Week #12: Stochastic Optimization

Notes by: Francisco Richter

May 15, 2025

Overview

Building upon the foundational concepts of stochastic optimization covered in Week 11, this week we delve deeper into advanced techniques that have revolutionized modern machine learning and large-scale optimization. We explore stochastic gradient methods, distributed optimization frameworks, the Expectation-Maximization algorithm, and Markov Chain Monte Carlo approaches for optimization. These methods are essential for tackling complex, high-dimensional problems where traditional deterministic approaches become computationally intractable.

The techniques presented in this lecture have found widespread applications across various domains, from training deep neural networks with millions of parameters to solving complex inference problems in statistical models with latent variables. By understanding the theoretical foundations and practical implementations of these methods, we gain powerful tools for addressing the computational challenges that arise in modern data science and artificial intelligence.

---

1 Stochastic Gradient Descent (SGD)

Gradient-based optimization methods form the backbone of many machine learning algorithms. While traditional gradient descent uses the entire dataset to compute gradients, stochastic variants use subsets of data, offering significant computational advantages for large-scale problems.

1.1 Deterministic Gradient Descent

For a differentiable objective function $f : \mathbb{R}^d \to \mathbb{R}$, gradient descent iteratively updates parameters in the direction of steepest descent:

Definition. Gradient Descent

The gradient descent algorithm iteratively updates parameters according to:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha_k \nabla f(\mathbf{x}_k)$$

where $\alpha_k > 0$ is the step size (learning rate) at iteration k, and $\nabla f(\mathbf{x}_k)$ is the gradient of f evaluated at $\mathbf{x}_k$.

Intuition: Gradient descent visualization

Imagine a hiker in a mountainous landscape trying to reach the lowest point (minimum). At each step, the hiker looks around to find the steepest downhill direction (the negative gradient) and takes a step in that direction. The size of the step is determined by the learning rate. If the steps are too large, the hiker might overshoot the valley; if too small, the journey takes unnecessarily long.

For convex functions with Lipschitz continuous gradients, gradient descent converges to the global minimum with an appropriate step size. The convergence rate depends on the properties of the objective function:

Theorem 1.1 (Convergence of Gradient Descent)**.** *For a convex function f with L-Lipschitz continuous gradients, gradient descent with constant step size $\alpha = \frac{1}{L}$ converges as:*

$$f(\mathbf{x}_k) - f(\mathbf{x}^*) \leq \frac{L\|\mathbf{x}_0 - \mathbf{x}^*\|^2}{2k}$$

where $\mathbf{x}^*$ is the global minimizer. For strongly convex functions with parameter $\mu > 0$, the convergence is linear:

$$f(\mathbf{x}_k) - f(\mathbf{x}^*) \leq \left(1 - \frac{\mu}{L}\right)^k [f(\mathbf{x}_0) - f(\mathbf{x}^*)]$$

Example 1.1 (Quadratic Function Optimization)**.** *Consider minimizing $f(x) = \frac{1}{2}x^2$. The gradient is $\nabla f(x) = x$. With a fixed step size $\alpha = 0.1$ and initial point $x_0 = 10$, the gradient descent iterations proceed as:*

$$\begin{aligned} x_1 &= 10 - 0.1 \times 10 = 9 \\ x_2 &= 9 - 0.1 \times 9 = 8.1 \\ x_3 &= 8.1 - 0.1 \times 8.1 = 7.29 \end{aligned}$$

The sequence converges to the global minimum at $x^* = 0$. Each iteration reduces the distance to the optimum by a factor of $(1 - \alpha) = 0.9$.

1.1.1 Newton's Method

For twice-differentiable functions, Newton's method incorporates second-order information to accelerate convergence:

Definition. Newton's Method

Newton's method updates parameters according to:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - [H_f(\mathbf{x}_k)]^{-1} \nabla f(\mathbf{x}_k)$$

where $H_f(\mathbf{x}_k)$ is the Hessian matrix of f evaluated at $\mathbf{x}_k$.

Intuition: Newton's method vs. gradient descent

While gradient descent only uses the slope (first derivative) to determine the direction of the next step, Newton's method also uses the curvature (second derivative) to adjust the step size. This is like a hiker who not only considers the steepness of the terrain but also how the steepness changes, allowing for larger steps in flat regions and smaller steps in highly curved areas.

Newton's method typically exhibits quadratic convergence near the optimum, making it much faster than gradient descent for well-behaved functions. However, computing and inverting the Hessian can be prohibitively expensive for high-dimensional problems.

