Non-Convex Optimization and Algorithms for Machine Learning

Roshan Avik Srivastava

July 2025

Abstract

The paper is a study of optimization algorithms for convex and non-convex optimization in the context of machine learning with a detailed proof of Stochastic Gradient Descent (SGD), the Polyak-Łojasiewicz (PL) condition, and an analysis of methods to escape saddle-points efficiently. It contrasts different improvements of the Stochastic Gradient Descent method and discusses improvements expected in the field.

1 Introduction

Convex and Non-convex optimization are crucial in machine learning because they allow for the modeling of complex relationships within data, especially in deep learning and other non-linear models. While finding the global minimum of a non-convex function is generally NP-hard, many practical optimization algorithms, like stochastic gradient descent, often find solutions that are close to optimal or perform well in practice. Research in non-convex optimization has led to the development of powerful algorithms like adaptive learning rates, second-order methods, and specialized techniques for deep learning. In essence, non-convex optimization provides the flexibility to model complex data and constraints, enabling the development of powerful machine learning models, even though the optimization process can be more challenging.

1.1 The Loss/ Cost Function

In machine learning, the loss or cost function $L(x)$ refers to the error or discrepancy between a machine learning or deep learning model's predicted output and the actual or target value. Loss quantifies how far off the model's predictions are from the true results, and a lower loss value generally indicates better model performance. At a high level, we can categorize Loss functions as Regression and Classification Loss functions depending on the model they come from. Below are some common types of Loss Functions in Machine Learning.
Minimizing the loss function is the key to improving the accuracy of the model.

$$\min_x f(x), \quad (f : \mathbb{R}^n \to \mathbb{R})$$

Table 1: Common types of loss functions and their use cases

Loss Function	Description
Mean Squared Error (MSE)	Calculates the average squared difference between predicted and actual values. Used in regression.
Cross-Entropy Loss	Measures the difference between predicted and actual probability distributions. Used in classification.
Binary Cross-Entropy	A special case of cross-entropy for binary classification tasks.
Categorical Cross-Entropy	Applied to multi-class classification where each sample belongs to one of many classes.
Hinge Loss	Commonly used with Support Vector Machines (SVMs).
Huber Loss	Combines MSE and MAE to be robust to outliers.

When we define a loss function, our goal is to minimize the loss.

Lower loss indicates better model performance. Ideally, we want the model to reach the global minimum, the lowest possible loss. However, most loss functions used in deep learning—such as focal loss or binary cross-entropy loss—are non-convex. This means the optimization process might get stuck in a local minimum, a point that appears optimal in a small region but isn't the best overall.

1.2 Properties of the Loss Function to consider

These are some properties of the loss function that significantly impact how we handle the loss function and the convergence and stability of the algorithm we choose to minimize the function.

1.3 Convexity

Definition. A function $f : \mathbb{R}^n \to \mathbb{R}$ is called *convex* if for every pair of vectors $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ and every $\lambda \in [0, 1]$, we have

$$f(\lambda\mathbf{x} + (1 - \lambda)\mathbf{y}) \leq \lambda f(\mathbf{x}) + (1 - \lambda)f(\mathbf{y}).$$

Strictly convex, if $\forall x, y, x \neq y, \forall\lambda \in (0, 1)$

$$f\left(\lambda x + (1 - \lambda)y\right) < \lambda f(x) + (1 - \lambda)f(y).$$

Geometrically, this means that the line segment connecting $(x, f(x))$ to $(y, f(y))$ sits above or on the graph of f, and not below it.

Definition. A function $f : \mathbb{R}^n \to \mathbb{R}$ is called *concave* if for every pair of vectors $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ and every $\lambda \in [0, 1]$, we have

$$f(\lambda\mathbf{x} + (1 - \lambda)\mathbf{y}) \geq \lambda f(\mathbf{x}) + (1 - \lambda)f(\mathbf{y}).$$

Strictly concave, if $\forall x, y, x \neq y, \forall\lambda \in (0, 1)$

$$f\left(\lambda x + (1 - \lambda)y\right) > \lambda f(x) + (1 - \lambda)f(y).$$

Geometrically, this means that the line segment connecting $(x, f(x))$ to $(y, f(y))$ goes below the graph of f, and not above it.

Note:

- f is concave if and only if $-f$ is convex. Similarly, f is strictly concave if and only if $-f$ is strictly convex.
- A function can be both *convex* and *concave* at the same time. A straight line, for example, is both *convex* and *concave*.

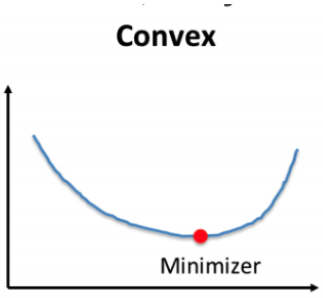

Figure 1: Convex function.

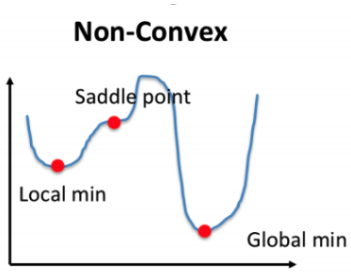

Figure 2: Non-Convex function.

1.3.1 Convexity and Local and Global Minima

Definition 1 A vector $\mathbf{x}$ is a *local minimum* of f if $f(\mathbf{x}) \leq f(\mathbf{y})$ for all $\mathbf{y}$ in a neighborhood of $\mathbf{x}$.
Definition 2 A vector $\mathbf{x}$ is a *global minimum* of f if $f(\mathbf{x}) \leq f(\mathbf{y})$ for all $\mathbf{y}$.

If f is convex, any local minimum is also a global one. This special property of convex functions helps to design powerful optimisation algorithms.

