Stochastic Gradient Descent: A Fundamental Method in Stochastic Optimization

Shihan Kanungo

Abstract

Stochastic Gradient Descent (SGD) is a cornerstone algorithm in modern optimization, especially prevalent in large-scale machine learning. This paper introduces the theoretical foundation of SGD, contrasts it with deterministic gradient descent, and explores its convergence properties, practical implementation considerations, and typical applications in applied mathematics and data science. We also give some basic numerical simulations which showcase the strengths of different variants of SGD.

1 Introduction

Optimization lies at the heart of many problems in applied mathematics, data science, and engineering. In particular, minimizing a loss function to fit models to data is ubiquitous. Gradient descent is a classical method for such optimization tasks. However, in contexts involving massive datasets, evaluating the full gradient at each step becomes computationally prohibitive. **Stochastic Gradient Descent (SGD)**, a stochastic approximation of gradient descent, offers a scalable alternative.

SGD introduces randomness into the optimization process by estimating gradients from random subsets of data. This makes the method both faster and more noise-resilient, but also introduces analytical complexity. However, there are many ways to improve the speed and convergence of SGD, and in general, it is far more efficient to use SGD rather than standard gradient descent.

SGD is ubiquitous in training neural networks, which have datasets that are far too large for standard gradient descent to deal with.

In Section 2 we introduce SGD and discuss its convergence properties.

There are many different variations on vanilla SGD that deal with some of its weaknesses. Mini-batch gradient descent combines the strengths of SGD and standard gradient descent, and it is the standard method for neural networks. Momentum is an approach that helps SGD navigate ravines, which are points where the loss function curves much more steeply in one direction than another. Adagrad is a method that automatically adjusts the step size of the algorithm, and it gives very fast convergence compared to the other methods. We discuss these approaches and others in more detail in Section 3.

Finally, in Section 4 we provide some numerical examples highlighting the strengths of the different variations of SGD.

2 Stochastic Gradient Descent

Let us consider the standard optimization problem:

$$\min_{\theta \in \mathbb{R}^d} f(\theta),$$

where $f : \mathbb{R}^d \to \mathbb{R}$ is a differentiable function and θ is the vector of parameters. Typically f has the form

$$f(\theta) = \frac{1}{n} \sum_{i=1}^{n} f_i(\theta),$$

where $f_i(\theta)$ is the loss function corresponding to the i'th point in the dataset.

As an example, consider least-squares linear regression, with data points $(x_1, y_1), \ldots, (x_n, y_n)$. We want to find the line $\theta_0 + \theta_1 x$ minimizing

$$f(\theta) = \frac{1}{n} \sum_{i=1}^{n} (\theta_0 + \theta_1 x_i - y_i)^2.$$

Gradient Descent updates parameters via:

$$\theta_{k+1} = \theta_k - \eta_k \nabla f(\theta_k),$$

where η_k is the step size (learning rate).

However, computing the gradient ∇f can be computationally expensive when n is very large. This happens in situations such as machine learning, where the training dataset is extremely big.

Stochastic Gradient Descent (SGD) provides a way to circumvent this problem. Instead of taking the full gradient of f, at each step it takes the gradient of a randomly chosen loss function f_{i_k}. In other words, the iteration step is

$$\theta_{k+1} = \theta_k - \eta_k \nabla f_{i_k}(\theta_k),$$

where $i_k \in \{1, \ldots, n\}$ is randomly chosen.[1] This process is much less expensive, because we only need to take the gradient of one of the f_i's at each step, rather than all n of them. Compared to standard gradient descent, SGD has some drawbacks. First of all, the convergence rate of SGD is significantly slower. Second, since SGD is a random process, it will inevitably have some noise, which makes it less accurate. But on the plus side, this noise lets it escape local minima and converge to a true minimum.

Let's see what the explicit form of SGD is in our example. We have

$$f_i(\theta) = (\theta_0 + \theta_1 x_i - y_i)^2.$$

[1] In general, we sample the i_k by randomly shuffling $1, \ldots, n$ and repeating. This ensures that each f_i is chosen the same number of times.