1.2 Stochastic Gradient Methods

In many machine learning problems, the objective function has the form:

$$f(\mathbf{x}) = \frac{1}{n}\sum_{i=1}^{n} f_i(\mathbf{x})$$

where each f_i corresponds to the loss for a single data point. Computing the full gradient requires evaluating all n component gradients, which becomes expensive for large datasets.

Definition. Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) approximates the true gradient using a subset of the data:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha_k \tilde{\nabla} f(\mathbf{x}_k)$$

where $\tilde{\nabla} f(\mathbf{x}_k)$ is a stochastic approximation of the true gradient $\nabla f(\mathbf{x}_k)$.

There are several variants of stochastic gradient methods, each with different approaches to approximating the gradient:

1.2.1 Vanilla SGD

The simplest form of SGD uses a single randomly selected data point to approximate the gradient:

$$\tilde{\nabla} f(\mathbf{x}_k) = \nabla f_{i_k}(\mathbf{x}_k)$$

where i_k is randomly sampled from $\{1, 2, \ldots, n\}$ at each iteration.

Intuition: Why SGD works

SGD works because, in expectation, the stochastic gradient equals the true gradient: $\mathbb{E}[\tilde{\nabla} f(\mathbf{x})] = \nabla f(\mathbf{x})$. While each step is noisy, the overall trajectory moves toward the minimum. This is like a drunk hiker who takes random steps but generally moves downhill—the path is erratic, but the destination is eventually reached.

1.2.2 Mini-Batch SGD

Mini-batch SGD strikes a balance between the computational efficiency of vanilla SGD and the stability of full-batch gradient descent by using a small batch of data points:

$$\tilde{\nabla} f(\mathbf{x}_k) = \frac{1}{|B_k|}\sum_{i \in B_k} \nabla f_i(\mathbf{x}_k)$$

where $B_k \subset \{1, 2, \ldots, n\}$ is a randomly selected mini-batch of size $|B_k| \ll n$.

Remark.

The choice of batch size involves a trade-off: larger batches provide more accurate gradient estimates but require more computation per iteration. In practice, batch sizes of 32 to 512 are common, with larger batches enabling better parallelization on modern hardware.

1.2.3 SGD with Momentum

SGD with momentum incorporates information from past gradients to smooth the optimization trajectory:

$$\mathbf{v}_{k+1} = \beta \mathbf{v}_k + (1-\beta)\tilde{\nabla} f(\mathbf{x}_k) \tag{1}$$
$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha_k \mathbf{v}_{k+1} \tag{2}$$

where $\beta \in [0, 1)$ is the momentum parameter, typically set around 0.9.

Intuition: Momentum analogy

Momentum is like a ball rolling down a hill. The ball accumulates velocity (momentum) as it descends, helping it overcome small bumps and local minima. Similarly, momentum in optimization helps overcome small local variations in the gradient and accelerates progress along consistent directions.

1.2.4 Adaptive Learning Rate Methods

Several algorithms adaptively adjust the learning rate for each parameter based on the history of gradients:

AdaGrad adjusts the learning rate for each parameter inversely proportional to the square root of the sum of squared historical gradients:

$$\mathbf{G}_k = \mathbf{G}_{k-1} + \tilde{\nabla} f(\mathbf{x}_k) \odot \tilde{\nabla} f(\mathbf{x}_k) \tag{3}$$
$$\mathbf{x}_{k+1} = \mathbf{x}_k - \frac{\alpha}{\sqrt{\mathbf{G}_k + \epsilon}} \odot \tilde{\nabla} f(\mathbf{x}_k) \tag{4}$$

where $\odot$ denotes element-wise multiplication and ϵ is a small constant to avoid division by zero.

RMSProp modifies AdaGrad to use an exponentially weighted moving average of squared gradients:

$$\mathbf{G}_k = \beta \mathbf{G}_{k-1} + (1-\beta)\tilde{\nabla} f(\mathbf{x}_k) \odot \tilde{\nabla} f(\mathbf{x}_k) \tag{5}$$
$$\mathbf{x}_{k+1} = \mathbf{x}_k - \frac{\alpha}{\sqrt{\mathbf{G}_k + \epsilon}} \odot \tilde{\nabla} f(\mathbf{x}_k) \tag{6}$$

Adam combines momentum with adaptive learning rates:

$$\mathbf{m}_k = \beta_1 \mathbf{m}_{k-1} + (1-\beta_1)\tilde{\nabla} f(\mathbf{x}_k) \tag{7}$$
$$\mathbf{v}_k = \beta_2 \mathbf{v}_{k-1} + (1-\beta_2)\tilde{\nabla} f(\mathbf{x}_k) \odot \tilde{\nabla} f(\mathbf{x}_k) \tag{8}$$
$$\hat{\mathbf{m}}_k = \frac{\mathbf{m}_k}{1-\beta_1^k} \tag{9}$$
$$\hat{\mathbf{v}}_k = \frac{\mathbf{v}_k}{1-\beta_2^k} \tag{10}$$
$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha \frac{\hat{\mathbf{m}}_k}{\sqrt{\hat{\mathbf{v}}_k} + \epsilon} \tag{11}$$

where β_1 and β_2 are decay rates for the moment estimates, typically set to 0.9 and 0.999, respectively.