Convexity can simplify the optimization landscape. Convex functions are particularly important because they have a unique global minimum. This means that if we want to optimize a convex function, we can be sure that we will always find the best solution by searching for the minimum value of the function. This makes optimization easier and more reliable. For gradient descent, convexity ensures that any local minimum is also a global minimum, making optimization easier. If a function is not convex, gradient descent might get stuck in local minima instead of finding the global minimum.

1.3.2 More About Strict and Strong Convexity

The function f is called *strictly convex* if and only if for all real $0 < t < 1$ and all $x_1, x_2 \in X$ such that $x_1 \neq x_2$:

$$f\left(tx_1 + (1-t)x_2\right) < tf(x_1) + (1-t)f(x_2)$$

A strictly convex function f is a function such that the straight line between any pair of points on the curve f is above the curve f except for the intersection

points between the straight line and the curve. An example of a function which is convex but not strictly convex is

$$f(x, y) = x^2 + y.$$

This function is not strictly convex because any two points sharing an x coordinate will have a straight line between them, while any two points *not* sharing an x coordinate will have a greater value of the function than the points between them.

The concept of *strong* convexity extends and parametrizes the notion of strict convexity. Intuitively, a strongly convex function is a function that grows as fast as a quadratic function. A strongly convex function is also strictly convex, but not vice versa. If a one-dimensional function f is twice continuously differentiable and the domain is the real line, then we can characterize it as follows:

- f is convex if and only if $f''(x) \geq 0$ for all x.
- f is strictly convex if $f''(x) > 0$ for all x (note: this is sufficient, but not necessary).
- f is strongly convex if and only if $f''(x) \geq m > 0$ for all x.

For example, let f be strictly convex, and suppose there is a sequence of points (x_n) such that $f''(x_n) = \frac{1}{n}$. Even though $f''(x_n) > 0$, the function is not strongly convex because $f''(x)$ can become arbitrarily small.
More generally, a differentiable function f is called *strongly convex with parameter* $m > 0$ if the following inequality holds for all points x, y in its domain:

$$\left(\nabla f(x) - \nabla f(y)\right)^T (x - y) \geq m\|x - y\|_2^2,$$

or, more generally,

$$\langle \nabla f(x) - \nabla f(y), x - y \rangle \geq m\|x - y\|^2,$$

where $\langle \cdot, \cdot \rangle$ is any inner product, and $\| \cdot \|$ is the corresponding norm.
An equivalent condition is the following:

$$f(y) \geq f(x) + \nabla f(x)^T (y - x) + \frac{m}{2}\|y - x\|_2^2.$$

This definition approaches the definition for strict convexity as $m \to 0$, and is identical to the definition of a convex function when $m = 0$. Despite this, functions exist that are strictly convex but are not strongly convex for any $m > 0$.
If the function f is twice continuously differentiable, then it is strongly convex with parameter m if and only if

$$\nabla^2 f(x) \succeq mI$$

for all x in the domain, where I is the identity matrix and $\nabla^2 f$ is the Hessian matrix, and $\succeq$ means that $\nabla^2 f(x) - mI$ is positive semi-definite. This is equivalent to requiring that the minimum eigenvalue of $\nabla^2 f(x)$ be at least m for all x.
If the domain is just the real line, then $\nabla^2 f(x)$ is simply the second derivative $f''(x)$, so the condition becomes $f''(x) \geq m$. If $m = 0$, then this means the Hessian is positive semidefinite, which implies the function is convex (and perhaps strictly convex), but not strongly convex.

Assuming still that the function is twice continuously differentiable, one can show that the lower bound of $\nabla^2 f(x)$ implies that it is strongly convex. Using Taylor's theorem, there exists

$$z \in \{tx + (1-t)y : t \in [0,1]\}$$

such that

$$f(y) = f(x) + \nabla f(x)^T (y - x) + \frac{1}{2}(y - x)^T \nabla^2 f(z)(y - x).$$

Then

$$(y - x)^T \nabla^2 f(z)(y - x) \geq m(y - x)^T (y - x)$$

by the assumption about the eigenvalues, and hence we recover the second strong convexity inequality above.

A function f is strongly convex with parameter m if and only if the function

$$x \mapsto f(x) - \frac{m}{2}\|x\|^2$$

is convex.

A twice continuously differentiable function f on a compact domain X that satisfies $f''(x) > 0$ for all $x \in X$ is strongly convex. The proof follows from the extreme value theorem, which states that a continuous function on a compact set has a maximum and minimum.

Strongly convex functions are generally easier to work with than convex or strictly convex functions, since they are a smaller class. Like strictly convex functions, strongly convex functions have unique minima on compact sets. It is one of the factors that influences the choice of algorithm used to minimize the fucntion.

1.4 L-smoothness

A function is L-smooth if its gradient is Lipschitz continuous with a constant L. This means the gradient doesn't change too quickly, preventing the algorithm from taking overly large steps that might overshoot the minimum. Specifically, it implies that:

$$\|\nabla f(x) - \nabla f(y)\| \leq L\|x - y\|.$$

L-smoothness ensures a bounded gradient and helps in selecting an appropriate step size for gradient descent, ensuring that the algorithm can take appropriate steps and converge towards the solution.

2 Gradient Descent Methods

To solve our minimization problem, we look at gradient descent methods. These are iterative algorithms. The general form of the iterations is usually as follows.

$$x_{k+1} = x_k + \alpha_k d_k$$

Where

$$\begin{aligned} k \in \mathbb{Z}_+ &: \text{ index of time (iteration number)} \\ x_k \in \mathbb{R}^n &: \text{ Current point} \\ d_k \in \mathbb{R}^n &: \text{ Direction to move along at iteration } k \\ x_{k+1} \in \mathbb{R}^n &: \text{ Next point} \\ \alpha_k \in \mathbb{R}_+ &: \text{ Step size at iteration } k \end{aligned}$$

The goal of these algorithms is to make the sequence $\{f(x_k)\}$ decrease as much as possible. So we start with the questions:

- How to choose d_k?
- How to choose α_k?

