Week #9: Stochastic Optimization

March 10, 2025

1 Stochastic Optimization

An optimization problem seeks to find the minimum or maximum of a function subject to constraints, formally defined as:

$$\text{Minimize } f(x) \quad \text{subject to } x \in \mathcal{X},$$

where $f : \mathbb{R}^n \to \mathbb{R}$ is the objective function and $\mathcal{X} \subseteq \mathbb{R}^n$ denotes the feasible set.

2 Evolutionary Algorithms

Evolutionary Algorithms (EAs) are a class of stochastic optimization algorithms inspired by biological evolution. They operate on a population P of candidate solutions (individuals), evolving this population through iterations (generations) using mechanisms analogous to those found in natural selection and genetics.

An evolutionary algorithm can generally be described by the following steps:

1. **Initialization:** Generate an initial population $P^{(0)}$ of N individuals randomly. Each individual $x_i^{(0)}$ represents a potential solution to the optimization problem.

2. **Evaluation:** Compute the fitness $f(x_i)$ for each individual x_i in the population, where $f : \mathcal{X} \to \mathbb{R}$ is the fitness function that measures the quality of solutions.

3. **Selection:** Select individuals based on their fitness to form a mating pool. The selection probability $P_s(x_i)$ of an individual x_i often depends on its relative fitness to the population:

$$P_s(x_i) = \frac{f(x_i)}{\sum_{j=1}^{N} f(x_j)}$$

4. **Crossover:** Apply the crossover operator to pairs of individuals in the mating pool to produce offspring, forming a new generation. A common method is the arithmetic crossover, where the offspring is a weighted average of the parents:

$$x_{\text{offspring}} = \alpha x_{\text{parent1}} + (1 - \alpha) x_{\text{parent2}}$$

   where x_{parent1} and x_{parent2} are the genetic vectors of the two selected parents, and α is a randomly chosen weight between 0 and 1.

5. **Mutation:** Apply the mutation operator with a small probability to new offspring to maintain genetic diversity within the population. Mutation randomly alters the genes of individuals, modeled as:

$$x_i' = x_i + \epsilon, \quad \epsilon \sim \mathcal{N}(0, \sigma^2)$$

   where ϵ represents a small random perturbation.

6. **Replacement:** Generate the next generation population by replacing some or all of the older individuals with new offspring, based on their fitness.

Theorem 1. Schema Theorem and Global Convergence

Under ideal conditions, such as infinite population sizes and generations, evolutionary algorithms can converge to a global optimum in continuous optimization problems. In genetic algorithms, patterns (schemata) with higher fitness than average tend to increase in frequency over generations. This is expressed as:

$$E(m_{H,t+1}) \geq m_{H,t} \frac{f_H}{\bar{f}}$$

where $m_{H,t}$ is the number of instances of schema H at generation t, f_H is the average fitness of those instances, and $\bar{f}$ is the average fitness of the population. This shows that favorable genetic patterns spread more, guiding the population towards better solutions.

Evolutionary algorithms exhibit robust performance across a broad spectrum of problems, benefiting from their ability to avoid local optima through stochastic behaviors such as mutations. The convergence rate can vary greatly and is influenced by factors such as the fitness landscape and algorithm parameters like mutation rates and crossover methods.

The complexity of evolutionary algorithms is primarily influenced by the population size, the dimensionality of the solution space, and the computational cost of fitness evaluations. The long-term behavior and stability of these algorithms can be modeled using dynamical systems theory and Markov chains, ensuring that performance does not deteriorate over time.

2.0.1 Genetic Algorithms (GA)

- Typically use binary strings or fixed-length vectors.
- Example: $\mathbf{x}_i = [x_{i1}, x_{i2}, \ldots, x_{in}]$, where $x_{ij} \in \{0, 1\}$.

Regarding mutation:

- Usually involves flipping bits in a binary string.
- Example: If $\mathbf{x}_i = [1, 0, 1]$, mutation might change it to $\mathbf{x}_i' = [1, 1, 1]$.

Crossover:

- Combines parts of two parent solutions to create offspring.
- Example: Single-point crossover:

$$\text{Parent 1: } \mathbf{x}_i = [1, 0, 1 \mid 0, 1, 1]$$

$$\text{Parent 2: } \mathbf{x}_j = [0, 1, 0 \mid 1, 0, 0]$$

$$\text{Offspring: } \mathbf{y}_k = [1, 0, 1 \mid 1, 0, 0]$$

Selection:

- Based on fitness, using methods like roulette wheel or tournament selection.
- Example: Probability of selection $P(\mathbf{x}_i) = \frac{f(\mathbf{x}_i)}{\sum_j f(\mathbf{x}_j)}$.

