).
  - When $c < U_{max}$, the landscape is modified. The lower the c, the "flatter" our landscape will become.

- The effect of ϵ can be observed in Fig. 2.6. It has an effect inverse to that of c: the higher ϵ, the more the landscape is flattened. We can say it "boosts" the effect of c. Thus, it might be of no practical value to vary both ϵ and c. In further investigations, we will keep ϵ constant at a fixed value such as 1, and vary only c.

2.2.2 The Gradient of $H(x)$

Depending on the choice of f, $H(x)$ might be difficult to compute. However, in practice, the algorithms we introduced in Chapter 1 only require

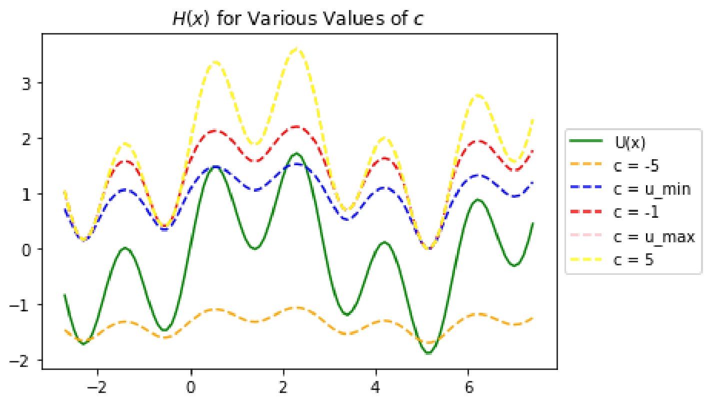


FIGURE 2.5: Examining the effect of c on $H(x)$, when $f(x) = x$ and $\epsilon = 1$.

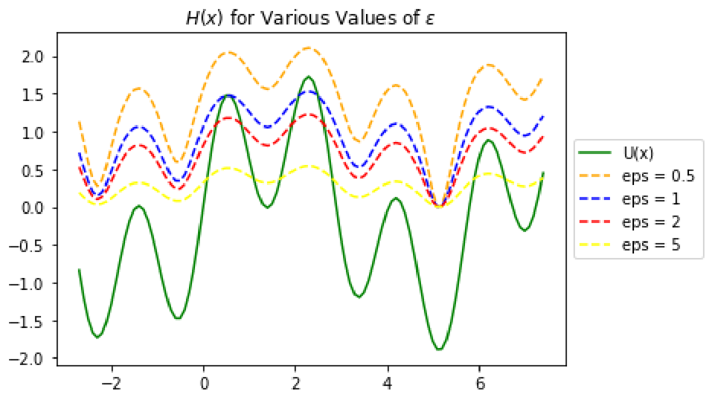


FIGURE 2.6: Examining the effect of ϵ on $H(x)$, when $f(x) = x$ and $c = U_{min}$.

the **gradient** of $U(x)$. Accordingly, we can directly consider the gradient of $H(x)$:

$$\nabla H(x) = \frac{1}{f((U(x) - c)_{+}) + \epsilon} \nabla U(x) \tag{2.12}$$

We can understand equation 2.12 as applying a state-dependent preconditioning to the gradient. Thus, landscape modification can easily be added to the state-of-art algorithms introduced in Chapter 1.

Chapter 3

Numerical Results

3.1 Introduction

In this Chapter we will present the methodology and results of experiments we ran in order to assess the performance of landscape modification in the context of machine learning optimization. We are mainly interested to examine whether applying the gradient preconditioning from Eq. 2.12 to the algorithms introduced in Chapter 1 improves their performance. By flattening the landscape, we hope to help the algorithm converge to a better minimum. However, recall from our discussion in Chapter 1 that, although minimizing error is the goal of optimization, the end-goal of training a model is generalization, i.e to increase prediction accuracy. Thus, we will report on two aspects: whether landscape modification helped the algorithm reach a lower point on the loss landscape (training error), and whether it improved testing accuracy.

3.2 Methodology

3.2.1 Experiment setting

We are using two datasets: CIFAR10 and CIFAR100, each consisting of 60000 color images (50000 training and 10000 testing) that belong to 10 different classes, for CIFAR10, and to 100 different classes, for CIFAR100. On these two datasets, we train different classification models using Cross-Entropy as a loss function, and SGD, Adam, Adagrad, or RMSProp as optimizers. The Python library Pytorch is used for the process. All four optimizers are pre-implemented in Pytorch, and we used these implementations to create four new optimizers that add landscape modification to the gradient as per Eq. 2.12. For example, the update for stochastic gradient descent with landscape modification will look like:

$$w_t \leftarrow w_{t-1} - \eta \frac{1}{|B|} \sum_{i \in \mathcal{B}} \frac{1}{f((U(w_{t-1}) - c)_+) + \epsilon} \nabla U_i(w_{t-1}). \tag{3.1}$$

3.2.2 Method and Metrics

We will follow a very simple methodology:

1. We fix a seed to ensure reproducibility (we want to induce the same randomness for each run).

2. We train the model using SGD/Adam/Adagrad/RMSProp with a set of parameters, as per each algorithm requires, for a number of epochs n (see Appendix A for details).

3. We train the model using SGD/Adam/Adagrad/RMSProp **with landscape modification** and with the same set of parameters, for

the same number of epochs n. For simplicity, we will be using $f(x) = x$, $\epsilon = 1$ and c the running minimum of $U(x)$ as parameters for the landscape transformation.

4. We compare the following metrics averaged over three runs: average training error over last 20 mini-batches and test accuracy.