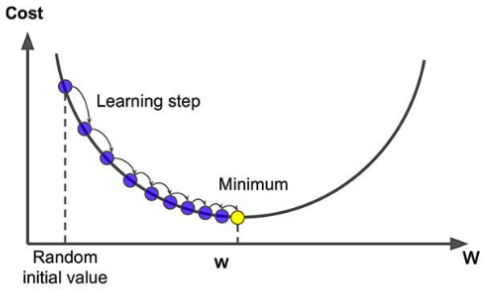

Figure 3: Gradient Descent.

2.1 Choosing the Descent Direction

Choosing the direction of descent in an optimization algorithm is an important first step. The following two Lemmas help us think about the Descent direction.

Lemma 1: Consider yourself sitting at a point $x \in \mathbb{R}^n$ and looking (locally) at the value of the function f in all directions around you. The direction with the maximum rate of decrease is along $-\nabla f(x)$.

When we speak of direction, the magnitude of the vector does not matter; e.g., $\nabla f(x), 5\nabla f(x), \dfrac{\nabla f(x)}{20}, \dfrac{\nabla f(x)}{\|\nabla f(x)\|}$ all are in the same direction.

For a given point $x \in \mathbb{R}^n$, a direction $d \in \mathbb{R}^n$ is called a descent direction, If there exists $\bar{\alpha} > 0$ ($\alpha \in \mathbb{R}$) such that

$$f(x + \alpha d) < f(x), \quad \forall \alpha \in (0, \bar{\alpha}).$$

There is a small enough (but nonzero) amount that you can move in direction d and be guaranteed to decrease the function value.

Proof of Lemma 1.
Consider a point x, a direction d, and the univariate function

$$g(\alpha) = f\left(x + \alpha \frac{d}{\|d\|}\right).$$

The rate of change of f at x in direction d is given by $g'(0)$, which by the chain rule equals

$$\frac{1}{\|d\|}\langle \nabla f(x), d\rangle.$$

By the Cauchy–Schwarz inequality (see, e.g., Theorem 2.3 of [CZ13] for a proof), we have:

$$-\frac{1}{\|d\|} \cdot \|\nabla f(x)\| \cdot \|d\| \leq \frac{1}{\|d\|}\langle \nabla f(x), d\rangle \leq \frac{1}{\|d\|} \cdot \|\nabla f(x)\| \cdot \|d\|,$$

which, after simplifying, gives

$$-\|\nabla f(x)\| \leq \frac{1}{\|d\|}\langle \nabla f(x), d\rangle \leq \|\nabla f(x)\|.$$

So the rate of change in any direction cannot be larger than $\|\nabla f(x)\|$, or smaller than $-\|\nabla f(x)\|$. However, if we take $d = \nabla f(x)$, the right inequality is achieved:

$$\frac{1}{\|\nabla f(x)\|}\langle \nabla f(x), \nabla f(x)\rangle = \frac{1}{\|\nabla f(x)\|} \cdot \|\nabla f(x)\|^2 = \|\nabla f(x)\|.$$

Similarly, if we take $d = -\nabla f(x)$, then the left inequality is achieved.

Lemma 2: Consider a point $x \in \mathbb{R}^n$. Any direction d satisfying

$$\langle d, \nabla f(x) \rangle < 0$$

is a descent direction. (In particular, $-\nabla f(x)$ is a descent direction.)

Proof of Lemma 2.
By Taylor's theorem, we have

$$f(x + \alpha d) = f(x) + \alpha \nabla f(x)^T d + o(\alpha).$$

Since

$$\lim_{\alpha \to 0} \frac{|o(\alpha)|}{\alpha} = 0,$$

there exists $\bar{\alpha} > 0$ such that

$$\frac{|o(\alpha)|}{\alpha} < |\nabla f(x)^T d|, \quad \forall \alpha \in (0, \bar{\alpha}).$$

This, together with our assumption that $\nabla f(x)^T d < 0$, implies that $\forall \alpha \in (0, \bar{\alpha})$, we must have:

$$f(x + \alpha d) - f(x) < 0.$$

Hence, d is a descent direction.

2.2 Choosing the Step Size

Another important thing to do is to choose step size. Gradient descent is based on the observation that if the multi-variable function $f(\mathbf{x})$ is defined and differentiable in a neighborhood of a point $\mathbf{a}$, then $f(\mathbf{x})$ decreases *fastest* if one goes from $\mathbf{a}$ in the direction of the negative gradient of f at $\mathbf{a}$, $-\nabla f(\mathbf{a})$. It follows that, if

$$\mathbf{a}_{n+1} = \mathbf{a}_n - \eta \nabla f(\mathbf{a}_n)$$

for a small enough step size or learning rate $\eta \in \mathbb{R}_+$, then $f(\mathbf{a}_n) \geq f(\mathbf{a}_{n+1})$. In other words, the term $\eta \nabla f(\mathbf{a})$ is subtracted from $\mathbf{a}$ because we want to move against the gradient, toward the local minimum. With this observation in mind, one starts with a guess $\mathbf{x}_0$ for a local minimum of f, and considers the sequence $\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2, \ldots$ such that

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \eta_n \nabla f(\mathbf{x}_n), \quad n \geq 0.$$

We have a *monotonic* sequence

$$f(\mathbf{x}_0) \geq f(\mathbf{x}_1) \geq f(\mathbf{x}_2) \geq \cdots,$$

so the sequence $(\mathbf{x}_n)$ converges to the desired local minimum. Note that the value of the *step size* η is allowed to change at every iteration.

