Search - optimisation in continuous space

March 2024

Topics to be discussed

Part I. Introduction

1. Introduction to Artificial intelligence

Part II. Search and optimisation.

2. Searching and its heuristics
3. Search - optimisation
4. Two-player deterministic games
5. Evolutionary and genetic algorithms

Part III. Machine learning and data analysis.

6. Regression, classification and clustering (Part I & II)
8. Artificial neural networks
9. Bayesian models
10. Reinforcement Learning

Part IV. Logic, Inference, Knowledge Representation

11. Propositional logic and predicate logic
12. Knowledge Representation

Part V. AI in Action: Language, Vision

13. AI in Natural language processing
14. AI in Vision and Perception

Part VI. Summary

15. AI engineering, Explainable AI, Ethics

AI Searching Methods in Continuous Space

Overview

- Introduction optimisation methods in AI
- Applications in AI
- "Classic" Local Optimization Methods
  - Newton's Method
  - Levenberg Method
  - Gradient Descent Method
  - Stochastic Gradient Descent Method
  - Mini-batch Gradient Descent Method
- Comparison of Methods

Introduction to optimization methods in AI

Discrete optimization: find the best discrete object

$$\min_{p \in \text{Paths}} \text{Cost}(p)$$

Example of a method: dynamic programming

Continuous optimization: find the best vector of real numbers

$$\min_{w \in \mathbb{R}^d} \text{TrainingError}(w)$$

Example of a method: gradient descent

AI Searching Methods in Continuous Space

- Local optimization methods are used to find the minimum or maximum of a function within a specific region of its domain.
- These methods are often used in AI and machine learning to optimize model parameters.
- We will discuss three local optimization methods for continuous spaces: Newton's method, Levenberg-Marquardt method, and gradient descent (including stochastic gradient descent).

Optimization Problems in AI

Definition: An optimization problem seeks to minimize or maximize an objective function $f(\mathbf{x})$ subject to some constraints.

Basic Notions:

- $\mathbf{x} \in \mathbb{R}^n$ is the vector of optimization variables. These variables can be thought of as the parameters that are to be adjusted to optimize the objective function. The dimension of the vector $\mathbf{x}$ is denoted by n.
- $f(\mathbf{x})$ is the objective function to be minimized or maximized. It takes as input the vector $\mathbf{x}$ and returns a scalar value, which is a measure of how well the system is performing. The goal of the optimization problem is to find the values of $\mathbf{x}$ that minimize or maximize $f(\mathbf{x})$.
- $\mathbf{g}(\mathbf{x}) \leq \mathbf{0}$ and $\mathbf{h}(\mathbf{x}) = \mathbf{0}$ are the inequality and equality constraints. They specify the conditions that the variables $\mathbf{x}$ must satisfy.
- $\mathbf{x}^*$ is a solution to the optimization problem if it satisfies the constraints and minimizes or maximizes $f(\mathbf{x})$.
- A solution $\mathbf{x}^*$ is said to be optimal if there is no other feasible solution that has a better objective function value than $\mathbf{x}^*$.

Examples of Applications in AI

- **Computer Vision:** Image processing tasks, such as image segmentation, object detection, and image recognition.
- **Natural Language Processing:** Optimization of language models for tasks such as text classification, sentiment analysis, and machine translation.
- **Robotics:** Optimization of the motion planning of robots, as well as to optimize control policies for robots to perform specific tasks.
- **Recommendation Systems:** Optimization recommendation algorithms, such as collaborative filtering and content-based filtering.
- **Game AI:** Optimization of the behavior of agents in games, such as the movement and decision-making of non-player characters.