2.0.2 Differential Evolution (DE)

Representation:

- Uses real-valued vectors.
- Example: $\mathbf{x}_i = [x_{i1}, x_{i2}, \dots, x_{in}]$, where $x_{ij} \in \mathbb{R}$.

Mutation:

- Creates a donor vector by adding the weighted difference between two population vectors to a third vector.
- Example:
$$\mathbf{v}_i = \mathbf{x}_{r1} + F \cdot (\mathbf{x}_{r2} - \mathbf{x}_{r3})$$
where $r1, r2, r3$ are distinct indices, and F is a scaling factor.

Crossover:

- Combines donor vector with target vector to create a trial vector.
- Example: Binomial crossover:
$$u_{ij} = \begin{cases} v_{ij} & \text{if } \text{rand}_j(0,1) \leq CR \text{ or } j = j_{rand} \\ x_{ij} & \text{otherwise} \end{cases}$$
where CR is the crossover probability.

Selection:

- Compares trial vector with target vector and selects the one with better fitness.
- Example:
$$\mathbf{x}_i^{(t+1)} = \begin{cases} \mathbf{u}_i & \text{if } f(\mathbf{u}_i) \leq f(\mathbf{x}_i) \\ \mathbf{x}_i & \text{otherwise} \end{cases}$$

3 Stochastic gradient descent

Deterministic Optimization

For convex and differentiable functions, the gradient descent update rule is:

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

ensuring convergence to the global minimum under appropriate conditions on the step size α_k.

[Gradient Descent on a Quadratic Function] Consider minimizing a simple quadratic function $f(x) = \frac{1}{2}x^2$. The gradient of this function is $\nabla f(x) = x$, and applying the gradient descent update rule with a fixed step size $\alpha_k = 0.1$, starting from $x_0 = 10$, we get:

$$x_1 = 10 - 0.1 \times 10 = 9,$$
$$x_2 = 9 - 0.1 \times 9 = 8.1,$$

and so forth. This sequence shows that x_k converges to 0 as k increases, illustrating the rate at which values approach the minimum.

Newton's Method: This is another powerful technique in deterministic optimization, particularly useful for functions that are twice differentiable. It improves upon gradient descent by considering the curvature of the function. The update formula for Newton's method is:

$$x_{k+1} = x_k - H^{-1}(x_k)\nabla f(x_k),$$

where $H^{-1}(x_k)$ is the inverse of the Hessian matrix at x_k. This method provides quadratic convergence near the optimum under suitable conditions, significantly faster than the linear convergence of gradient descent.

However, deterministic methods can be limited in practical scenarios where the objective function is noisy or only partially known, which leads us to Stochastic Optimization.

Stochastic optimization is necessary when the function f is noisy or evaluations are computationally expensive, requiring methods that can handle uncertainty and incomplete information.

Updates in Stochastic Gradient Descent (SGD) are based on noisy estimates of the gradient:

$$\beta_{k+1} = \beta_k - \alpha_k \tilde{\nabla} f(\beta_k),$$

where α_k is the learning rate, and $\tilde{\nabla} f(\beta_k)$ represents a stochastic approximation of the gradient.

Monte Carlo Approximation

Monte Carlo methods estimate expectations by random sampling. To approximate the gradient $\nabla f(\beta_k)$ when the exact calculation is computationally expensive, we use:

$$\tilde{\nabla} f(\beta_k) \approx \frac{1}{N}\sum_{i=1}^{N} \nabla f(\beta_i),$$

where β_i are independent and identically distributed (i.i.d.) samples drawn from the distribution of β, and N is the number of samples. This approach is particularly useful when the gradient $\nabla f(\beta)$ cannot be computed exactly or when it's too expensive to evaluate over the entire dataset.

Monte Carlo methods leverage the law of large numbers, which states that as the number of samples N increases, the average of the sampled gradients converges to the true gradient. This reduces the variance of the gradient estimate, making the update direction more reliable while maintaining computational efficiency.

3.0.1 ADAM Optimizer

ADAM (Adaptive Moment Estimation) is an extension of stochastic gradient descent that incorporates momentum and adaptive learning rates for each parameter. It combines the advantages of two other extensions of stochastic gradient descent, namely AdaGrad and RMSProp.

Key Features of ADAM:

- It stores an exponentially decaying average of past squared gradients v_t and an exponentially decaying average of past gradients m_t.
- It corrects these moments to counteract their initialization at the origin.