It is possible to guarantee the *convergence* to a local minimum under certain assumptions on the function f (for example, f convex and ∇f Lipschitz) and particular choices of η. Those include the sequence

$$\eta_n = \frac{\left|(\mathbf{x}_n - \mathbf{x}_{n-1})^\top [\nabla f(\mathbf{x}_n) - \nabla f(\mathbf{x}_{n-1})]\right|}{\|\nabla f(\mathbf{x}_n) - \nabla f(\mathbf{x}_{n-1})\|^2}$$

as in the *Barzilai–Borwein method*, or a sequence η_n satisfying the *Wolfe conditions* (which can be found by using *line search*). When the function f is *convex*, all local minima are also global minima, so in this case, gradient descent can converge to the global solution.

3 The Gradient Descent Algorithm

Gradient descent is a method for unconstrained mathematical optimization. It is a first-order iterative algorithm for minimizing a differentiable multivariate function.

The idea is to take repeated steps in the opposite direction of the gradient (or approximate gradient) of the function at the current point, because this is the direction of steepest descent. Conversely, stepping in the direction of the gradient will lead to a trajectory that maximizes that function; the procedure is then known as gradient ascent. It is particularly useful in machine learning for minimizing the cost or loss function.

Gradient descent is generally attributed to Augustin-Louis Cauchy, who first suggested it in 1847.

Gradient descent is based on the observation that if the multi-variable function $f(\mathbf{x})$ is defined and differentiable in a neighborhood of a point $\mathbf{a}$, then $f(\mathbf{x})$ decreases *fastest* if one goes from $\mathbf{a}$ in the direction of the negative gradient of f at $\mathbf{a}$, $-\nabla f(\mathbf{a})$. It follows that, if

$$\mathbf{a}_{n+1} = \mathbf{a}_n - \eta \nabla f(\mathbf{a}_n)$$

for a small enough step size or learning rate $\eta \in \mathbb{R}_+$, then $f(\mathbf{a}_n) \geq f(\mathbf{a}_{n+1})$. In other words, the term $\eta \nabla f(\mathbf{a})$ is subtracted from $\mathbf{a}$ because we want to move against the gradient, toward the local minimum. With this observation in mind, one starts with a guess $\mathbf{x}_0$ for a local minimum of f, and considers the sequence $\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2, \ldots$ such that

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \eta_n \nabla f(\mathbf{x}_n), \quad n \geq 0.$$

We have a *monotonic* sequence

$$f(\mathbf{x}_0) \geq f(\mathbf{x}_1) \geq f(\mathbf{x}_2) \geq \cdots,$$

so the sequence $(\mathbf{x}_n)$ converges to the desired local minimum. Note that the value of the *step size* η is allowed to change at every iteration.

It is possible to guarantee the *convergence* to a local minimum under certain assumptions on the function f (for example, f convex and ∇f Lipschitz) and particular choices of η. Those include the sequence

$$\eta_n = \frac{\left|(\mathbf{x}_n - \mathbf{x}_{n-1})^\top [\nabla f(\mathbf{x}_n) - \nabla f(\mathbf{x}_{n-1})]\right|}{\|\nabla f(\mathbf{x}_n) - \nabla f(\mathbf{x}_{n-1})\|^2}$$

as in the *Barzilai–Borwein method*, or a sequence η_n satisfying the *Wolfe conditions* (which can be found by using *line search*). When the function f is *convex*, all local minima are also global minima, so in this case gradient descent can converge to the global solution.

In many ML problems, the objective function is not convex, which means there are multiple local minima. Traditional gradient descent algorithms may get stuck in one of the local minima, resulting in suboptimal models. SGD, on the other hand, is less likely to get stuck because it updates the parameters using only a few data points at a time, making it more likely to find the global minimum.

4 The Stochastic Gradient Descent Algorithm

Stochastic Gradient Descent (SGD) is an effective and popular optimization algorithm for machine learning. Its key strength is its ability to process large datasets and reach convergence quickly. It also has high computational efficiency due to the fact that the gradient can be estimated with a random sample of data points instead of requiring the full dataset.

Unlike traditional Gradient Descent, SGD relies on a single example or samples at each iteration, introducing randomness into the learning process. This makes your model learn faster as it converges quicker than other optimization methods such as Mini-Batch or Batch Gradient Descent. When applied to models with thousands of features, SGD performs extremely well because of its significant computational speed and accuracy.

4.1 Other conditions for successful Gradient Descent

Unbiased Gradient

In batch gradient descent, the gradient is calculated over the entire dataset, making it an unbiased estimate of the true gradient. However, for large datasets, this can be computationally expensive. Stochastic gradient descent (SGD) uses a mini-batch of data points to estimate the gradient, introducing some bias. The bias in SGD updates can lead to faster convergence on large datasets, but it also introduces variance.

Bounded Variance

In the context of SGD, a bounded variance refers to the fact that the variance of the stochastic gradient estimate should not be too large. If the variance is too high, the updates can be erratic, making it difficult for the algorithm to converge to the minimum. A good estimate of the variance helps in selecting a suitable step size for SGD and can lead to faster convergence.

Need for Stochastic Gradient Descent (SGD) in Machine Learning

There was a need for SGD (Stochastic Gradient Descent) in Machine Learning (ML) because of the following reasons:

1. **Large Datasets:** In ML, it is common to have very large datasets with millions or even billions of data points. Using traditional gradient descent algorithms on such large datasets is computationally expensive and impractical. In such cases, SGD works well because it uses randomly selected data points to update the model parameters, making the process faster and more efficient.