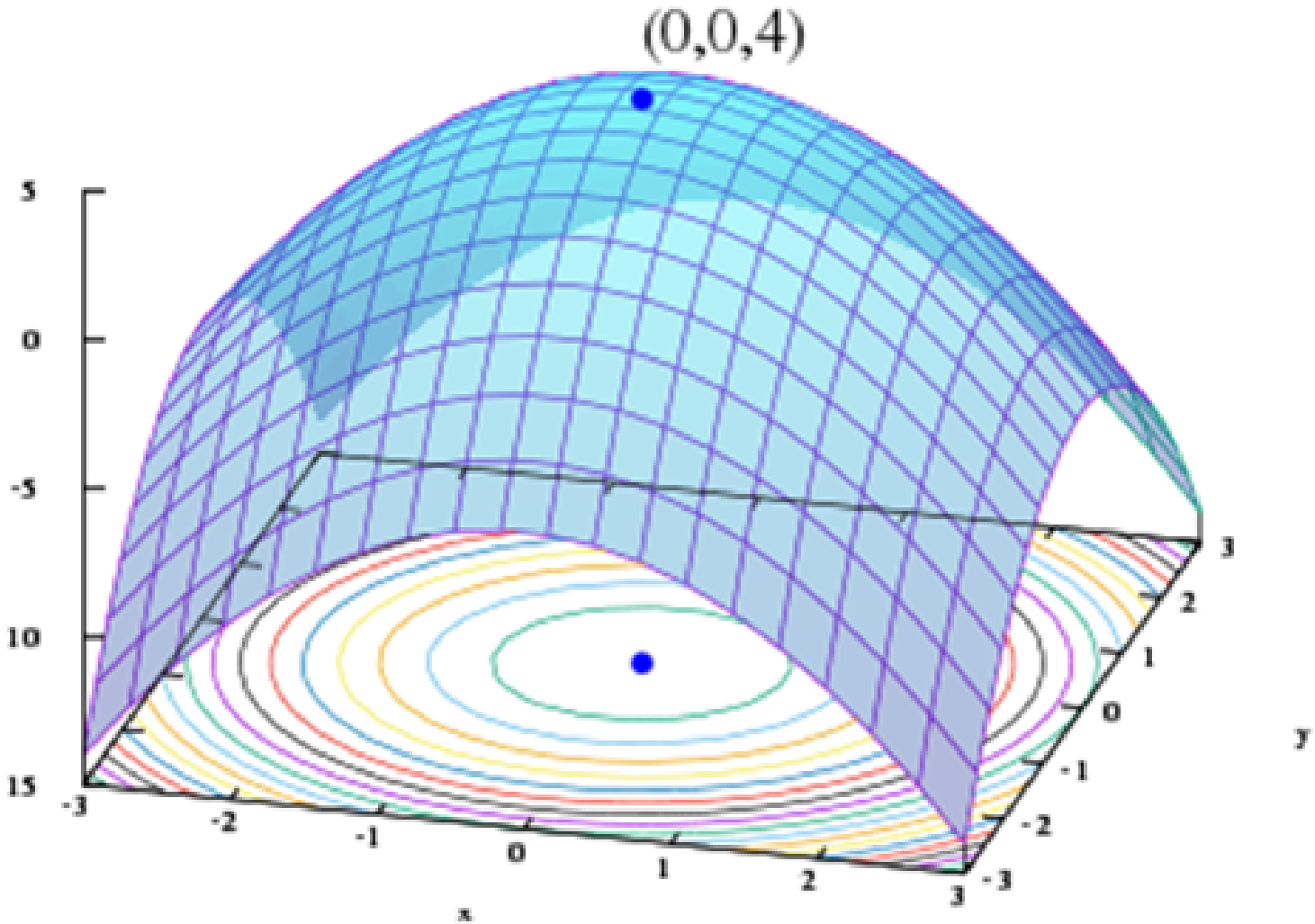


Figure: Simple convex function

Newton's Method

- Newton's Method is a local optimization algorithm used to find the minimum of a function.
- It uses second-order derivative information to quickly converge to the minimum.
- The method works by approximating the function at each iteration with a quadratic Taylor polynomial, and solving for its minimum.

Formula

The update rule for Newton's Method is:

$$x_{k+1} = x_k - \frac{f'(x_k)}{f''(x_k)}$$

where $f'(x_k)$ and $f''(x_k)$ denote the first and second derivative of the function f evaluated at x_k, respectively.

Taylor Polynomial - reminder

Definition

A Taylor polynomial is an approximation of a function by a polynomial around a specific point in its domain.

- The Taylor polynomial of degree n for a function $f(x)$ centered at a point a is given by:

$$P_n(x) = f(a)+f'(a)(x-a)+\frac{f''(a)}{2!}(x-a)^2+\frac{f'''(a)}{3!}(x-a)^3+\ldots+\frac{f^{(n)}(a)}{n!}(x-a)^n$$

- Where $f^{(k)}(a)$ denotes the k^{th} derivative of $f(x)$ evaluated at $x = a$.
- The polynomial approximates the behavior of the function near the point a.
- The accuracy of the approximation depends on the degree of the polynomial and how close the point a is to the point at which we want to approximate the function.

Newton-Raphson's Method: Example

Let's consider the function $f(x) = x^2 - 2$ and use Newton-Raphson's method to find its positive root with an initial guess of $x_0 = 1$.

1. Then: $f(x_0) = 1^2 - 2 = -1$, $f'(x_0) = 2x_0 = 2$
2. The linear approximation of $f(x)$ around x_0 is:

$$f(x) \approx -1 + 2(x - 1)$$

3. The root of the linear approximation is:

$$-1 + 2(x_1 - 1) = 0 \quad \Rightarrow \quad x_1 = \frac{3}{2}$$

4. Repeat the process with x_1 as the new initial guess:
5. $f(x_1) = \frac{3^2}{2^2} - 2 = \frac{1}{4}$
6. $f'(x_1) = 2x_1 = 3$
7. The root of the linear approximation is:

$$\frac{1}{4} + 3(x_2 - \frac{3}{2}) = 0 \quad \Rightarrow \quad x_2 \approx 1.414$$

Newton-Raphson's Method (cont.)