Remark.

Adam has become the default optimizer for many deep learning applications due to its robustness to hyperparameter choices and good performance across a wide range of problems. However, recent research has shown that SGD with momentum often generalizes better for some tasks, particularly in computer vision.

Algorithm 1 Adam Optimizer

Require: Learning rate α, decay rates $\beta_1, \beta_2 \in [0, 1)$, small constant ϵ

Initialize parameters $\mathbf{x}_0$, first moment vector $\mathbf{m}_0 \leftarrow \mathbf{0}$, second moment vector $\mathbf{v}_0 \leftarrow \mathbf{0}$, time step $t \leftarrow 0$
while not converged **do**
  $t \leftarrow t + 1$
  Compute stochastic gradient $\mathbf{g}_t \leftarrow \tilde{\nabla} f(\mathbf{x}_{t-1})$
  Update biased first moment estimate: $\mathbf{m}_t \leftarrow \beta_1 \mathbf{m}_{t-1} + (1 - \beta_1)\mathbf{g}_t$
  Update biased second moment estimate: $\mathbf{v}_t \leftarrow \beta_2 \mathbf{v}_{t-1} + (1 - \beta_2)\mathbf{g}_t^2$
  Correct bias in first moment: $\hat{\mathbf{m}}_t \leftarrow \mathbf{m}_t / (1 - \beta_1^t)$
  Correct bias in second moment: $\hat{\mathbf{v}}_t \leftarrow \mathbf{v}_t / (1 - \beta_2^t)$
  Update parameters: $\mathbf{x}_t \leftarrow \mathbf{x}_{t-1} - \alpha \cdot \hat{\mathbf{m}}_t / (\sqrt{\hat{\mathbf{v}}_t} + \epsilon)$
end while
return $\mathbf{x}_t$

1.3 Convergence Analysis of SGD

The convergence properties of SGD differ from those of deterministic gradient descent due to the noise introduced by stochastic gradients.

Theorem 1.2 (Convergence of SGD for Convex Functions)**.** *For a convex function f with L-Lipschitz continuous gradients, if the stochastic gradients satisfy $\mathbb{E}[\tilde{\nabla} f(\mathbf{x})] = \nabla f(\mathbf{x})$ and $\mathbb{E}[\|\tilde{\nabla} f(\mathbf{x}) - \nabla f(\mathbf{x})\|^2] \leq \sigma^2$, then SGD with step size $\alpha_k = \frac{\alpha}{\sqrt{k}}$ converges as:*

$$\mathbb{E}[f(\bar{\mathbf{x}}_K) - f(\mathbf{x}^*)] \leq \frac{L\|\mathbf{x}_0 - \mathbf{x}^*\|^2}{2\alpha K} + \frac{\alpha\sigma^2}{2\sqrt{K}}$$

where $\bar{\mathbf{x}}_K = \frac{1}{K}\sum_{k=1}^{K} \mathbf{x}_k$ is the average iterate.

This result highlights a key difference between SGD and deterministic gradient descent: SGD converges at a slower rate of $O(1/\sqrt{K})$ compared to $O(1/K)$ for gradient descent. This is the price paid for the computational efficiency of not computing the full gradient.

Practice Problems

1. Implement vanilla SGD, mini-batch SGD, and SGD with momentum for minimizing the function $f(x, y) = x^2 + 2y^2$. Compare their convergence rates and trajectories.
2. Derive the update rule for AdaGrad from first principles, starting with the idea of adapting the learning rate based on the historical gradient information.
3. Consider a linear regression problem with n data points. Compare the computational complexity per iteration and the number of iterations required for convergence for full-batch gradient descent, mini-batch SGD (with batch size b), and vanilla SGD.

Solution:

For problem 3: The computational complexity per iteration for full-batch gradient descent is $O(nd)$, where d is the dimension of the parameter vector. For mini-batch SGD, it's $O(bd)$, and for vanilla SGD, it's $O(d)$.

For strongly convex functions, full-batch gradient descent requires $O(\log(1/\epsilon))$ iterations to reach an ϵ-accurate solution, while SGD requires $O(1/\epsilon)$ iterations. Therefore, the total computational complexity is:

- Full-batch GD: $O(nd\log(1/\epsilon))$
- Mini-batch SGD: $O(bd/\epsilon)$
- Vanilla SGD: $O(d/\epsilon)$

For large datasets where n is very large, SGD and mini-batch SGD can be much more efficient despite requiring more iterations.