2. **Non-Convex Optimization Problems:** In many ML problems, the objective function is not convex, which means there are multiple local minima. Traditional gradient descent algorithms may get stuck in one of the local minima, resulting in suboptimal models. SGD, on the other hand, is less likely to get stuck because it updates the parameters using only a few data points at a time, making it more likely to find the global minimum.

3. **Online Learning:** In some ML applications, new data is constantly being generated and added to the dataset. In such cases, it is necessary to update the model parameters in real-time, as new data becomes available. SGD is well-suited for such tasks because it updates the model parameters using small batches of data and can be applied to data as it arrives.

Overall, SGD is a more efficient and practical alternative to traditional gradient descent algorithms in many ML applications, particularly for large datasets, non-convex optimization problems, and online learning.

4.2 The Math Behind SGD

SGD works by iteratively updating the model parameters in the direction of the negative gradient of the loss function. The gradient of the loss function is a vector that points in the direction of the steepest ascent of the loss function. By moving in the direction of the negative gradient, SGD is able to find the minimum of the loss function.

SGD works by iteratively updating the model parameters in the direction of the negative gradient of the loss function. The gradient of the loss function is a vector that points in the direction of the steepest ascent of the loss function. By moving in the direction of the negative gradient, SGD is able to find the minimum of the loss function.

The loss function is a measure of how well the model fits the data. The goal of SGD is to find the set of model parameters that minimizes the loss function.

The gradient of the loss function can be calculated using the following formula:

$$\text{gradient} = \frac{\partial \mathcal{L}}{\partial \theta}$$

where L is the loss function and θ is the vector of model parameters.

The gradient can be used to update the model parameters using the following formula:

$$\theta = \theta - \eta \cdot \text{gradient}$$

where η is the learning rate, which is a hyperparameter that controls how much the model parameters are updated in each iteration.

4.3 Convex Case: Convergence to minimum

Here's a mathematical proof of convergence for Stochastic Gradient Descent (SGD) under some standard assumptions.
This first proof focuses on the expected convergence of SGD when minimizing a smooth and convex function.

We want to minimize:

$$f(x) = \mathbb{E}[f(x)]$$

with update rule:

$$x_{k+1} = x_k - \eta_k \nabla f(x_k)$$

Assumptions

- Convexity: $f(y) \geq f(x) + \nabla f(x)^T (y - x)$ for all x, y
- L-smoothness: $\|\nabla f(x) - \nabla f(y)\| \leq L\|x - y\|$
- Unbiased gradient: $\mathbb{E}[\nabla f(x_k)] = \nabla f(x_k)$
- Bounded variance: $\mathbb{E}[\|\nabla f(x_k) - \nabla f(x_k)\|^2] \leq \sigma^2$

Proof

$$\begin{aligned}\|x_{k+1} - x^*\|^2 &= \|x_k - \eta_k \nabla f(x_k) - x^*\|^2 \\ &= \|x_k - x^*\|^2 - 2\eta_k \nabla f(x_k)^T (x_k - x^*) + \eta_k^2 \|\nabla f(x_k)\|^2\end{aligned}$$

Taking expectation:

$$\mathbb{E}[\|x_{k+1} - x^*\|^2] \leq \mathbb{E}[\|x_k - x^*\|^2] - 2\eta_k \mathbb{E}[f(x_k) - f(x^*)] + \eta_k^2 C$$

Summing over k and using Jensen's inequality:

$$\mathbb{E}[f(\bar{x}_T)] - f(x^*) \leq \frac{\|x_1 - x^*\|^2}{2\eta T} + \frac{\eta C}{2}$$

If the function is strongly convex, SGD can achieve O(1/T) convergence with decaying step sizes. If the function is non-convex, different convergence results apply (e.g., convergence to a stationary point in expectation).

4.4 Non-Convex Case: Convergence to Stationary Point

Here we drop convexity and have the following assumptions -

Assumptions

- L-smoothness
- Unbiased gradient
- Bounded variance: σ^2
- Lower bounded function: $f(x) \geq f^*$

Proof

Using smoothness:

$$f(x_{k+1}) \leq f(x_k) - \eta \nabla f(x_k)^T \nabla f(x_k) + \frac{L\eta^2}{2} \|\nabla f(x_k)\|^2$$

Taking expectation and bounding variance:

$$\mathbb{E}[f(x_{k+1})] \leq \mathbb{E}[f(x_k)] - \eta \left(1 - \frac{L\eta}{2}\right) \mathbb{E}[\|\nabla f(x_k)\|^2] + \frac{L\eta^2}{2}\sigma^2$$

Summing over k:

$$\frac{1}{T} \sum_{k=0}^{T-1} \mathbb{E}[\|\nabla f(x_k)\|^2] \leq \frac{f(x_0) - f^*}{\eta \rho T} + \frac{L\eta\sigma^2}{2\rho}$$

Choosing $\eta = \mathcal{O}(1/\sqrt{T})$ gives:

$$\mathbb{E}[\|\nabla f(x_k)\|^2] = \mathcal{O}(1/\sqrt{T})$$

So SGD finds approximate stationary points in expectation for non-convex smooth functions.

5 Escaping Saddle Points

To effectively escape saddle points during optimization, especially in machine learning, techniques like adding noise (stochastic gradient perturbation) or adjusting the learning rate schedule are crucial. These methods help the optimization process move away from flat regions (saddle points) where gradient descent can get stuck.

1. Stochastic Gradient Perturbation

Concept: Introduce small random noise to the gradient during optimization, especially when the gradient's magnitude is small (indicating proximity to a saddle point).
How it helps: This noise introduces a random component that can push the optimization process out of the flat region of the saddle point, allowing it to explore other directions and potentially find a better solution.
Implementation: Common approaches involve adding a small random vector to the gradient or using a noise distribution with a specific variance.