Let $f(x)$ be a differentiable function and let x_0 be an initial guess for the root of $f(x)$. The Newton-Raphson's method algorithm is as follows:

1. Calculate the function value and derivative at x_0: $f(x_0)$ and $f'(x_0)$.
2. Construct a linear approximation of $f(x)$ around x_0 using the first two terms of its Taylor series expansion:

$$f(x) \approx f(x_0) + f'(x_0)(x - x_0)$$

3. Find the root of the linear approximation by setting it equal to zero and solving for x:

$$f(x_0) + f'(x_0)(x - x_0) = 0 \quad \Rightarrow \quad x = x_0 - \frac{f(x_0)}{f'(x_0)}$$

4. Repeat steps 1-3 with $x_1 = x$ as the new initial guess until a sufficiently accurate root is found.

Why Newton's Method Works

Now, suppose we want to minimize a function $f(x)$ instead of finding its root.

We can use a second-order Taylor series expansion of $f(x)$ around x_0 to get:

$$f(x) \approx f(x_0) + f'(x_0)(x - x_0) + \frac{1}{2} f''(x_0)(x - x_0)^2$$

To find the critical point where $f'(x) = 0$ we take the derivative of $f(x)$ with respect to x and set it equal to zero:

$$f'(x_0) + f''(x_0)(x - x_0) = 0 \quad \Rightarrow \quad x = x_0 - \frac{f'(x_0)}{f''(x_0)}$$

This is the update rule for Newton's Method when we are finding the minimum of a function.

Newton's Method pseudo-code

Algorithm Newton's Method

Input: Function $f(x)$, starting point x_0, tolerance ϵ, maximum iterations N
Output: Optimal point x^*

$k \leftarrow 0$ **while** $k \leq N$ **do**
  $f'(x_k) \leftarrow$ first derivative of $f(x)$ evaluated at x_k
  $f''(x_k) \leftarrow$ second derivative of $f(x)$ evaluated at x_k
  if $f''(x_k) \neq 0$ **then**
    $x_{k+1} \leftarrow x_k - \frac{f'(x_k)}{f''(x_k)}$
  else
    return x_k
  if $|f(x_{k+1}) - f(x_k)| < \epsilon$ **then**
    return x_{k+1}
  else
    $k \leftarrow k + 1$
return x_{k+1}

Newton's Method

- Newton's method is an optimization algorithm that uses second-order information to converge to a local minimum of a function.
- It works by iteratively updating the current guess for the optimal solution based on the gradient and Hessian matrix of the function.
- The update rule for Newton's method is:

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

  where $H_f(x_k)$ is the Hessian matrix of the function f evaluated at the current guess x_k, $\nabla f(x_k)$ is the gradient of f evaluated at x_k, and α_k is a step size parameter.
- Newton's method is guaranteed to converge to a local minimum of f if f is twice differentiable and the initial guess is sufficiently close to a local minimum.

The Hessian Matrix

The Hessian matrix, denoted $H_f(x_k)$, is the square matrix of second-order partial derivatives of a function f evaluated at point x_k.

Definition

Given a function $f(x_1, x_2, \ldots, x_n)$ with continuous second-order partial derivatives at $x = (x_1, x_2, \ldots, x_n)$, the Hessian matrix $H_f(x)$ is defined as:

$$H_f(x) = \begin{pmatrix} \frac{\partial^2 f}{\partial x_1^2}(x) & \frac{\partial^2 f}{\partial x_1 \partial x_2}(x) & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n}(x) \\ \frac{\partial^2 f}{\partial x_2 \partial x_1}(x) & \frac{\partial^2 f}{\partial x_2^2}(x) & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n}(x) \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1}(x) & \frac{\partial^2 f}{\partial x_n \partial x_2}(x) & \cdots & \frac{\partial^2 f}{\partial x_n^2}(x) \end{pmatrix}$$

The Hessian matrix provides information about the local curvature of the function f at point x_k.