3.2.3 Models - Convolutional Neural Networks

Convolutional neural networks (CNNs) are deep learning algorithms designed for the purpose of vision-related tasks, such as image/video recognition/classification. Presently, they are the most widely used algorithms in this area, which is why we wish to test the performance of landscape modification on a few selected CNN architectures. Given an input image, a CNN will be able to identify features of this image, such as corners and edges, and determine how important or representative each feature is. Previously, feature extraction would be done manually (humans would determine what features the machine is looking for), but the revolutionary aspect of CNNs is that they learn features by themselves [25]. The building block of a CNN is the process of convolution, by which kernels "scan" an image in search of features [16], as described in Fig. 3.1. We can apply many layers of convolution, in order to extract more and more detailed/nuanced characteristics of an image, and a CNN will learn the convolution kernels through backpropagation [18]. In order to allow nonlinearity in our model, an activation function is applied to the convolved image [18]. Furthermore, in order to reduce noise and decrease the computational power required, pooling layers are added after convolutional layers to summarize the information and reduce the dimension

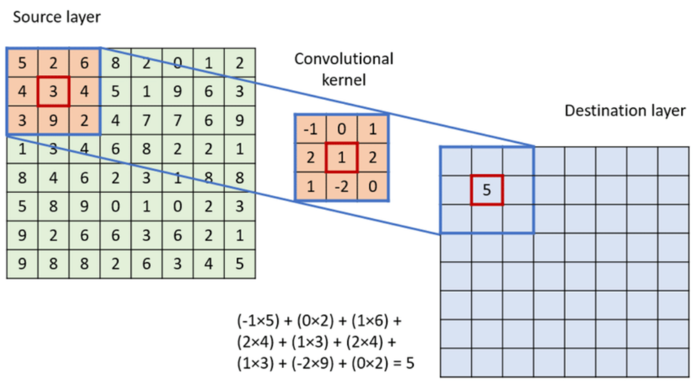


FIGURE 3.1: We slide a kernel over an input image (represented as an array), we calculate the element wise multiplication between the selected window on the original photo and the kernel, and finally we sum all the resulting elements [22].

of the output. Finally, the extracted features are flattened (to a 1D array) and passed on to an artificial neural network that performs classification [18]. The whole process can be visualized in Fig. 3.2.

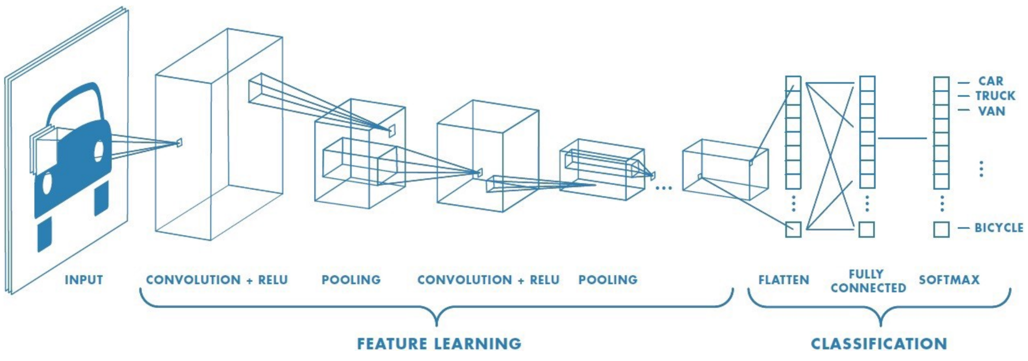


FIGURE 3.2: A basic CNN architecture: Convolutional layers activated by ReLu accompanied by pooling layers. Flattened output is fed into a neural network that generates predictions [18].

3.2.4 VGG and ResNet

From our discussion in the previous section, it can be inferred that increasing the number of convolutional layers in a CNN will allow us to capture more information about an input image. An example of such a deep CNN widely used in image classification is VGG16 [24], shown in Fig. 3.9. VGG16 over-performed its precursors by a few unique characteristics, such as convolutional layers using small filters (3x3) [28]. Besides the 16-layer VGG, there exists a variant of 19 layes: VGG19. After VGG, literature has questioned the possibility of even deeper architectures. However, deepening the network poses the challenge of vanishing

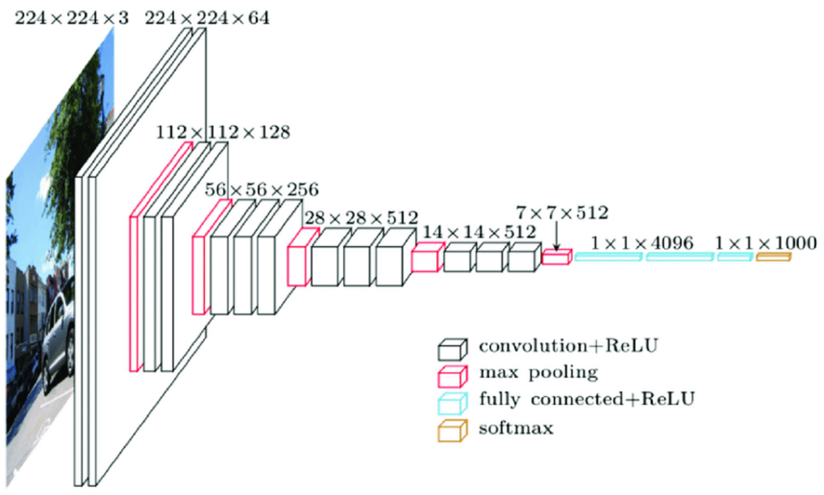


FIGURE 3.3: VGG16 Architecture [20]: 16 layers with trainable weights, comprised of blocks of convolutional layers with 3x3 filters of stride 1, followed by max-pooling layers of stride 2. The neural network at the end has 3 fully-connected layers [28].

gradients and reduced training accuracy [10]. The activation function in convolutional layers often maps a large input space to a small one, which causes derivatives to be small, and the gradient to approach zero as we backpropagate through the layers [27]. Thus, for the first layers in the network, the update steps of our gradient-based optimizer will be too small, and the optimization algorithm might get stuck in a suboptimal point. ResNet [10] is a CNN architecture specifically designed to overcome this issue, by allowing the algorithm to "skip" layers, as illustrated in Fig. 3.4, and thus prevent the gradients from vanishing during backpropagation. In our experiments, we use the 16-layer version of VGG, VGG16, and a further improved version of ResNet, a 110-layer deep network we call PreResNet110 [11]. The implementations used are taken from [12].