---

2 Distributed and Decentralized Optimization

As datasets grow and computational resources become more distributed, optimization algorithms that can leverage parallel and distributed computing have become increasingly important.

2.1 Distributed Optimization Framework

In distributed optimization, the objective function is typically decomposed across multiple computing nodes:

$$\min_{\mathbf{x}\in\mathbb{R}^d} f(\mathbf{x}) = \min_{\mathbf{x}\in\mathbb{R}^d} \frac{1}{M}\sum_{m=1}^{M} F_m(\mathbf{x})$$

where F_m represents the objective function component associated with node m.

Definition. Distributed Gradient Descent

In distributed gradient descent, each node computes a local gradient based on its data, and these gradients are aggregated to update the global model:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha_k \frac{1}{M}\sum_{m=1}^{M} \tilde{\nabla} F_m(\mathbf{x}_k)$$

where $\tilde{\nabla} F_m(\mathbf{x}_k)$ is the stochastic gradient computed at node m.

Intuition: Distributed optimization

Distributed optimization is like a team of hikers exploring different parts of a mountain range, each reporting back the steepest descent direction in their area. A central coordinator combines these reports to determine the overall best direction for the team to move. This parallel exploration allows covering more ground efficiently.

2.2 Decentralized Optimization

In decentralized optimization, there is no central coordinator. Instead, nodes communicate only with their neighbors in a network.

Definition. Decentralized Gradient Descent

In decentralized gradient descent, each node updates its local model by combining information from its neighbors and its local gradient:

$$\mathbf{x}_m^{(k+1)} = \sum_{n\in\mathcal{N}(m)} w_{mn}\mathbf{x}_n^{(k)} - \alpha_k \tilde{\nabla} F_m(\mathbf{x}_m^{(k)})$$

where $\mathcal{N}(m)$ is the set of neighbors of node m, and w_{mn} are mixing weights that satisfy $\sum_{n\in\mathcal{N}(m)} w_{mn} = 1$ for all m.

The mixing weights are typically derived from the network topology and are often represented as a matrix W with entries w_{mn}. The convergence of decentralized algorithms depends on the spectral properties of this matrix.

Theorem 2.1 (Convergence of Decentralized Gradient Descent)**.** *For a convex function f with L-Lipschitz continuous gradients, if the mixing matrix W is doubly stochastic and the network is connected, then decentralized gradient descent converges to the global optimum at a rate of $O(1/k)$.*

2.3 Communication-Efficient Distributed Optimization

Communication between nodes can become a bottleneck in distributed optimization. Several techniques have been developed to reduce communication overhead:

2.3.1 Quantization and Sparsification

Gradient quantization reduces the precision of gradient values to decrease communication bandwidth:

$$Q(\nabla f(\mathbf{x})) = \text{sign}(\nabla f(\mathbf{x})) \cdot \|\nabla f(\mathbf{x})\|_1/d$$

Gradient sparsification communicates only the largest components of the gradient:

$$S_k(\nabla f(\mathbf{x})) = \text{top-}k(\nabla f(\mathbf{x}))$$

where top-k selects the k components with the largest magnitude.

2.3.2 Local SGD

In Local SGD, nodes perform multiple local updates before communicating:

$$\mathbf{x}_{m,t+\tau}^{(k)} = \mathbf{x}_{m,t}^{(k)} - \alpha \sum_{j=0}^{\tau-1} \tilde{\nabla} F_m(\mathbf{x}_{m,t+j}^{(k)}) \quad \text{(local updates)} \tag{12}$$

$$\mathbf{x}_{m,t+\tau}^{(k+1)} = \frac{1}{M} \sum_{m=1}^{M} \mathbf{x}_{m,t+\tau}^{(k)} \quad \text{(synchronization)} \tag{13}$$

Remark.

The choice of synchronization period τ involves a trade-off: larger values reduce communication frequency but may lead to divergence between local models, especially in heterogeneous data settings.

2.4 Federated Learning

Federated learning is a special case of distributed optimization where data remains on local devices (e.g., mobile phones), and only model updates are communicated to a central server.