Hence,

$$\nabla f_i(\theta) = \begin{bmatrix} 2(\theta_0 + \theta_1 x_i - y_i) \\ 2x_i(\theta_0 + \theta_1 x_i - y_i) \end{bmatrix}.$$

Thus, the iteration step of SGD has the form

$$\theta_{k+1} = \theta_k - \eta_k \nabla \begin{bmatrix} 2(\theta_0 + \theta_1 x_{i_k} - y_{i_k}) \\ 2x_{i_k}(\theta_0 + \theta_1 x_{i_k} - y_{i_k}) \end{bmatrix}.$$

Finally, there is a third type of gradient descent, called **Mini-Batch Gradient Descent**, which gives us the "best of both worlds". Instead of sampling a single f_i at a time, we sample m at a time for some $m < n$, so the iteration step is

$$\theta_{k+1} = \theta_k - \eta_k \sum_{p=1}^{m} f_{i_k^{(p)}}(\theta_k),$$

where $i_k^{(1)}, \ldots, i_k^{(m)}$ are randomly chosen. Additionally, it can make use of highly optimized matrix optimizations that make computing the gradient with respect to a mini-batch This reduces variance of parameter updates and hence increases stability. Mini-Batch gradient descent is the method of choice for training neural networks, which have huge data sets.

2.1 Convergence of SGD

When f is a convex function, gradient descent is guaranteed to converge to the global minimum, even when the step size η is held constant. Furthermore, it can be shown that under certain conditions on f, the convergence rate is in fact linear.

However, SGD does not behave as nicely. In particular, we need

$$\lim_{k \to \infty} \frac{\sum_{t=1}^{k} \eta_k^2}{\sum_{t=1}^{k} \eta_k} = 0.$$

Thus, a constant step size will not give us convergence, and instead SGD will oscillate around the minimum. We need to have a decreasing step size for SGD to converge. Assuming a step size of the form

$$\frac{\eta}{k^\alpha}, \qquad \alpha = \tfrac{1}{2}$$

gives the best convergence rate. Unfortunately, even this does not give the linear convergence rate enjoyed by standard gradient descent.

Thus, SGD provides more efficiency at the cost of a lower convergence rate.

3 Variations of SGD

In this section, we discuss some extensions of SGD that offer solutions to the drawbacks of standard SGD. For simplicity, we will only consider the single sample case, but these can all be generalized to the mini-batch case.

3.1 Momentum

Standard SGD has trouble navigating ravines, which are areas where the surface curves much more steeply in one direction than the other. In this type of situation, SGD oscillates in the ravine and only makes slow progress along the bottom, towards the minimum.

Momentum is a way to fix this problem. As the title suggest, the update vector v_{k+1} has an extra term that "remembers" the previous step, so we tend to go in the same direction as before. Explicity:

$$\begin{aligned} v_t &= \gamma v_{t-1} + \eta \nabla f_{i_k}(\theta_k) \\ \theta_{t+1} &= \theta_t - v_t. \end{aligned}$$

Here γ is a constant which is usually set to be $1 - \varepsilon$ for a small positive constant ε. The momentum approach can be thought of as rolling a ball down a hill, and the ball gets faster and faster until it hits its terminal velocity. This helps SGD deal with ravines and also makes convergence faster.

3.2 Nesterov Accelerated Gradient

Unfortunately, the momentum approach has some drawbacks of its own. Consider valley that the ball is rolling into. With the momentum approach, the ball will be hurtling down so fast that it will overshoot the minimum before it sees that it should turn around, and then it will start accelerating. We would like a more intelligent ball that can sense when hill is starting to slope. What Nesterov Accelerated Gradient does is to take the gradient *after* we take the momentum step. In other words:

$$\begin{aligned} v_t &= \gamma v_{t-1} + \eta \nabla f_{i_k}(\theta_k - \gamma v_{t-1}) \\ \theta_{t+1} &= \theta_t - v_t. \end{aligned}$$

This way, the ball knows when to slow down.

3.3 Adagrad

In both of the previous two variations, the step size η remains constant. However, an adaptive step size makes the algorithm both more accurate and more efficient.

The basic idea is as follows: parameters with low-frequency features (i.e. the gradient is small) should have large step size, and parameters with high-frequency features (i.e. the gradient is large) should have small step size.