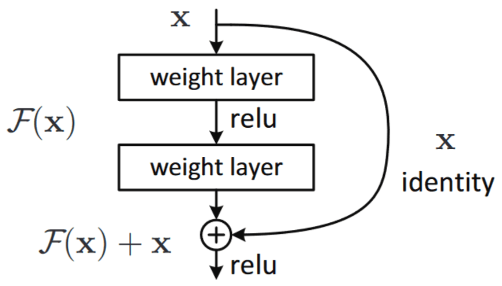
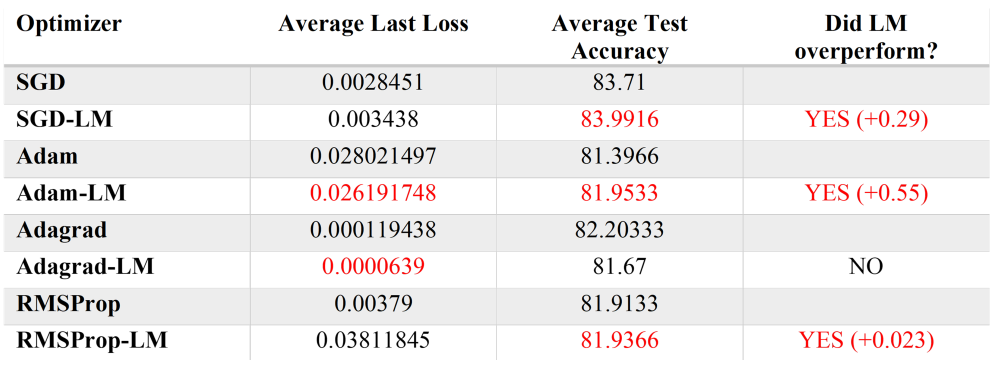

x
weight layer
$\mathcal{F}(\mathbf{x})$
relu
weight layer
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

FIGURE 3.4: ResNet allows the algorithm to skip layers by passing the output of one layer to a layer further down in the architecture. An identity function is used to preserve the gradient [10].

3.3 CIFAR10 Results

In this section we report the numeric results from training classifiers on CIFAR10. Firstly, in order to assess whether landscape modification is a viable proposition, we trained a few "shallow" CNNs. We report the results for NN2 [17], a CNN with 6 convolutional layers, in Fig. 3.5. We can notice that, in these preliminary experiments, adding landscape modification **improves accuracy** when **SGD (with momentum)** and **Adam** were used, **by ~ 0.3 and ~ 0.6 percentage points** respectively. Very small or no improvement is shown for RMSProp and Adagrad. Interestingly, this pattern is present in the results for **PreResNet**, summarized in Fig. 3.6, where **landscape modification improved SGD performance by ~ 0.2 percentage points** (see example in Fig. 3.7), and **Adam performance by ~ 1.3 percentage points**. A possible explanation for this pat-



Optimizer	Average Last Loss	Average Test Accuracy	Did LM overperform?
SGD	0.0028451	83.71	
SGD-LM	0.003438	83.9916	YES (+0.29)
Adam	0.028021497	81.3966	
Adam-LM	0.026191748	81.9533	YES (+0.55)
Adagrad	0.000119438	82.20333	
Adagrad-LM	0.0000639	81.67	NO
RMSProp	0.00379	81.9133	
RMSProp-LM	0.03811845	81.9366	YES (+0.023)

FIGURE 3.5: Results of Landscape Modification for NN2 on CIFAR10.

tern could be the presence of momentum in the update step of the first two optimizers. As mentioned in Section 2.1.2, the very idea of landscape modification is introduced in the context of kinetic simulated annealing, which incorporates a momentum variable. Intuitively, landscape modification "flattens" the objective function, and momentum could be aiding the algorithm to move more efficiently in the direction of the minimum on this flattened landscape. We also notice that, for NN2, better results did not necessarily correspond to a lower training error, and the same can be said for PreResNet. The most interesting findings were the ones obtained from training **VGG**. Here, landscape modification brought **a considerable improvement in accuracy, across all optimizers**, as can be seen in Fig. 3.8. Notably, for **SGD and Adam**, the algorithm got stuck in a sub-optimal point 2/3 runs with the original optimizer (see example in Fig. 3.7), while landscape modification prevented it from doing so altogether. Thus, we see considerable improvments of $\sim$ **52 percentage points** and $\sim$ **37 percentage points** respectively. In light of our findings in Chapter 2, we can interpret this as landscape modification helping the algorithm escape local minima. For **Adagrad** we notice an improvement of $\sim$ **10 percentage points**, as visualised in Fig. 3.9, and, for **RMSProp**, of $\sim$ **1.9 percentage points**. For VGG, all landscape modification variants of the optimizers resulted in a lower error, alongside increased accuracy. Finally, the best accuracy across all experiments for CIFAR10, **89.57**%, was achieved with PreResNet, when SGD with landscape modification was used.

Optimizer	**Average Last Loss**	**Average Test Accuracy**	**Did LM overperform?**
SGD	0.0313	89.38	
SGD-LM	0.028	89.57	YES (+0.19)
Adam	0.03	87.34	
Adam-LM	0.03	88.68	YES (+1.34)
Adagrad	0.0362	79.68	
Adagrad-LM	0.0392	79.40	NO
RMSProp	0.0261	89.25	
RMSProp-LM	0.0241	88.75	NO

FIGURE 3.6: Results of Landscape Modification for PreResNet on CIFAR10.

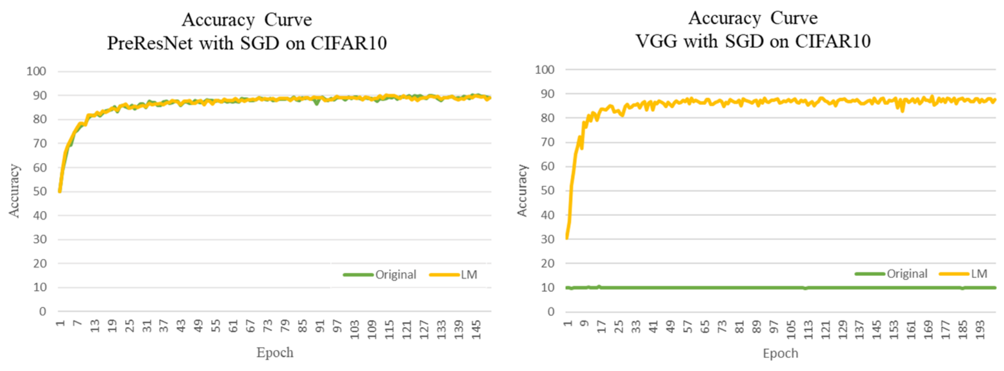


FIGURE 3.7: Left: For PreResNet with SGD on CIFAR10, landscape modification brings a minor improvement, but the behaviour is very similar to the original algorithm. Right: For VGG with SGD on CIFAR10, landscape modification helps the algorithm escape local minima.