Definition. **Federated Averaging**

Federated Averaging (FedAvg) is an algorithm for federated learning that combines local SGD with weighted averaging:

$$\mathbf{x}_{m,t+\tau}^{(k)} = \mathbf{x}_{m,t}^{(k)} - \alpha \sum_{j=0}^{\tau-1} \tilde{\nabla} F_m(\mathbf{x}_{m,t+j}^{(k)}) \quad \text{(local updates)} \tag{14}$$

$$\mathbf{x}^{(k+1)} = \sum_{m=1}^{M} \frac{n_m}{n} \mathbf{x}_{m,t+\tau}^{(k)} \quad \text{(weighted averaging)} \tag{15}$$

Algorithm 2 Local SGD with Periodic Averaging

Require: Initial model $\mathbf{x}_0$, learning rate α, synchronization period τ, number of nodes M

Distribute $\mathbf{x}_0$ to all nodes
for $k = 0, 1, 2, \ldots$ **do**
  for each node $m = 1, 2, \ldots, M$ in parallel **do**
    $\mathbf{x}_{m,0}^{(k)} \leftarrow \mathbf{x}^{(k)}$ ▷ Initialize local model
    for $t = 0, 1, \ldots, \tau - 1$ **do**
      Sample mini-batch $B_{m,t}$ from local data
      Compute stochastic gradient $\mathbf{g}_{m,t} \leftarrow \nabla F_m(\mathbf{x}_{m,t}^{(k)}; B_{m,t})$
      Update local model: $\mathbf{x}_{m,t+1}^{(k)} \leftarrow \mathbf{x}_{m,t}^{(k)} - \alpha \mathbf{g}_{m,t}$
    end for
  end for
  $\mathbf{x}^{(k+1)} \leftarrow \frac{1}{M} \sum_{m=1}^{M} \mathbf{x}_{m,\tau}^{(k)}$ ▷ Average models across nodes
  Distribute $\mathbf{x}^{(k+1)}$ to all nodes
end for

where n_m is the number of data points at node m, and $n = \sum_{m=1}^{M} n_m$ is the total number of data points.

Federated learning faces unique challenges, including:

- Non-IID data: Local datasets may have very different distributions
- System heterogeneity: Devices have varying computational capabilities
- Communication constraints: Limited bandwidth and intermittent connectivity
- Privacy concerns: Sensitive data must remain on local devices

Practice Problems

1. Consider a network of 5 nodes arranged in a ring topology. Design a doubly stochastic mixing matrix W for this network and analyze its spectral properties.
2. Compare the convergence rates of centralized SGD, distributed SGD with full gradient communication, and Local SGD with synchronization period $\tau = 10$ for a strongly convex objective function.
3. In federated learning, how does the heterogeneity of data across clients affect the convergence of FedAvg? Propose a modification to the algorithm to address this challenge.

---

3 Expectation-Maximization (EM) Algorithm

The Expectation-Maximization (EM) algorithm is a powerful method for finding maximum likelihood estimates in models with latent variables or missing data.

3.1 Problem Formulation

Consider a probabilistic model with observed data X, latent variables Z, and parameters θ. The likelihood function is:

$$L(\theta; X) = p(X|\theta) = \int p(X, Z|\theta)\, dZ$$

Direct maximization of this likelihood can be difficult due to the integral over latent variables. The EM algorithm provides an iterative approach to this problem.

Definition. EM Algorithm

The EM algorithm alternates between two steps:

- **E-step**: Compute the expected log-likelihood with respect to the conditional distribution of latent variables given the current parameter estimate:

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

- **M-step**: Find the parameters that maximize this expected log-likelihood:

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

Intuition: EM algorithm

The EM algorithm can be understood as a coordinate ascent method on a lower bound of the log-likelihood. In the E-step, we compute the tightest lower bound given the current parameters. In the M-step, we maximize this bound with respect to the parameters. This process guarantees that the likelihood increases with each iteration.

3.2 Theoretical Properties

The EM algorithm has several important theoretical properties:

Theorem 3.1 (Monotonicity of EM)**.** *The EM algorithm guarantees that the likelihood increases with each iteration:*

$$L(\theta^{(t+1)}; X) \geq L(\theta^{(t)}; X)$$

Theorem 3.2 (Convergence of EM)**.** *Under mild regularity conditions, the EM algorithm converges to a stationary point of the likelihood function, which may be a local maximum or a saddle point.*

Remark.

While the EM algorithm is guaranteed to increase the likelihood with each iteration, it may converge to a local maximum rather than the global maximum. Multiple random initializations are often used to mitigate this issue.

3.3 Examples and Applications

3.3.1 Gaussian Mixture Models

A Gaussian Mixture Model (GMM) represents a probability distribution as a weighted sum of Gaussian components:

$$p(x|\theta) = \sum_{k=1}^{K} \pi_k \mathcal{N}(x|\mu_k, \Sigma_k)$$

where $\theta = \{\pi_k, \mu_k, \Sigma_k\}_{k=1}^{K}$ are the mixture weights, means, and covariances.