2. Learning Rate Annealing / Scheduling

Concept: Gradually reduce the learning rate (step size) during training.
How it helps: A higher initial learning rate allows for faster initial progress. As training progresses, a smaller learning rate enables more precise exploration of the loss landscape, potentially escaping saddle points that might trap a larger step size.

Common methods:

- **Step Decay:** Reduce the learning rate by a factor (e.g., halving) every few epochs.
- **Cosine Annealing:** Adjust the learning rate according to a cosine function over time.

3. Adaptive Learning Rates

Concept: Adaptive methods like Adam, Adagrad, and RMSProp automatically adjust the learning rate for each parameter based on past gradients.
How it helps: Adaptive methods can be more effective at escaping saddle points because they dynamically adjust the learning rate based on the local curvature of the loss function.
Empirical evidence: Adaptive methods have been shown to escape saddle points faster and converge faster to second-order stationary points compared to vanilla SGD, according to research published in the *Proceedings of Machine Learning Research*.

4. Other Strategies

Second-order information: Some methods leverage second-order information (Hessian matrix) to identify saddle points and find directions of negative curvature to escape.
Perturbed Gradient Descent: This approach combines gradient descent with random perturbations, ensuring that the algorithm avoids saddle points with high probability, as discussed in an *arXiv* article.
Escaping saddle points requires careful consideration of the optimization process. Adding noise to the gradient, employing learning rate schedules, or using adaptive methods can significantly improve the chances of finding better solutions in non-convex optimization problems by avoiding or escaping saddle points.

5.1 SGD with PL Condition: Linear Convergence

Let's now assume that f satisfies the PL condition, which says:

PL Condition

$$\frac{1}{2}\|\nabla f(x)\|^2 \geq \mu(f(x) - f^*)$$

This is weaker than strong convexity, yet it still gives linear convergence.

Proof

Using smoothness and variance bounds:

$$\mathbb{E}[f(x_{k+1})] \leq \mathbb{E}[f(x_k)] - \eta(1 - \frac{L\eta}{2})\mathbb{E}[\|\nabla f(x_k)\|^2] + \frac{L\eta^2\sigma^2}{2}$$

Apply PL condition:

$$\mathbb{E}[f(x_{k+1})] - f^* \leq (1 - \eta\mu)\mathbb{E}[f(x_k) - f^*] + \frac{\eta\sigma^2}{2\mu}$$

Recursively:

$$\mathbb{E}[f(x_k)] - f^* \leq (1 - \eta\mu)^k(f(x_0) - f^*) + \frac{\eta\sigma^2}{2\mu}$$

5.2 Momentum

SGD has trouble navigating ravines, i.e. areas where the surface curves much more steeply in one dimension than in another **ref4**, which are common around local optima. In these scenarios, SGD oscillates across the slopes of the ravine while only making hesitant progress along the bottom towards the local optimum, as in Image 2.

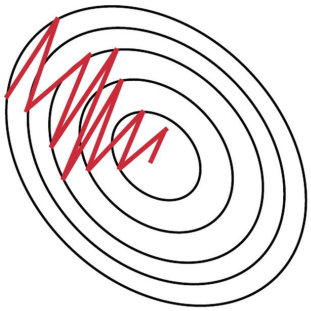

Image 2: SGD without momentum

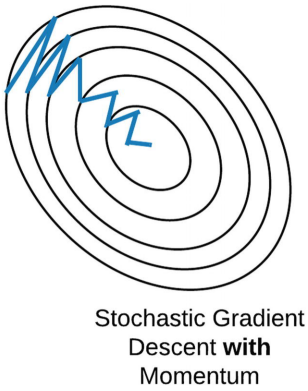

Image 3: SGD with momentum

Momentum is a method that helps accelerate SGD in the relevant direction and dampens oscillations, as can be seen in Image 3. It does this by adding a fraction γ of the update vector of the past time step to the current update vector:

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$$
$$\theta = \theta - v_t$$

Note: Some implementations exchange the signs in the equations. The momentum term γ is usually set to 0.9 or a similar value.

6 Variations of SGD with adaptive step sizes

6.1 Setup to evaluate generic adaptive methods

We now provide a framework of adaptive methods that gives us insights into the differences between different adaptive methods and is useful for understanding the flaws in a few popular adaptive methods. Algorithm **??** provides a generic adaptive framework that encapsulates many popular adaptive methods. Note the algorithm is still abstract because the "averaging" functions ϕ_t and ψ_t have not been specified.

Generic Adaptive Method Setup
Input: $x_1 \in \mathcal{F}$, step size $\{\alpha_t > 0\}_{t=1}^T$, sequence of functions $\{\phi_t, \psi_t\}_{t=1}^T$

FOR $t = 1$ to T
$g_t = \nabla f_t(x_t)$
$m_t = \phi_t(g_1, \ldots, g_t)$ and $V_t = \psi_t(g_1, \ldots, g_t)$
$\hat{x}_{t+1} = x_t - \alpha_t m_t / \sqrt{V_t}$
$x_{t+1} = \Pi_{\mathcal{F}, \sqrt{V_t}}(\hat{x}_{t+1})$
ENDFOR

Here $\phi_t : \mathcal{F}^t \to \mathbb{R}^d$ and $\psi_t : \mathcal{F}^t \to \mathcal{S}_+^d$. For ease of exposition, we refer to α_t as step size and $\alpha_t V_t^{-1/2}$ as learning rate of the algorithm and furthermore, restrict ourselves to diagonal variants of adaptive methods encapsulated by the Algorithm where $V_t = \mathrm{diag}(v_t)$.