Optimizer	**Average Last Loss**	**Average Test Accuracy**	**Did LM overperform?**
SGD	1.36	35.18	
SGD-LM	0.296	86.90	YES (+51.72)
Adam	2.30	48.66	
Adam-LM	0.319	85.49	YES (+36.83)
Adagrad	0.0622	76.04	
Adagrad-LM	0.00818	85.94	YES (+9.9)
RMSProp	0.385	84.87	
RMSProp-LM	0.272	86.73	YES (+1.86)

FIGURE 3.8: Results of Landscape Modification for VGG on CIFAR10.

3.4 CIFAR100 Results

Now, let us examine the experiments performed on CIFAR100. For **PreResNet**, the results can be seen in Fig. 3.10. We can notice that the pattern we observed for CIFAR10 holds true for CIFAR100. When using **PreResNet**, landscape modification improved accuracy when **SGD** or **Adam** were used, by $\sim$ **1.1** and $\sim$ **0.7 percentage points** respectively. While no improvement can be noticed for Adagrad, **RMSProp** accuracy increased

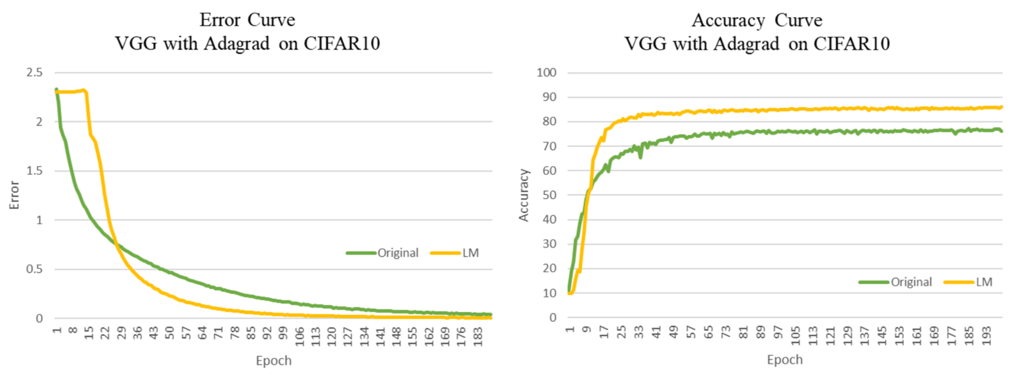


FIGURE 3.9: Left: VGG with Adagrad and landscape modification on CIFAR10 converges faster. Right: It achieves a better accuracy as well.

by $\sim$ **0.5 percentage points**, unlike what we saw for CIFAR10. For **VGG**, we once again notice **an improvement across all optimizers**, in Fig. 3.11. For **SGD** (see Fig. 3.12) and **Adam**, we notice a $\sim$ **4.3** and a $\sim$ **0.7 percentage points** improvement respectively. For **RMSProp** and **Adagrad**, the original algorithm got stuck in sub-optimal points, while landscape modification provided a better result, showing sizable improvements of $\sim$ **14** and $\sim$ **39 percentage points** respectively. This is the converse of what happened with CIFAR10, where this situation occurred for SGD and Adam. Once again, when PreResNet was used, improvement in accuracy was not necessarily accompanied by a lower training error. For VGG, however, higher accuracy of the landscape modification variant over the original optimizer corresponds to a lower training error. Finally, the highest accuracy for CIFAR100 classification was achieved when PreResNet was used with the landscape modification variant of SGD, and the value is **60.96**%.

Optimizer	**Average Last Loss**	**Average Test Accuracy**	**Did LM overperform?**
SGD	1.04	59.84	
SGD-LM	1.06	60.96	YES (+1.12)
Adam	0.6	40.41	
Adam-LM	0.59	41.15	YES (+0.74)
Adagrad	2.76	24.355	
Adagrad-LM	2.74	24.04	NO
RMSProp	0.098	59.84	
RMSProp-LM	0.093	60.3	YES (+0.46)

FIGURE 3.10: Results of Landscape Modification for Pre-ResNet on CIFAR100.

Optimizer	**Average Last Loss**	**Average Test Accuracy**	**Did LM overperform?**
SGD	1.69	54.34	
SGD-LM	1.34	58.62	YES (+4.28)
Adam	0.13	55.2	
Adam-LM	0.12	55.9	YES (+0.7)
Adagrad	1.11	32.9	
Adagrad-LM	0.75	46.54	YES (+13.64)
RMSProp	4.44	3.31	
RMSProp-LM	2.59	41.87	YES (+38.56)

FIGURE 3.11: Results of Landscape Modification for VGG on CIFAR100.

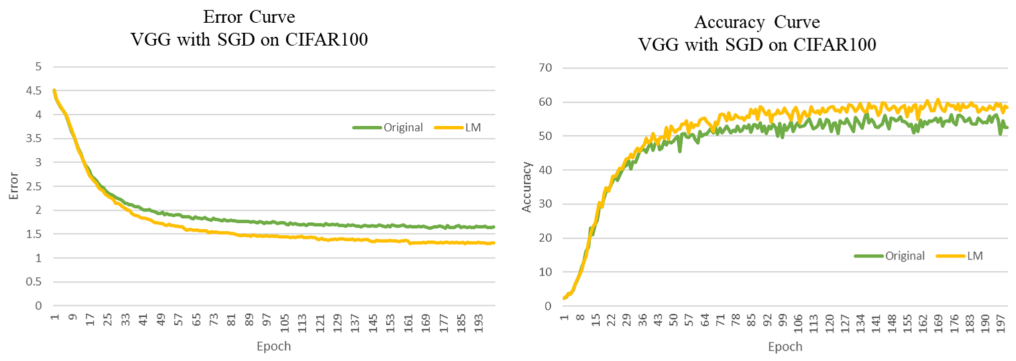


FIGURE 3.12: Left: For VGG on CIFAR100, the error decreases faster when the landscape modification variant of SGD is used. Right: A better accuracy is achieved as well.

Chapter 4

Conclusion

In Chapter 2, we provided a mathematical intuition for why landscape modification, as introduced in [4], can help improve the performance of gradient-based optimizers in machine learning applications. By "flattening" the loss landscape, we can reduce the height of "the hills" that an optimization algorithm must climb in order to overcome local, suboptimal minima, thus providing a "better" landscape for optimization. We backed our claims with theoretical results achieved in the field of simulated annealing, presented in [4].