We introduce the notation:

$$g_{k,i} = (\nabla f_i(\theta_k))_i,$$

and define the diagonal matrix G_k to have the (i, i)-th entry equal to the sum of the squares of all the previous $g_{k,i}$, i.e.

$$(G_k)_{ii} = \sum_{t=0}^{k} g_{t,i}^2.$$

Then the update step has the form

$$(\theta_{k+1})_i = (\theta_k)_i - \frac{\eta}{\sqrt{(G_t)_{ii} + \epsilon}} \cdot g_{k,i},$$

or in vector form,

$$\theta_{k+1} = \theta_k - \frac{\eta}{\sqrt{G_k + \epsilon}} \odot g_k,$$

with $\odot$ denoting element-wise vector multiplication. The ϵ term is simply there to avoid division by zero. Adagrad performs much better than standard SGD on sparse datasets. It was used to train large-scale neural networks at Google and was able to recognize cats in Youtube videos. Additionally, SGD famously has trouble escaping saddle points. Adagrad (and RMSprop below) are much better at dealing with saddle points.

However, Adagrad has a significant drawback, which is its aggressively decreasing learning rate. Variants of Adagrad deal with this using a similar idea to the Momentum approach. One such method is RMSprop.

3.4 RMSprop

RMSProp, or Root Mean Square Propagation, is an adaptive learning rate optimization algorithm used in training deep neural networks. It adjusts the learning rate for each parameter based on the average of the squared gradients, helping to improve convergence speed and stability during training.

The idea of RMSprop is to restrict the window of the past squared gradients to some fixed size. We define a weighted average $\mathbb{E}(g^2)_k$ recursively by

$$\mathbb{E}(g^2)_{k+1} = \gamma \mathbb{E}(g^2)_k + (1-\gamma) g_k^2,$$

where γ, like in the Momentum approach, is a constant less than 1. This way, RMSprop can adapt to the local behavior of the loss function. Then our update will be

$$\theta_{k+1} = \theta_k - \frac{\eta}{\sqrt{\mathbb{E}[g^2]_k + \epsilon}} g_k.$$

There are other variations, such as Adadelta and Adam, which behave similarly but are a bit more complicated.

4 Numerical examples

A Python implementation of SGD for the least-squares problem discussed earlier gives the following outputs, shown in **Fig. 1.** and **Fig. 2.**

As seen from **Fig. 2.**, the loss function $Q(\theta)$ has noise due to random fluctuations, but it does tend to the minimum value. In contrast, standard gradient descent would have a smooth plot.

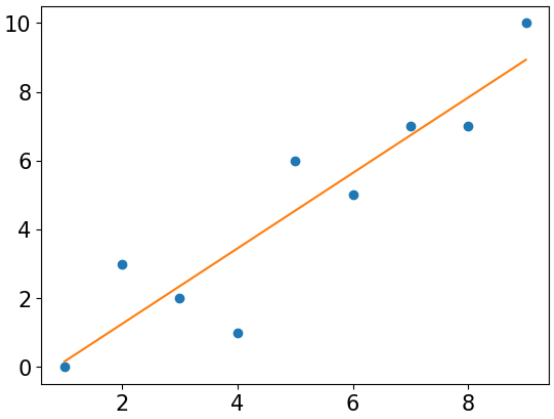

Fig. 1. Least-squares minimizing line

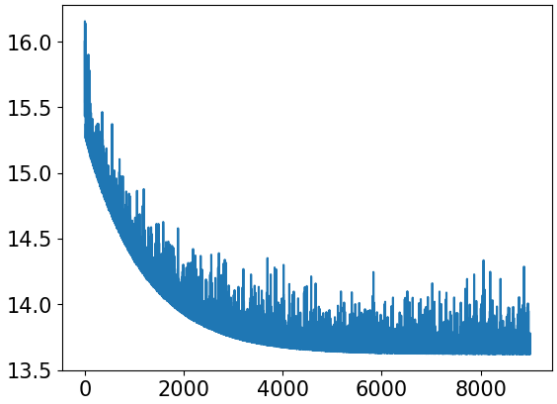

Fig. 2. $Q(\theta_k)$ versus k

Next, implementing Momentum, we get the following plot comparing it to SGD:

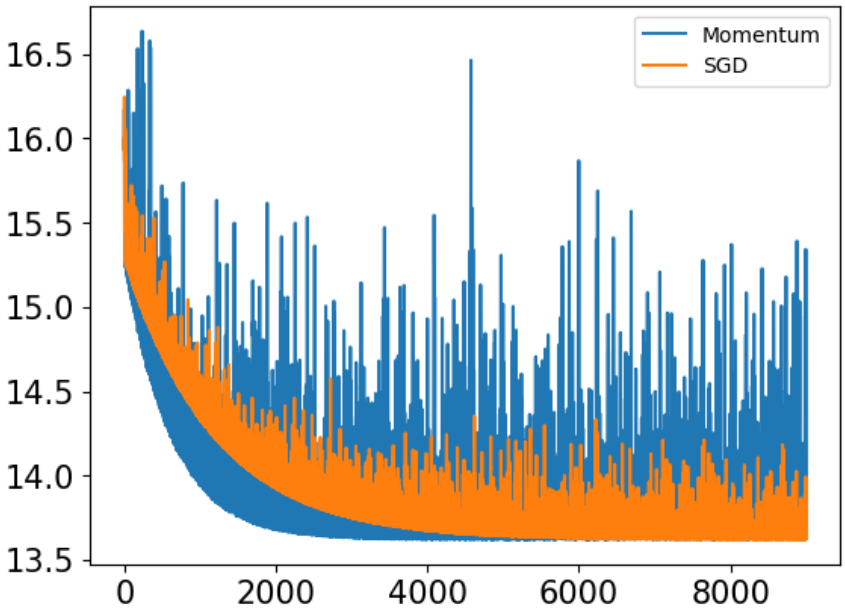

Fig. 3. Momentum vs. standard SGD

As expected, Momentum converges faster than SGD.

Next, we look at a similar problem involving a very large dataset, and show how SGD is more efficient than standard gradient descent.

Consider data points of the form (x_1, x_2, y) where $y = 2x_1 - 3x_2 + \text{noise}$. Similarly to the least-squares case, we use $\hat{y} = \theta_1 x_1 + \theta_2 x_2$ as our estimator, and we want to minimize the mean square error (i.e. least-squares regression). Implementing both standard gradient descent and SGD, the comparisons of the loss functions are shown:

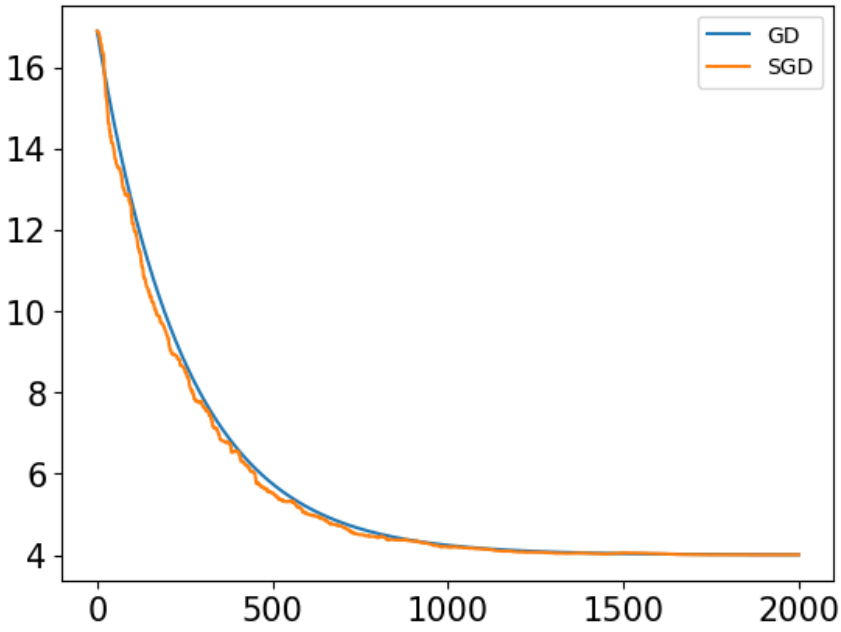

Fig. 4. SGD vs Standard Gradient Descent

They give equally good results, but SGD is about 3 times faster for $5 \cdot 10^5$ data points, and even faster for more.