We first observe that standard stochastic gradient algorithm falls in this framework by using:

$$\phi_t(g_1, \ldots, g_t) = g_t \quad \text{and} \quad \psi_t(g_1, \ldots, g_t) = \mathbb{I}, \tag{SGD}$$

and $\alpha_t = \alpha/\sqrt{t}$ for all $t \in [T]$. While the decreasing step size is required for convergence, such an aggressive decay of the learning rate typically translates into poor empirical performance. The key idea of adaptive methods is to choose averaging functions appropriately so as to entail good convergence.
For instance, the first adaptive method ADAGRAD (Duchi et al., 2011), which propelled the research on adaptive methods, uses the following averaging functions:

$$\phi_t(g_1, \ldots, g_t) = g_t \quad \text{and} \quad \psi_t(g_1, \ldots, g_t) = \frac{\mathrm{diag}(\sum_{i=1}^t g_i^2)}{t}, \tag{ADAGRAD}$$

and step size $\alpha_t = \alpha/\sqrt{t}$ for all $t \in [T]$. In contrast to a learning rate of $\alpha/\sqrt{t}$ in SGD, such a setting effectively implies a modest learning rate decay of $\alpha/\sqrt{\sum_{i=1}^t g_{i,j}^2}$ for $j \in [d]$. When the gradients are sparse, this can potentially lead to huge gains in terms of convergence (see Duchi et al. (2011)). These gains have also been observed in practice for even non-sparse settings.

Adaptive methods based on Exponential Moving Averages

Exponential moving average variants of ADAGRAD are popular in the deep learning community. RMSPROP, ADAM, NADAM, and ADADELTA are some prominent algorithms that fall in this category. The key difference is to use an exponential moving average as function ψ_t instead of the simple average function used in ADAGRAD. ADAM, a particularly popular variant, uses the following averaging functions:

$$\phi_t(g_1, \ldots, g_t) = (1-\beta_1)\sum_{i=1}^t \beta_1^{t-i} g_i \quad \text{and} \quad \psi_t(g_1, \ldots, g_t) = (1-\beta_2)\mathrm{diag}\left(\sum_{i=1}^t \beta_2^{t-i} g_i^2\right), \tag{ADAM}$$

for some $\beta_1, \beta_2 \in [0, 1)$. This update can alternatively be stated by the following simple recursion:

$$m_{t,i} = \beta_1 m_{t-1,i} + (1-\beta_1) g_{t,i}, \quad v_{t,i} = \beta_2 v_{t-1,i} + (1-\beta_2) g_{t,i}^2 \tag{1}$$

and $m_{0,i} = 0$ and $v_{0,i} = 0$ for all $i \in [d]$, and $t \in [T]$. A value of $\beta_1 = 0.9$ and $\beta_2 = 0.999$ is typically recommended in practice. We note the additional projection operation in Algorithm **??** in comparison to ADAM. When $\mathcal{F} = \mathbb{R}^d$, the projection operation is an identity operation and this corresponds to the algorithm in (Kingma & Ba, 2015).
For theoretical analysis, one requires $\alpha_t = 1/\sqrt{t}$ for $t \in [T]$, although a more aggressive choice of constant step size seems to work well in practice. RMSPROP, which appeared in an earlier unpublished work (Tieleman & Hinton, 2012), is essentially a variant of ADAM with $\beta_1 = 0$. In practice, especially in deep learning applications, the momentum term arising due to non-zero β_1 appears to significantly boost the performance. We will mainly focus on ADAM algorithm due to this generality but our arguments also apply to RMSPROP and other algorithms such as ADADELTA, NADAM.

7 ADAptive Moment estimation (ADAM)

Adaptive Moment Estimation (Adam) **ref14** is another method that computes adaptive learning rates for each parameter. In addition to storing an exponentially decaying average of past squared gradients v_t like Adadelta and RMSprop, Adam also keeps an exponentially decaying average of past gradients m_t, similar to momentum. Whereas momentum can be seen as a ball running down a slope, Adam behaves like a heavy ball with friction, which thus prefers flat minima in the error surface **ref15**. We compute the decaying averages of past and past squared gradients m_t and v_t respectively as follows:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

m_t and v_t are estimates of the first moment (the mean) and the second moment (the uncentered variance) of the gradients, respectively, hence the name of the method. As m_t and v_t are initialized as vectors of 0's, the authors of Adam observe that they are biased towards zero, especially during the initial time steps, and especially when the decay rates are small (i.e. β_1 and β_2 are close to 1).

They counteract these biases by computing bias-corrected first and second moment estimates:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$
$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

They then use these to update the parameters just as we have seen in Adadelta and RMSprop, which yields the Adam update rule:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon}\hat{m}_t.$$

The authors propose default values of 0.9 for β_1, 0.999 for β_2, and 10^{-8} for ϵ. They show empirically that Adam works well in practice and compares favorably to other adaptive learning-method algorithms.