In Chapter 3, we conducted numerical experiments to test the hypothesis that landscape modification can improve the performance of state-of-art optimization algorithms introduced in Chapter 1, and we presented our results. We summarize the following concluding remarks. When we trained a classification model on CIFAR10 and CIFAR100 using PreResNet, adding landscape modification through the preconditioning of the gradient as per Eq. 2.1 showed an improvement in training accuracy between 0.1 and 2 percentage points, when Stochastic Gradient Descent with Momentum or Adam were used as optimizers. We hypothesise that the presence of a momentum variable could be a possible cause for this pattern. For the experiments using PreResNet, higher accuracy for the landscape modification variant did not seem to be correlated with a lower training error, as compared to the "vanilla' optimizer. When VGG was used, landscape modification provided an improvement in testing accuracy across all optimizers, for both CIFAR10 and CIFAR100. In some cases, "vanilla" optimizers got trapped in a suboptimal point early on in the optimization process, while the landscape modification variants continued the minimization and converged to a better minima, which is why we notice significant improvements of 30 to 50 percentage points. In other cases, both "vanilla" optimizers and their landscape modification variants seemed to scan the landscape throughout the training process, but the latter still settled in better minima, providing an improvement of 0.7 to 13 percentage points. When VGG was used, an improvement in accuracy for landscape modification was accompanied by a lower training error. Overall, SGD with landscape modification was the optimizer that provided the highest testing accuracy, for classifying both CIFAR10 and CIFAR100.

Finally, we can conclude that landscape modification has a high potential to improve the performance of machine learning applications. However, our exploration was limited to two datasets and two architectures, and thus further experiments would be required to assess the robustness of landscape modification. Furthermore, we restricted our experiments to use a fixed set of hyperparameters. Additional findings might be added when different parameters are used for landscape modification and for the underlying optimizers.

Bibliography

[1] Léon Bottou, Frank E. Curtis, and Jorge Nocedal. *Optimization Methods for Large-Scale Machine Learning*. 2016. DOI: 10.48550/ARXIV.1606.04838. URL: https://arxiv.org/abs/1606.04838.

[2] Tzuu-Shuh Chiang, Chii-Ruey Hwang, and Shuenn Jyi Sheu. "Diffusion for global optimization in Rˆn". In: *SIAM Journal on Control and Optimization* 25.3 (1987), pp. 737–753.

[3] Michael Choi. *Algorithms Seminar - E3: Michael Choi*. Youtube. 2020. URL: https://www.youtube.com/watch?v=nC0dyeXZWAo&t=1879s&ab_channel=WarwickStatisticsDepartment.

[4] Michael C. H. Choi. *On the convergence of an improved and adaptive kinetic simulated annealing*. 2020. arXiv: 2009.00195 [math.PR].

[5] Pierre Collet and Jean-Philippe Rennard. "Stochastic Optimization Algorithms". In: (2007). DOI: 10.48550/ARXIV.0704.3780. URL: https://arxiv.org/abs/0704.3780.

[6] Corrosionpedia. *Annealing*. 2018. URL: https://www.corrosionpedia.com/definition/90/annealing-metallurgy.

[7] John Duchi, Elad Hazan, and Yoram Singer. "Adaptive subgradient methods for online learning and stochastic optimization." In: *Journal of machine learning research* 12.7 (2011).

[8] Haitao Fang, Minping Qian, and Guanglu Gong. "An improved annealing method and its large-time behavior". In: *Stochastic Processes and their Applications* 71.1 (1997), pp. 55–74. ISSN: 0304-4149. DOI: https://doi.org/10.1016/S0304-4149(97)00069-0. URL: https://www.sciencedirect.com/science/article/pii/S0304414997000690.

[9] Lauren A Hannah. "Stochastic optimization". In: *International Encyclopedia of the Social & Behavioral Sciences* 2 (2015), pp. 473–481.

[10] Kaiming He et al. "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[11] Kaiming He et al. "Identity mappings in deep residual networks". In: *European conference on computer vision*. Springer. 2016, pp. 630–645.

[12] Pavel Izmailov et al. *Averaging Weights Leads to Wider Optima and Better Generalization*. 2018. DOI: 10.48550/ARXIV.1803.05407. URL: https://arxiv.org/abs/1803.05407.

[13] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

[14] Scott Kirkpatrick, C Daniel Gelatt Jr, and Mario P Vecchi. "Optimization by simulated annealing". In: *science* 220.4598 (1983), pp. 671–680.

[15] Claude Lemaréchal. "Cauchy and the gradient method". In: *Doc Math Extra* 251.254 (2012), p. 10.

[16] Vijaysinh Lendave. *What is a Convolutional Layer?* 2021. URL: https://analyticsindiamag.com/what-is-a-convolutional-layer/.

[17] Siddharth M. *Convolutional Neural network - PyTorch implementation on CIFAR-10 Dataset*. 2021. URL: https://www.analyticsvidhya.com/blog/2021/09/convolutional-neural-network-pytorch-implementation-on-cifar10-dataset/.

[18] Mayank Mishra. *Convolutional neural Networks, Explained*. 2020. URL: https://towardsdatascience.com/convolutional-neural-networks-explained-9cc5188c4939.

[19] Pierre Monmarché. "Hypocoercivity in metastable settings and kinetic simulated annealing". In: *Probability Theory and Related Fields* 172.3 (2018), pp. 1215–1248.

[20] Will Nash, Tom Drummond, and Nick Birbilis. "A review of deep learning in the study of materials degradation". In: *npj Materials Degradation* 2 (Dec. 2018). DOI: 10.1038/s41529-018-0058-x.

[21] Kirill Neklyudov. *Day 5, lecture 3. Langevin dynamics for sampling and global optimization*. Youtube. 2019. URL: https://www.youtube.com/watch?v=3-KzIjoFJy4&t=171s&ab_channel=BayesGroup.ru.

[22] Damian Podareanu et al. "Best Practice Guide - Deep Learning". In: (Feb. 2019). DOI: 10.13140/RG.2.2.31564.05769.

[23] Sebastian Ruder. *An overview of gradient descent optimization algorithms*. 2016. DOI: 10.48550/ARXIV.1609.04747. URL: https://arxiv.org/abs/1609.04747.

[24] Karen Simonyan and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition". In: *arXiv preprint arXiv:1409.1556* (2014).

[25] Matthew Stewart. *Simple Introduction to Convolutional neural Networks*. 2019. URL: https://towardsdatascience.com/simple-introduction-to-convolutional-neural-networks-cdf8d3077bac.