References

1. Bottou, L. (2010). *Large-Scale Machine Learning with Stochastic Gradient Descent.* Proceedings of COMPSTAT.
2. Bubeck, S. (2015). *Convex Optimization: Algorithms and Complexity.* Foundations and Trends in Machine Learning.
3. Garrigos, G. and Gower, R. M. (2023). *Handbook of Convergence Theorems for (Stochastic) Gradient Methods.* Preprint [arXiv]
4. Monro, S. and Robbins, H. (1951). *A Stochastic Approximation Method.* Annals of Mathematical Statistics.
5. Ruder, S. (2016). *An Overview of Gradient Descent Optimization Algorithms.* Preprint [arXiv]

Department of Mathematics, San José State University, San José, CA 95192

Email address: shihan.kanungo@sjsu.edu

Appendix: Code

Here is the code used for the first plot:

```
import matplotlib.pyplot as plt
import numpy as np

plt.rc('xtick', labelsize=15)
plt.rc('ytick', labelsize=15)

# step size
eta = 0.001

# data
x = [1,2,3,4,5,6,7,8,9]
y = [0,3,2,1,6,5,7,7,10]

n = 9

# loss function
def Q(w):
  q = 0
  for i in range(n):
    q += (w[0]+w[1]*x[i]-y[i])**2
  return q

# initialize parameters
w = np.empty(2)
w[0]=0
w[1]=1

# number of steps
n_steps = 1000

t_list = [0]
t = 0
Q_list = [Q(w)]

# create a list of the indices from 0 to n
indices = []
for i in range(n):
  indices.append(i)

# implement SGD
for _ in range(n_steps):
  s = np.random.permutation(indices)
  for i in range(n):
    w -= eta*np.array([2*(w[0]+w[1]*x[s[i]]-y[s[i]]),2*x[s[i]]*(w
    [0]+w[1]*x[s[i]]-y[s[i]])])
    t += 1
    t_list.append(t)
    Q_list.append(Q(w))

# plot the data along with the least-squares appproximation
a = np.linspace(min(x),max(x),2)
```

```
b = w[0]+w[1]*a
plt.plot(x,y, "o")
plt.plot(a,b)
plt.show()

# plot the loss function versus time
plt.plot(t_list,Q_list)
```

To implement the Momentum algorithm, we only need to make a small modification to the code:

```
v = np.array([0,0])
# implement Momentum
for _ in range(n_steps):
  s = np.random.permutation(indices)
  for i in range(n):
    vnew = eta *np.array([2*(w[0]+w[1]*x[s[i]]-y[s[i]]),2*x[s[i
    ]]*(w[0]+w[1]*x[s[i]]-y[s[i]])])
    w -= gamma * v
    v = vnew
    w -= v
    t += 1
    t_list.append(t)
    Q_list.append(Q(w))
```

For the third plot, I used the following code

```
import matplotlib.pyplot as plt
import numpy as np
import time

plt.rc('xtick', labelsize=15)
plt.rc('ytick', labelsize=15)

N = 500000

X = np.random.randn(N, 2)
true_w = np.array([2.0, -3.0])
y = X @ true_w + np.random.randn(N) * 2  # add noise

def compute_loss(w, X, y):
    preds = X @ w
    return np.mean((preds - y) ** 2)

def compute_gradient(w, X, y):
    return 2 * X.T @ (X @ w - y) / len(y)

def gradient_descent(X, y, eta=0.01, steps=1000):
    start_time = time.perf_counter()
    w = np.zeros(2)
    losses = []
```

```
    for _ in range(steps):
        grad = compute_gradient(w, X, y)
        w -= eta * grad
        losses.append(compute_loss(w, X, y))
    end_time = time.perf_counter()
    print(end_time-start_time)
    return w, losses

def stochastic_gradient_descent(X, y, eta=0.01, steps=1000):
    start_time = time.perf_counter()
    w = np.zeros(2)
    losses = []
    for i in range(steps):
        idx = np.random.randint(len(X))
        xi = X[idx:idx+1]
        yi = y[idx:idx+1]
        grad = compute_gradient(w, xi, yi)
        w -= eta * grad
        losses.append(compute_loss(w, X, y))  # eval on full data
    end_time = time.perf_counter()
    print(end_time-start_time)
    return w, losses

# print(gradient_descent(X,y,0.001,100000)[0])
gd = gradient_descent(X,y,0.001,2000)
sgd = stochastic_gradient_descent(X,y,0.001,2000)
plt.plot(gd[1],label = 'GD')
plt.plot(sgd[1],label = 'SGD')
print(gd[0])
print(sgd[0])
plt.legend()
```