7.1 The Non-Convergence of ADAM

With the problem setup in the previous section, we discuss fundamental flaw in the current exponential moving average methods like ADAM. We show that ADAM can fail to converge to an optimal solution even in simple one-dimensional convex settings. These examples of non-convergence contradict the claim of convergence in (Kingma & Ba, 2015), and the main issue lies in the following quantity of interest:

$$\Gamma_{t+1} = \left(\frac{\sqrt{V_{t+1}}}{\alpha_{t+1}} - \frac{\sqrt{V_t}}{\alpha_t} \right). \tag{2}$$

This quantity essentially measures the change in the inverse of the learning rate of the adaptive method with respect to time. One key observation is that for SGD and ADAGRAD, $\Gamma_t \succeq 0$ for all $t \in [T]$. This simply follows from update rules of SGD and ADAGRAD in the previous section. In particular, update rules for these algorithms lead to "non-increasing" learning rates. However, this is not necessarily the case for exponential moving average variants like ADAM and RMSPROP, i.e., Γ_t can potentially be indefinite for $t \in [T]$. We show that this violation of positive definiteness can lead to undesirable convergence behavior for ADAM and RMSPROP.
Consider the following simple sequence of linear functions for $\mathcal{F} = [-1, 1]$:

$$f_t(x) = \begin{cases} Cx, & \text{for } t \mod 3 = 1 \\ -x, & \text{otherwise,} \end{cases}$$

where $C > 2$. For this function sequence, it is easy to see that the point $x = -1$ provides the minimum regret. Suppose $\beta_1 = 0$ and $\beta_2 = 1/(1 + C^2)$. We show that ADAM converges to a highly suboptimal solution of $x = +1$ for this setting. Intuitively, the reasoning is as follows. The algorithm obtains the large gradient C once every 3 steps, and while the other 2 steps it observes the gradient -1, which moves the algorithm in the wrong direction. The large gradient C is unable to counteract this effect since it is scaled down by a factor of almost C for the given value of β_2, and hence the algorithm converges to 1 rather than -1. We formalize this intuition in the result below.

8 Future and changes expected

Looking ahead, the field of optimization in machine learning is rapidly evolving. One direction involves improved versions of Adam, such as AdaBelief and Lion,

which aim to combine adaptivity with better generalization and robustness. On the theoretical front, researchers are exploring why stochastic gradient descent (SGD) performs so well on highly non-convex problems like deep neural networks. Ongoing studies also focus on efficient pre-training methods, improving training stability, developing stronger convergence guarantees, and uncovering scaling laws that govern model performance as size and data increase. These advances promise to make optimization algorithms more efficient, reliable, and theoretically grounded for the next generation of AI models.

9 Conclusion

In this paper, we've explored the mathematical foundations that drive how we train modern machine learning models, especially when dealing with the challenging landscapes of non-convex optimization. We began with the basic concept of minimizing a loss function, then examined the differences between convex and non-convex functions, and how those differences impact optimization. We learned how gradient descent and its stochastic variants navigate these landscapes, and how tools like L-smoothness and the Polyak–Łojasiewicz (PL) condition help us prove convergence, even in non-convex settings. We also discussed practical solutions like adding random noise to escape saddle points, and how optimizers like AdamW further enhance stability and generalization. Finally, we looked at how these ideas power real-world models like Vision Transformers.

As machine learning continues to tackle increasingly complex problems, a strong mathematical understanding of optimization will remain essential, not just for making models work, but for making them faster, more stable, and more trustworthy. The future holds exciting advances in theory and practice, and the tools we've covered today form the bedrock of that progress.

10 References

Works Cited

Bengio, Yoshua. Representation Learning and Deep Learning. Université de Montréal Course Material. www.iro.umontreal.ca/~bengioy/dlbook/.

Bottou, Léon, et al. "Optimization Methods for Large-Scale Machine Learning". *SIAM Review*, vol. 60, no. 2, 2018, pp. 223–311.

Boyd, Stephen, and Lieven Vandenberghe. *Convex Optimization*. Cambridge UP, 2004.

Bubeck, Sébastien. "Convex Optimization: Algorithms and Complexity". *Foundations and Trends in Machine Learning*, vol. 8, nos. 3–4, 2015, pp. 231–357.

Chong, Edwin K. P., and Stanislaw H. Zak. *An Introduction to Optimization*. 4th ed., Wiley, 2013.

Duchi, John. CS229R: Nonconvex Optimization. 2022, Stanford University Course Notes. web.stanford.edu/class/cs229r/.

Foret, Pierre, et al. "Sharpness-Aware Minimization for Efficiently Improving Generalization". *arXiv preprint arXiv:2010.01412*, 2021.

Goodfellow, Ian, et al. *Deep Learning*. MIT P, 2016.

Jin, Chi, et al. "How to Escape Saddle Points Efficiently". *arXiv preprint arXiv:1703.00887*, 2017.

Karimi, Hessam, et al. "Linear Convergence of Gradient and Proximal-Gradient Methods under the Polyak–Łojasiewicz Condition". *arXiv preprint arXiv:1608.04636*, 2016.

Sra, Suvrit, et al. *Optimization for Machine Learning*. MIT P, 2011.

11 Appendix - Math for the Paper

Linear Algebra Background

Gradient – In the context of multivariable calculus, the gradient of a function, denoted by ∇f, is a vector field that points in the direction of the greatest rate of increase of the function at any given point, and whose magnitude is that rate of increase.

$$\nabla f = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \right).$$

Hessian – A Hessian matrix is a square matrix of second-order partial derivatives of a scalar-valued function. It essentially describes the local curvature of the function, providing information about its concavity or convexity at a given point.

For example, if a function has two variables (x and y), the Hessian matrix will include terms like $\frac{\partial^2 f}{\partial x^2}$, $\frac{\partial^2 f}{\partial y^2}$, and $\frac{\partial^2 f}{\partial x \partial y}$ (and $\frac{\partial^2 f}{\partial y \partial x}$, which is equal to $\frac{\partial^2 f}{\partial x \partial y}$ due to Schwarz's theorem).

Jacobian – The Jacobian matrix represents the differential of f at every point where f is differentiable. The Jacobian matrix, whose entries are functions of x, is denoted in various ways; other common notations include Df, ∇f. Some authors define the Jacobian as the transpose of the form given above $\nabla^T f$.

Then the Jacobian matrix of function f, denoted J_f, is the $m \times n$ matrix whose (i, j) entry is $\frac{\partial f_i}{\partial x_j}$.