[26] Tijmen Tieleman, Geoffrey Hinton, et al. "Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude". In: *COURSERA: Neural networks for machine learning* 4.2 (2012), pp. 26–31.

[27] Chi-Feng Wang. *The Vanishing Gradient Problem*. 2019. URL: https://towardsdatascience.com/the-vanishing-gradient-problem-69bf08b15484.

[28] Jerry Wei. *VGG Neural Networks: the Next Step After AlexNet*. 2019. URL: https://towardsdatascience.com/vgg-neural-networks-the-next-step-after-alexnet-3f91fa9ffe2c.

[29] Aston Zhang et al. *Dive into Deep Learning*. https://d2l.ai. 2020.

Appendix A

Table of Hyperparameters Used in Experiments

Below we list the hyperparameters used when conducting the experiments in Chapter 3. Number of epochs for training as well as SGD parameter values are inspired by [12]. If not listed, default value (as per the Pytorch implementation) is used.

NN2	**Epochs**	**Learning Rate**	**Momentum**	**Weight Decay**	**Alpha**	**Betas**
SGD	50	0.01	0.9	0	N/A	N/A
Adam	50	0.001	N/A	0	N/A	(0.9, 0.999)
Adagrad	50	0.01	N/A	0	N/A	N/A
RMSProp	50	0.001	0	0	0.99	N/A
PreResNet						
SGD	150	0.1	0.9	0	N/A	N/A
Adam	150	0.001	N/A	0	N/A	(0.9, 0.999)
Adagrad	150	0.01	N/A	0	N/A	N/A
RMSProp	150	0.001	0	0	0.99	N/A

VGG						
SGD	200	0.05	0.9	0	N/A	N/A
Adam	200	0.001	N/A	0	N/A	(0.9, 0.999)
Adagrad	200	0.01	N/A	0	N/A	N/A
RMSProp	200	0.001	0	0	0.99	N/A

TABLE A.1: Hyperparameters used for CIFAR10 experiments.

NN2	**Epochs**	**Learning Rate**	**Momentum**	**Weight Decay**	**Alpha**	**Betas**
SGD	50	0.01	0.9	0	N/A	N/A
Adam	50	0.001	N/A	0	N/A	(0.9, 0.999)
Adagrad	50	0.01	N/A	0	N/A	N/A
RMSProp	50	0.001	0	0	0.99	N/A
PreResNet						
SGD	150	0.1	0.9	0	N/A	N/A
Adam	150	0.001	N/A	0	N/A	(0.9, 0.999)
Adagrad	150	0.01	N/A	0	N/A	N/A
RMSProp	150	0.001	0	0	0.99	N/A
VGG						
SGD	200	0.05	0.9	0	N/A	N/A
Adam	200	0.001	N/A	0	N/A	(0.9, 0.999)
Adagrad	200	0.01	N/A	0	N/A	N/A
RMSProp	200	0.001	0	0	0.99	N/A

TABLE A.2: Hyperparameters used for CIFAR100 experiments.

Appendix B

Example of Python Script for Landscape Modification Experiments

Below we present a Python implementation of a landscape modification experiment that trains PreResNet110 for CIFAR10 classification, using both SGD and SGD with landscape modification, and saves a series of relevant metrics.
Other code used for generating the results in this thesis can be found at `https://github.com/IoanaTodea22/LandscapeModification.git`.

CIFAR10 Classification using PreResNet110

Comparing results of SGD and SGD with landscape modification

Firstly, we import the necessary libraries.

```
import numpy
import pandas as pd
import random
import matplotlib.pyplot as plt
import torch
import torchvision
import torchvision.transforms as transforms
import torch.optim as optim
import torch.nn as nn
import os
from google.colab import files
# import modified optimizer
from SGD_IKSA import SGD_IKSA
# import model
from preresnet import PreResNet
```

We set a seed to ensure reproducibility.

```
# We set the seed at the beginning of each experiment to
# ensure reproducibility.
def setup_seed(s):
    random.seed(s)
    numpy.random.seed(s)
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False
```

Next, we define a function that will perform an experiment,

by training PreResNet on CIFAR10, using both SGD and SGD with

landscape modification, and report the results.