For a dataset $X = \{x_1, x_2, \ldots, x_n\}$, the EM algorithm for GMMs proceeds as follows:

Algorithm 3 EM Algorithm for Gaussian Mixture Models

Require: Data $X = \{x_1, x_2, \ldots, x_n\}$, number of components K

Initialize parameters $\theta^{(0)} = \{\pi_k^{(0)}, \mu_k^{(0)}, \Sigma_k^{(0)}\}_{k=1}^K$
while not converged **do**
  E-step: Compute responsibilities
  for $i = 1, 2, \ldots, n$ and $k = 1, 2, \ldots, K$ **do**
    $\gamma_{ik} \leftarrow \frac{\pi_k^{(t)} \mathcal{N}(x_i | \mu_k^{(t)}, \Sigma_k^{(t)})}{\sum_{j=1}^K \pi_j^{(t)} \mathcal{N}(x_i | \mu_j^{(t)}, \Sigma_j^{(t)})}$
  end for
  M-step: Update parameters
  for $k = 1, 2, \ldots, K$ **do**
    $N_k \leftarrow \sum_{i=1}^n \gamma_{ik}$
    $\pi_k^{(t+1)} \leftarrow \frac{N_k}{n}$
    $\mu_k^{(t+1)} \leftarrow \frac{1}{N_k} \sum_{i=1}^n \gamma_{ik} x_i$
    $\Sigma_k^{(t+1)} \leftarrow \frac{1}{N_k} \sum_{i=1}^n \gamma_{ik} (x_i - \mu_k^{(t+1)})(x_i - \mu_k^{(t+1)})^T$
  end for
  $t \leftarrow t + 1$
end while
return $\theta^{(t)} = \{\pi_k^{(t)}, \mu_k^{(t)}, \Sigma_k^{(t)}\}_{k=1}^K$

Example 3.1 (GMM for Clustering)**.** *Consider a dataset of customer purchase patterns. By fitting a GMM with $K = 3$ components, we can identify three distinct customer segments. The responsibilities γ_{ik} represent the probability that customer i belongs to segment k, providing a soft clustering of customers. The means μ_k represent the typical purchase pattern for each segment, while the covariances Σ_k capture the variability within each segment.*

3.3.2 Hidden Markov Models

Hidden Markov Models (HMMs) are used for sequential data with latent state sequences. The EM algorithm for HMMs is known as the Baum-Welch algorithm and uses the forward-backward algorithm for the E-step.

3.3.3 Factor Analysis

Factor Analysis models high-dimensional data as being generated from a lower-dimensional set of latent factors. The EM algorithm provides an efficient way to estimate the factor loadings and noise variances.

3.4 Variants and Extensions

3.4.1 Monte Carlo EM (MCEM)

When the E-step involves intractable expectations, Monte Carlo methods can be used to approximate them:

$$Q(\theta|\theta^{(t)}) \approx \frac{1}{S} \sum_{s=1}^{S} \log p(X, Z^{(s)}|\theta)$$

where $Z^{(s)} \sim p(Z|X, \theta^{(t)})$ are samples from the conditional distribution of latent variables.

3.4.2 Stochastic EM

Stochastic EM replaces the expectation in the E-step with a single sample:

$$Z^{(t)} \sim p(Z|X, \theta^{(t)}) \quad \text{(S-step)} \tag{16}$$

$$\theta^{(t+1)} = \arg\max_{\theta} \log p(X, Z^{(t)}|\theta) \quad \text{(M-step)} \tag{17}$$

3.4.3 Generalized EM

In some cases, the M-step may be difficult to solve exactly. Generalized EM (GEM) relaxes the requirement to find the global maximum in the M-step, requiring only that $Q(\theta^{(t+1)}|\theta^{(t)}) > Q(\theta^{(t)}|\theta^{(t)})$.

Practice Problems

1. Derive the EM update equations for a mixture of two univariate Gaussian distributions with unknown means but fixed variances.
2. Implement the EM algorithm for a Gaussian Mixture Model with $K = 2$ components on a synthetic dataset. Visualize the evolution of the model parameters and the log-likelihood across iterations.
3. Consider a coin-flipping experiment where we have two biased coins with unknown probabilities of heads θ_1 and θ_2. We randomly select one of the coins for each of n flips, but we don't know which coin was used for each flip. Formulate this as a latent variable problem and derive the EM algorithm to estimate θ_1 and θ_2.

Solution:

For problem 3: Let $X_i \in \{0, 1\}$ be the outcome of the i-th flip (0 for tails, 1 for heads), and let $Z_i \in \{1, 2\}$ be the latent variable indicating which coin was used.