Update Rule: The parameters are updated as follows:

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1-\beta_1)\nabla f(x_t), \\ v_t &= \beta_2 v_{t-1} + (1-\beta_2)(\nabla f(x_t))^2, \\ \hat{m}_t &= \frac{m_t}{1-\beta_1^t}, \\ \hat{v}_t &= \frac{v_t}{1-\beta_2^t}, \\ x_{t+1} &= x_t - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t}+\epsilon}. \end{aligned}$$

where:

- β_1 and β_2 are parameters controlling the exponential decay rates of these moving averages.
- α is the learning rate.
- ϵ is a small scalar used to prevent division by zero.

This update mechanism adjusts the learning rate for each parameter based on estimates of first and second moments of the gradients, enabling efficient convergence even for functions with noisy or sparse gradients.

Advanced Gradient Methods focus on modern techniques with momentum and adaptive rates, crucial for applications in deep learning. Distributed Optimization cover optimization over finite sums and distributed techniques, including the decentralized gradient descent model:

$$x_i^{k+1} = x_i^k - \alpha_k \left(\nabla f_i(x_i^k) + \sum_{j \in \mathcal{N}_i} w_{ij}(x_i^k - x_j^k) \right).$$

Mini-Batch Gradient Descent

Another common approach in SGD is to use a mini-batch of data points to approximate the gradient. Let B_k be the mini-batch at iteration k. The approximate gradient is:

$$\tilde{\nabla} f(\beta_k) = \frac{1}{|B_k|} \sum_{i \in B_k} \nabla f_i(\beta_k),$$

where $f_i(\beta_k)$ is the objective function evaluated at sample i in the mini-batch, and $|B_k|$ denotes the number of samples in the mini-batch.

Why These Methods Work

- *Stochastic Nature*: Functions like $f(\beta)$ might represent expectations over a probability distribution. By sampling, we capture the variability in the data, and the average gradient of these samples provides an unbiased estimate of the true gradient.
- *Law of Large Numbers*: As the number of samples increases, the average of the sampled gradients converges to the true gradient, reducing the variance of the estimate.
- *Computational Efficiency*: Using subsets of data or random samples significantly reduces the computational cost per iteration compared to deterministic gradient descent.

Both Monte Carlo approximation and mini-batch gradient descent are essential for efficiently handling large-scale and noisy optimization problems, offering a balance between computational efficiency and gradient estimation accuracy.

3.0.2 Distributed Optimization

Distributed optimization involves performing optimization tasks across multiple processors or nodes to reduce the computational burden on a single processor and enhance scalability. Let $\mathcal{D}$ be the entire dataset, partitioned into M subsets $\mathcal{D}_m$ such that:

$$\mathcal{D} = \bigcup_{m=1}^{M} \mathcal{D}_m.$$

Each node m holds a subset $\mathcal{D}_m$ and performs local computations. The global optimization problem is:

$$\min_{x \in \mathbb{R}^n} \frac{1}{M} \sum_{m=1}^{M} F_m(x),$$

where $F_m(x) = \frac{1}{|\mathcal{D}_m|} \sum_{i \in \mathcal{D}_m} f_i(x)$.

In distributed gradient descent, each node computes its local gradient:

$$\tilde{\nabla} F_m(x_k) = \frac{1}{|\mathcal{D}_m|} \sum_{i \in \mathcal{D}_m} \nabla f_i(x_k),$$

and the gradients are aggregated to update the global parameter x:

$$x_{k+1} = x_k - \alpha_k \frac{1}{M} \sum_{m=1}^{M} \tilde{\nabla} F_m(x_k).$$

3.0.3 Decentralized Optimization

Decentralized optimization is performed in a networked system where data or computation is distributed among multiple agents or nodes, minimizing the need for centralized control. Each node m holds its own data $\mathcal{D}_m$ and communicates only with its neighbors. The optimization objective remains the same:

$$\min_{x \in \mathbb{R}^n} \frac{1}{M} \sum_{m=1}^{M} F_m(x).$$

One common method is decentralized gradient descent, where each node updates its parameters based on local data and sporadic communication with neighboring nodes. Let $\mathcal{N}(m)$ denote the set of neighbors of node m. Each node performs the following updates:

1. Compute local gradient:

$$\tilde{\nabla} F_m(x_k) = \frac{1}{|\mathcal{D}_m|} \sum_{i \in \mathcal{D}_m} \nabla f_i(x_k).$$

2. Communicate with neighbors and update local parameter:

$$x_m^{(k+1)} = \sum_{n \in \mathcal{N}(m)} w_{mn} x_n^{(k)} - \alpha_k \tilde{\nabla} F_m(x_m^{(k)}),$$

where w_{mn} are weights representing the influence of neighboring nodes.