Training process inspired by:
https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html,
https://pytorch.org/docs/stable/notes/randomness.html

```
def compare_SGD_SGD_LM(device,
                       seed,
                       model,
                       trainset,
                       testset,
                       batch_size,
                       no_epochs,
                       lr,
                       momentum,
                       weight_decay,
                         optimizer_type,
                         LM_f,
                         LM_c,
                         LM_c_run_min):
    """This function trains a given model on a given train dataset,
    using both SGD and SGD with landscape modification. Subsequently,
    it computes the testing accuracy on the given test data.


    Args:
        device : CPU/GPU used for training process;
        seed : seed to be used in the random processes involved;
        model : classification model to be used;
        trainset : training data;
        testset : testing data;
        batch_size: batch-size to be used in training;
        no_epochs : number of epochs that the model will be
        trained for;
        lr : learninf rate of SGD optimizer;
        momentum : momentum value of the SGD optimizer;
        weight_decay : weight decay value of the SGD optimizer;
        optimizer_type : bool indicating whether landscape
        modification will be used;
        LM_f : function to be used when performing landscape
        modification
        LM_c : c value to be used when performing landscape
        modification
        LM_c_run_min : bool indicating whether c will be
        taken as the running minimum of U(x), with LM_c as initial
value.

    Returns:
        [dict]: a dictionary that contains: a dictionary of
parameters, a list
        of loss values, a list of c values, a list of average loss
values,
        a list of testing accuracy values, and the final testing
accuracy value.
    """
    # We want to save the hyperparameters used for this experiment in
a dictionary.
    param_dict = {"no_epochs": no_epochs,
                  "batch_size": batch_size,
                  "lr": lr,
                  "momentum": momentum,
                  "wd": weight_decay,
                  "optimizer_type": optimizer_type,
                  "LM_f": LM_f.__name__,
                  "LM_c": LM_c,
                  "LM_c_run_min": LM_c_run_min}
    # We keep the loss values.
    loss_list = []
    # We keep the c values.
    c_list = []
    # We save a list of loss values averaged every 20 minibatches.
    average_loss_list = []
    # We save a list of accuracy scores for each epoch.
    accuracy_list = []

    # We set up a seed.
    setup_seed(seed)

    # We preserve reproducibility in the data loading process
    def seed_worker(worker_id):
        worker_seed = torch.initial_seed() % 2**32

    g = torch.Generator()
    g.manual_seed(seed)

    # We prepare our data for training.
    trainloader = torch.utils.data.DataLoader(trainset,
                                              batch_size=batch_size,
                                              shuffle=True,
                                              num_workers=4,
                                              worker_init_fn =
seed_worker,
                                              generator = g)
    testloader = torch.utils.data.DataLoader(testset,
                                             batch_size=batch_size,
                                             shuffle=False,
                                             num_workers=4,
                                             worker_init_fn =
seed_worker,
                                             generator = g)

    # Move model to device.
    model.to(device)

    # We define the loss function.
    criterion = nn.CrossEntropyLoss()

    # We define an optimizer, depending on the optimizer type given.
    if optimizer_type == "Original":
        optimizer = optim.SGD(model.parameters(), lr=lr,
                              momentum=momentum,
weight_decay=weight_decay)
    elif optimizer_type == "LM":
        optimizer = SGD_IKSA(model.parameters(), LM_f, lr=lr,
                             momentum=momentum, weight_decay=
weight_decay)

    # We train the model for the given number of epochs.
    for epoch in range(no_epochs):

        running_loss = 0.0
        # We go through all minibatches in the trainloader:
        for i, data in enumerate(trainloader, 0):

            # We move data to device.
            inputs = data[0].to(device)
            labels = data[1].to(device)

            # We empty the gradients of the parameters.
            optimizer.zero_grad()

            # We compute predictions.
            outputs = model(inputs)

            # We compute the loss.
            loss = criterion(outputs, labels)
            loss_list.append(loss)
            # We backpropagate.
            loss.backward()

            # If we are using landscape modification, check if we are
            # taking the running loss as c.
            if optimizer_type == "LM":
                if LM_c_run_min:
                    # Update c as the running minimum of the loss.
                    if loss < LM_c:
                      LM_c = loss.item()
                # Let the optimizer take a step.
                optimizer.step(LM_c, loss)
            else:
                optimizer.step()

            c_list.append(LM_c)
            running_loss += loss.item()

            # Compute the average loss every 20 mini-batches.
            if i % 20 == 19:
                print(epoch + 1, i + 1, running_loss / 20)
                average_loss_list.append(running_loss / 20)
                running_loss = 0.0

        # For each epoch, compute testing accuracy.
        correct = 0
        total = 0
        with torch.no_grad():
```

```python
        # Iterate through the testing data.
        for data in testloader:
            images, labels = data
            images = images.to(device)
            labels = labels.to(device)
            # Compute output.
            outputs = model(images)
            # Compute prediction, by taking the max output.
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
    # Compute accuracy.
    accuracy = 100 * correct / total
    accuracy_list.append(accuracy)
    print('Accuracy:', (100 * correct / total))


print('End of training.')

# Compute final testing accuracy:
correct = 0
total = 0
with torch.no_grad():
    # Iterate through the testing data.
    for data in testloader:
        images, labels = data
        images = images.to(device)
        labels = labels.to(device)
        # Compute output.
        outputs = model(images)
        # Compute prediction, by taking the max output.
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()
# Compute accuracy.
accuracy = 100 * correct / total
accuracy_list.append(accuracy)
print('Accuracy:', (100 * correct / total))

# Return a dictionary with all necessary information.
results_dict = {"param_dict": param_dict,
                "loss_list": loss_list,
                "c_list": c_list,
                "average_loss_list": average_loss_list,
                "accuracy": accuracy,
                "accuracy_list": accuracy_list}

return results_dict
```

Next, we define some functions that create useful reports for our analysis.

```python
def create_report_accuracy_list(list_of_results_dicts):
    """This function takes a list of dictionaries as resulted
    from the function defined above, and produces a dataframe with
    accuracy values for the two models (SGD with/without
    landscape modification).
    """
    accuracy_dict = {}
    # Iterate through all dictionaries of results:
    for d in list_of_results_dicts:
        optimizer = d["param_dict"]["optimizer_type"]
        epochs = d["param_dict"]["no_epochs"]
        lr = d["param_dict"]["lr"]
        momentum = d["param_dict"]["momentum"]
        wd = d["param_dict"]["wd"]
        f = d["param_dict"]["LM_f"]
        c = d["param_dict"]["LM_c"]
        c_running_loss = str(d["param_dict"]["LM_c_run_min"])

        # We create a unique key for each experiment.
        key =
f"{optimizer}_{epochs}_{lr}_{momentum}_{wd}_{f}_{c}_{c_running_loss}"
        # We assign the corresponding loss values to the key.
        value = d["accuracy_list"]
        # We create a dictionary that will be turned to a dataframe.
        accuracy_dict[key] = value

    accuracy_df = pd.DataFrame(accuracy_dict)

    return accuracy_df


def create_report(list_of_results_dicts):
    """This function takes a list of dictionaries as resulted
    from the function defined above, and produces a dataframe with
    the following values for the two models (SGD with/without
    landscape modification): optimizer used, epochs,
    learning rate, meomentum, weight decay, landscape
    modification parameters, minimum loss value,
    last loss value, last average loss value,
    test accuracy.
    """
    columns = ["Optimizer", "Epochs", "Learning rate",
               "Momentum", "Weight Decay", "LM_f", "LM_C",
               "C_running_min", "Min Loss", "Last Loss",
               "Last Average Loss", "Test Accuracy"]

    rows = []
    for d in list_of_results_dicts:

        optimizer = d["param_dict"]["optimizer_type"]
        epochs = d["param_dict"]["no_epochs"]
        lr = d["param_dict"]["lr"]
        momentum = d["param_dict"]["momentum"]
        wd = d["param_dict"]["wd"]
        f = d["param_dict"]["LM_f"]
        c = d["param_dict"]["LM_c"]
        c_running_loss = str(d["param_dict"]["LM_c_run_min"])
        min_loss = min(d["loss_list"]).item()
        last_loss = d["loss_list"][-1].item()
        last_average_loss = d["average_loss_list"][-1]
        test_accuracy = d["accuracy"]
        rows.append([optimizer, epochs, lr, momentum, wd,\
            f, c, c_running_loss, min_loss, last_loss,
last_average_loss, test_accuracy])


    report_df = pd.DataFrame(columns = columns, data = rows)

    return report_df

def create_loss_sheet(list_of_results_dicts):

    loss_dict = {}
    for d in list_of_results_dicts:

        optimizer = d["param_dict"]["optimizer_type"]
        epochs = d["param_dict"]["no_epochs"]
        lr = d["param_dict"]["lr"]
        momentum = d["param_dict"]["momentum"]
        wd = d["param_dict"]["wd"]
        f = d["param_dict"]["LM_f"]
        c = d["param_dict"]["LM_c"]
        c_running_loss = str(d["param_dict"]["LM_c_run_min"])

        key =
f"{optimizer}_{epochs}_{lr}_{momentum}_{wd}_{f}_{c}_{c_running_loss}"
        value = d["loss_list"]

        # turn into list of floats
        map_obj = map(torch.Tensor.item, value)
        value = list(map_obj)

        loss_dict[key] = value

    loss_df = pd.DataFrame(loss_dict)
    return loss_df

def create_c_sheet(list_of_results_dicts):

    c_dict = {}
    for d in list_of_results_dicts:

        optimizer = d["param_dict"]["optimizer_type"]
        epochs = d["param_dict"]["no_epochs"]
        lr = d["param_dict"]["lr"]
        momentum = d["param_dict"]["momentum"]
        wd = d["param_dict"]["wd"]
        f = d["param_dict"]["LM_f"]
        c = d["param_dict"]["LM_c"]
        c_running_loss = str(d["param_dict"]["LM_c_run_min"])

        key =
f"{optimizer}_{epochs}_{lr}_{momentum}_{wd}_{f}_{c}_{c_running_loss}"
        value = d["c_list"]

        c_dict[key] = value

    c_df = pd.DataFrame(c_dict)

    return c_df


def create_average_loss_sheet(list_of_results_dicts):

    average_loss_dict = {}
    for d in list_of_results_dicts:

        optimizer = d["param_dict"]["optimizer_type"]
        epochs = d["param_dict"]["no_epochs"]
        lr = d["param_dict"]["lr"]
        momentum = d["param_dict"]["momentum"]
        wd = d["param_dict"]["wd"]
        f = d["param_dict"]["LM_f"]
        c = d["param_dict"]["LM_c"]
        c_running_loss = str(d["param_dict"]["LM_c_run_min"])

        key =
f"{optimizer}_{epochs}_{lr}_{momentum}_{wd}_{f}_{c}_{c_running_loss}"
        value = d["average_loss_list"]

        average_loss_dict[key] = value

    average_loss_df = pd.DataFrame(average_loss_dict)
    return average_loss_df
```