The complete data likelihood is:

$$p(X, Z|\theta) = \prod_{i=1}^{n} p(Z_i)p(X_i|Z_i, \theta)$$

Assuming equal probability of selecting either coin, $p(Z_i = 1) = p(Z_i = 2) = 0.5$, and $p(X_i = 1|Z_i = j, \theta) = \theta_j$.

In the E-step, we compute:

$$\gamma_{ij} = p(Z_i = j|X_i, \theta^{(t)}) = \frac{p(Z_i = j)p(X_i|Z_i = j, \theta^{(t)})}{p(X_i|\theta^{(t)})}$$

For $X_i = 1$:

$$\gamma_{i1} = \frac{0.5 \cdot \theta_1^{(t)}}{0.5 \cdot \theta_1^{(t)} + 0.5 \cdot \theta_2^{(t)}} = \frac{\theta_1^{(t)}}{\theta_1^{(t)} + \theta_2^{(t)}}$$

In the M-step, we update:

$$\theta_j^{(t+1)} = \frac{\sum_{i:X_i=1} \gamma_{ij}}{\sum_{i=1}^{n} \gamma_{ij}}$$

4 Markov Chain Monte Carlo (MCMC) for Optimization

While Markov Chain Monte Carlo (MCMC) methods are primarily used for sampling from complex probability distributions, they can also be adapted for optimization tasks.

4.1 Simulated Annealing Revisited

Simulated annealing, which we studied in Week 11, can be viewed as an MCMC method for optimization. It uses the Metropolis-Hastings algorithm to sample from a sequence of distributions that gradually concentrate on the global optimum.

For a function $f(x)$ that we want to minimize, we define a Boltzmann distribution:

$$p_\beta(x) \propto e^{-\beta f(x)}$$

where $\beta > 0$ is an inverse temperature parameter.

As $\beta \to \infty$, the distribution concentrates on the global minimum of $f(x)$. Simulated annealing gradually increases β while sampling from $p_\beta(x)$ using MCMC methods.

4.2 Stochastic Gradient Langevin Dynamics (SGLD)

SGLD combines SGD with Langevin dynamics to perform approximate Bayesian inference:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha_k \tilde{\nabla} f(\mathbf{x}_k) + \sqrt{2\alpha_k}\, \mathbf{z}_k$$

where $\mathbf{z}_k \sim \mathcal{N}(0, I)$ is standard Gaussian noise.

Intuition: SGLD

SGLD can be viewed as SGD with added noise. The noise helps escape local minima and explore the parameter space more thoroughly. As the learning rate decreases, the algorithm transitions from exploration to exploitation, eventually converging to a local minimum.

For optimization purposes, we can use a variant called preconditioned SGLD:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha_k G(\mathbf{x}_k)^{-1} \tilde{\nabla} f(\mathbf{x}_k) + \sqrt{2\alpha_k}\, G(\mathbf{x}_k)^{-1/2} \mathbf{z}_k$$

where $G(\mathbf{x}_k)$ is a preconditioning matrix that adapts to the local geometry of the objective function.

4.3 Parallel Tempering

Parallel tempering runs multiple MCMC chains at different temperatures and occasionally swaps states between chains:

Remark.

Parallel tempering is particularly effective for multi-modal objective functions, as the lower temperature chains can explore the space more freely, while the higher temperature chains focus on promising regions.

4.4 Evolutionary MCMC

Evolutionary MCMC combines ideas from evolutionary algorithms with MCMC methods. It maintains a population of states and uses genetic operators (crossover, mutation) to propose new states, which are accepted or rejected based on Metropolis-Hastings criteria.

Algorithm 4 Parallel Tempering for Optimization

Require: Objective function $f(x)$, number of chains M, temperatures $\beta_1 < \beta_2 < \ldots < \beta_M$

Initialize states $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_M$
for $t = 1, 2, \ldots$ **do**
  for $m = 1, 2, \ldots, M$ **do**
    Update $\mathbf{x}_m$ using MCMC targeting $p_{\beta_m}(x) \propto e^{-\beta_m f(x)}$
  end for
  for $m = 1, 2, \ldots, M-1$ **do**
    Compute swap probability:
    $\alpha = \min\{1, \exp((\beta_m - \beta_{m+1})(f(\mathbf{x}_m) - f(\mathbf{x}_{m+1})))\}$
    With probability α, swap $\mathbf{x}_m$ and $\mathbf{x}_{m+1}$
  end for
end for
return $\mathbf{x}_M$ (state from the highest temperature chain)

Practice Problems

1. Implement simulated annealing and SGLD for minimizing a multi-modal function such as the Rastrigin function. Compare their performance in terms of finding the global minimum.
2. Derive the acceptance probability for state swaps in parallel tempering from the detailed balance condition.
3. Design an evolutionary MCMC algorithm for a combinatorial optimization problem, such as the traveling salesman problem. Specify the state representation, genetic operators, and acceptance criteria.