This method ensures that each node converges to a common solution while maintaining a decentralized structure, which is crucial for large-scale and privacy-sensitive applications.

Both distributed and decentralized optimization techniques are essential for efficiently solving large-scale optimization problems by leveraging the computational power of multiple nodes and minimizing the drawbacks of centralized control.

4 Expectation-Maximization Algorithm

The Expectation-Maximization (EM) algorithm is an iterative method designed to find the maximum likelihood estimates of parameters in probabilistic models, particularly when data are incomplete or have missing values.

Algorithm Overview

- **Expectation Step (E-step):** Calculate the expected value of the log-likelihood function, with respect to the conditional distribution of the latent variables given the observed data and the current parameter estimates.

$$Q(\theta|\theta^{(t)}) = \mathbb{E}_{Z|X,\theta^{(t)}}[\log L(\theta; X, Z)]$$

  where $\theta^{(t)}$ is the parameter estimate at iteration t, X represents the observed data, Z represents the latent variables, and $L(\theta; X, Z)$ is the complete-data likelihood function.

- **Maximization Step (M-step):** Update the parameters to maximize the expected log-likelihood found in the E-step:

$$\theta^{(t+1)} = \arg\max_{\theta} Q(\theta|\theta^{(t)})$$

The EM algorithm alternates between these two steps until convergence, typically achieving a local maximum of the likelihood function.

[Biomedical Imaging] Consider the task of image reconstruction in positron emission tomography (PET), where the E-step estimates the distribution of decay events given the observed data, and the M-step updates the image to maximize the likelihood of the observed data given the decay events.

Monte Carlo Expectation Maximization (MCEM)

The Monte Carlo Expectation Maximization (MCEM) algorithm extends the EM framework to situations where the E-step cannot be computed analytically. Instead, it relies on Monte Carlo methods to approximate the expectation.

MCEM Procedure

1. **Simulation Step:** Generate samples from the distribution of the latent variables conditioned on the observed data and current parameter estimates.

2. **Monte Carlo E-step:** Approximate the expectation of the complete-data log-likelihood using the sampled values:

$$\tilde{Q}(\theta|\theta^{(t)}) = \frac{1}{N}\sum_{n=1}^{N} \log L(\theta; X, Z^{(n)})$$

   where $Z^{(n)}$ are the samples of latent variables.

3. **M-step:** Maximize the estimated Q-function:

$$\theta^{(t+1)} = \arg\max_{\theta} \tilde{Q}(\theta|\theta^{(t)})$$

Importance Sampling in MCEM

Importance sampling is used in the MCEM to improve the efficiency of the Monte Carlo approximation, especially when direct sampling from the posterior distribution is challenging.

$$\hat{Q}(\theta|\theta^{(t)}) = \sum_{n=1}^{N} w_n \log L(\theta; X, Z^{(n)}) \tag{1}$$

where w_n are the importance weights correcting for the bias in the sampling distribution, and $Z^{(n)}$ are the sampled latent variables.

[Epidemiological Modeling] In modeling the spread of an infectious disease, use MCEM to estimate the transmission rates and recovery rates from incomplete data, where importance sampling addresses the variability in individual susceptibility and recovery rates.

5 Markov Chain Monte Carlo for Optimization

Markov Chain Monte Carlo (MCMC) methods are primarily known for their application in Bayesian statistics for sampling from complex posterior distributions. However, these techniques can also be effectively used for solving optimization problems, especially those involving high-dimensional spaces or complex, multi-modal landscapes.

The Markov chain generated by the Metropolis-Hastings algorithm converges to a stationary distribution that is proportional to $f(x)$. For optimization purposes, $f(x)$ can be chosen as a function that peaks at the global optimum, such as $e^{\beta f(x)}$, where β is a parameter controlling the sharpness of the peak.

In machine learning, MCMC methods can be used to optimize likelihood functions or posterior distributions in settings where direct optimization is challenging due to the complexity of the model or the presence of multiple local optima.

No Free Lunch Theorems

These theorems articulate that no optimization algorithm outperforms others when averaged across all possible problems.

Theorem 2. No Free Lunch Theorems

The performance of any optimization algorithm averaged over all possible problems yields the same result, suggesting that no universally optimal algorithm exists.