Next, we load our CIFAR10 data, applying transformations

as per https://github.com/timgaripov/swa.

```
seed = 0
setup_seed(seed)

transform_train = transforms.Compose([
    transforms.RandomCrop(32, padding=4),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994,
0.2010)),
])
transform_test = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994,
0.2010)),
])

batch_size = 128


trainset = torchvision.datasets.CIFAR10(root='./data',
                                        train=True,
                                        download=True,
                                        transform=transform_train)
testset = torchvision.datasets.CIFAR10(root='./data',
                                       train=False,
                                       download=True,
                                       transform=transform_test)
```

```
Files already downloaded and verified
Files already downloaded and verified
```

We define the parameters:

```
# Use a GPU if available.
device = torch.device("cuda:0" if torch.cuda.is_available() else
"cpu")
print(device)

# SGD Parameters
lr = 0.1
momentum = 0.9
wd = 0

# LM parameters
def x(x):
    return x

c = 10**5

# seeds
seed_list = [0, 10, 100]

# number of epochs
epochs = 150

# model
model = PreResNet()

cuda:0
```

We run the experiments and keep the results in a list:

```
final_results = []

for seed in seed_list:
    results_dict_SGD =  compare_SGD_SGD_LM(device, seed,
                                           model, trainset, testset,
                                           batch_size, epochs,
                                           lr, momentum, wd,
                                           "Original", x, c, True)
    final_results.append(results_dict_SGD)
    results_dict_SGD_LM = compare_SGD_SGD_LM(device, seed,
                                             model, trainset, testset,

                                             batch_size, epochs,
                                             lr, momentum, wd,
                                             "LM", x, c, True)
    final_results.append(results_dict_SGD_LM)
```

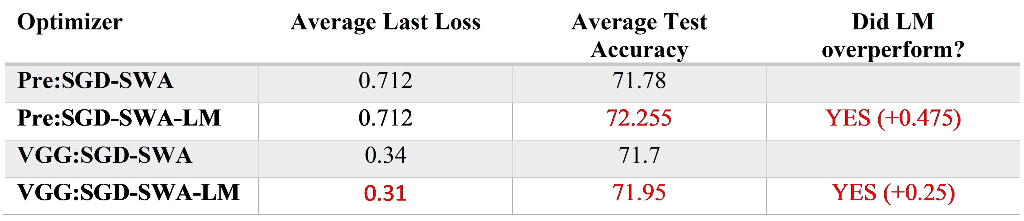
Finally, we save and download relevant information:

```
report_df = create_report(final_results)
report_df.to_csv("report.csv")
files.download("report.csv")
loss_df = create_loss_sheet(final_results)
loss_df.to_csv("report_loss.csv")
files.download("report_loss.csv")
avg_loss_df = create_average_loss_sheet(final_results)
avg_loss_df.to_csv("report_avg_loss.csv")
files.download("report_avg_loss.csv")
c_list = create_c_sheet(final_results)
c_list.to_csv("report_c.csv")
files.download("report_c.csv")
accuracy_list = create_report_accuracy_list(final_results)
accuracy_list.to_csv("report_accuracy.csv")
files.download("report_accuracy.csv")
```

Appendix C

Additional Results

In order to obtain a the testing accuracy on CIFAR100 closer to the state-of-art results, we use stochastic weighted averaging (SWA), as presented in [12]. We report the performance enhancement (in Fig. C.1) of landscape modification on VGG and PreResNet enhanced with SWA (SGD is used). Implementation is inspired by the same work.



Optimizer	Average Last Loss	Average Test Accuracy	Did LM overperform?
Pre:SGD-SWA	0.712	71.78	
Pre:SGD-SWA-LM	0.712	72.255	YES (+0.475)
VGG:SGD-SWA	0.34	71.7	
VGG:SGD-SWA-LM	**0.31**	71.95	YES (+0.25)

FIGURE C.1: Results of Landscape Modification for PreResNet and VGG on CIFAR100, using SWA.