5 No Free Lunch Theorems

The No Free Lunch (NFL) theorems provide fundamental limits on the performance of optimization algorithms across all possible problems.

Theorem 5.1 (No Free Lunch Theorem for Optimization)**.** *When averaged over all possible objective functions, every optimization algorithm has the same expected performance. In other words, no algorithm is universally better than any other algorithm for all possible problems.*

Intuition: NFL intuition

The NFL theorem can be understood through a simple analogy: If you're searching for a specific book in a library where books are arranged randomly, no search strategy is better than any other in the long run. You might get lucky with a particular strategy for a specific arrangement, but across all possible arrangements, all strategies perform equally well on average.

5.1 Implications and Practical Considerations

The NFL theorems have several important implications:

- Algorithm selection should be based on problem-specific knowledge

- Performance guarantees must be tied to specific problem classes
- Hyper-parameter tuning is essential for good performance
- Domain expertise is valuable for designing effective algorithms

Remark.

While the NFL theorems state that no algorithm is universally superior, in practice, we often work with specific problem classes that have structure (e.g., convexity, smoothness). Within these classes, some algorithms can indeed outperform others consistently.

5.2 Beyond NFL: Algorithm Portfolios

Given the NFL theorems, one approach is to use algorithm portfolios that combine multiple optimization methods:

- Run multiple algorithms in parallel and select the best result
- Use meta-learning to predict which algorithm will perform best for a given problem
- Develop hybrid algorithms that adapt their behavior based on the observed performance

Practice Problems

1. Provide a formal proof sketch of the NFL theorem for optimization, starting from the assumption that all objective functions are equally likely.
2. Design an algorithm portfolio for a class of optimization problems, such as continuous black-box optimization. Specify the algorithms in the portfolio and the strategy for selecting or combining their results.
3. Discuss how the NFL theorems relate to the concept of "inductive bias" in machine learning. How can we reconcile the success of specific algorithms in practice with the theoretical limitations imposed by the NFL theorems?

---

6 Some comments

In this lecture, we have explored advanced stochastic optimization techniques that are essential for solving large-scale and complex optimization problems. We began with stochastic gradient methods, which form the backbone of modern machine learning algorithms, and examined their variants and convergence properties. We then discussed distributed and decentralized optimization frameworks that enable leveraging parallel computing resources while managing communication constraints.

The Expectation-Maximization algorithm provided a powerful approach for maximum likelihood estimation in models with latent variables, with applications ranging from mixture models to hidden Markov models. We also explored how Markov Chain Monte Carlo methods, primarily designed for sampling, can be adapted for optimization tasks through techniques like simulated annealing and stochastic gradient Langevin dynamics.

Finally, the No Free Lunch theorems reminded us of the fundamental limitations of optimization algorithms and the importance of matching algorithms to specific problem classes.

References

1. Bottou, L., Curtis, F. E., & Nocedal, J. (2018). Optimization methods for large-scale machine learning. SIAM Review, 60(2), 223-311.

2. Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980.

3. Lian, X., Zhang, C., Zhang, H., Hsieh, C. J., Zhang, W., & Liu, J. (2017). Can decentralized algorithms outperform centralized algorithms? A case study for decentralized parallel stochastic gradient descent. In Advances in Neural Information Processing Systems.

4. McMahan, H. B., Moore, E., Ramage, D., Hampson, S., & y Arcas, B. A. (2017). Communication-efficient learning of deep networks from decentralized data. In Artificial Intelligence and Statistics.

5. Dempster, A. P., Laird, N. M., & Rubin, D. B. (1977). Maximum likelihood from incomplete data via the EM algorithm. Journal of the Royal Statistical Society: Series B, 39(1), 1-22.

6. Welling, M., & Teh, Y. W. (2011). Bayesian learning via stochastic gradient Langevin dynamics. In Proceedings of the 28th International Conference on Machine Learning.

7. Wolpert, D. H., & Macready, W. G. (1997). No free lunch theorems for optimization. IEEE Transactions on Evolutionary Computation, 1(1), 67-82.

8. Ruder, S. (2016). An overview of gradient descent optimization algorithms. arXiv preprint arXiv:1609.04747.

9. Neal, R. M. (2001). Annealed importance sampling. Statistics and Computing, 11(2), 125-139.

10. Nedic, A., & Ozdaglar, A. (2009). Distributed subgradient methods for multi-agent optimization. IEEE Transactions on Automatic Control, 54(1), 48-61.