Newton Optimization Method with Hessian Matrix

Algorithm Newton Optimization Method

Input : $f(x, y)$, starting point (x_0, y_0), tolerance ϵ
Output: Optimal point (x^*, y^*)
while $||\nabla f(x_k, y_k)||_2 > \epsilon$ **do**
    Calculate $H_f(x_k, y_k)^{-1}$
    Calculate search direction $d_k = -H_f(x_k, y_k)^{-1} \nabla f(x_k, y_k)$
    Update $(x_{k+1}, y_{k+1}) = (x_k, y_k) + \alpha_k (d_{x,k}, d_{y,k})$, where α_k is found by line search

Example: $f(x, y) = x^2 + y^2$, $\nabla f(x, y) = (2x, 2y)$, $H_f(x, y) = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}$

Calculation: $H_f(x_k, y_k)^{-1} = \begin{bmatrix} 1/2 & 0 \\ 0 & 1/2 \end{bmatrix}$, $d_k = -\begin{bmatrix} 1/2 & 0 \\ 0 & 1/2 \end{bmatrix} \begin{bmatrix} 2x_k \\ 2y_k \end{bmatrix} = \begin{bmatrix} -x_k \\ -y_k \end{bmatrix}$

Levenberg Method for Local Optimization

The Levenberg method is an algorithm for finding the local minimum of a function $f(x)$ by adjusting a parameter λ that controls the tradeoff between the gradient descent step and the Newton step:

- If the gradient descent step overshoots the minimum, the Levenberg method dampens the step by increasing λ.
- If the Newton step is ill-conditioned or does not significantly improve the estimate, the Levenberg method increases the step by decreasing λ.

The Levenberg method starts with an initial guess x_0 and iteratively updates the estimate using the formula:

$$x_{k+1} = x_k - [H_f(x_k) + \lambda I]^{-1} \nabla f(x_k)$$

where $H_f(x_k)$ is the Hessian matrix of $f(x)$ evaluated at x_k, I is the identity matrix, and λ is the damping parameter.

Gradient Descent Algorithm

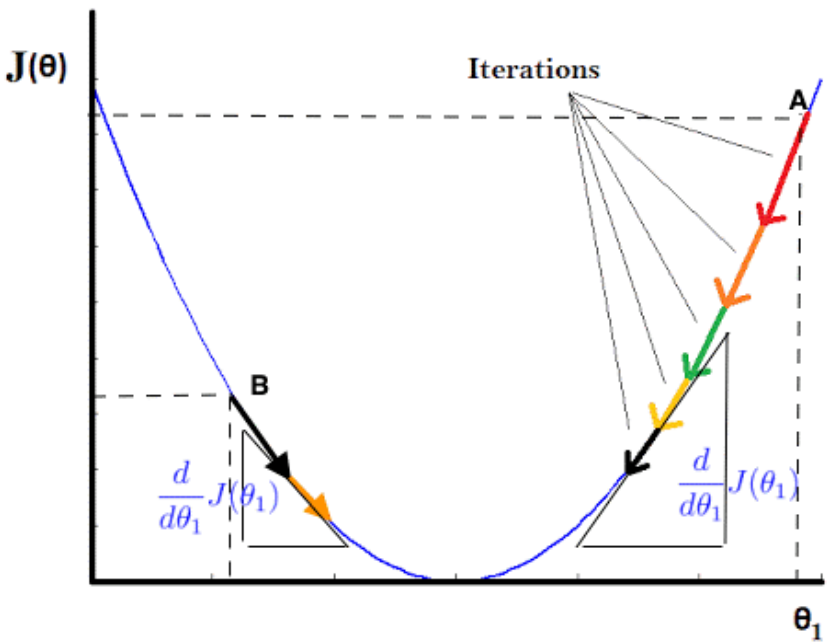


Figure: Gradient descent example

Consider the following function:

$$f(x) = x^2$$

We want to find its minimum using gradient descent.

- Start at some random point x_0
- Compute the gradient of the function at that point: $f'(x_0)$
- Update the point to a new point $x_1 = x_0 - \eta f'(x_0)$, where η is a learning rate parameter
- Repeat the process with x_1, x_2, and so on until convergence

Gradient Descent

- **Basic Principle**: Gradient descent is an optimization algorithm used to minimize a cost function by iteratively moving in the direction opposite to the gradient of the cost function with respect to the parameters.
- **Update Rule**: In standard gradient descent, the update rule for updating the parameters θ in each iteration t is given by:

$$\theta_{t+1} = \theta_t - \alpha \nabla J(\theta_t)$$

  where:
  - θ_t is the parameter vector at iteration t.
  - α is the learning rate.
  - $\nabla J(\theta_t)$ is the gradient of the cost function J with respect to the parameters evaluated at θ_t.
- **Convergence**: Gradient descent converges to a local minimum of the cost function, but it may not find the global minimum if the cost function is non-convex.

Gradient Descent Algorithm - pseudocode

Algorithm Gradient Descent

Input : α: step size, x_0: initial guess

Output: x^*: approximate minimum

do

$\quad x_{k+1} \leftarrow x_k - \alpha \nabla f(x_k)$; // update guess

while $\nabla f(x^*) \approx 0$;

Legend:

- α: step size or learning rate determines the step size at each iteration while moving toward a minimum of a loss function.
- x_0: initial guess or starting point determines the starting point from which the algorithm will begin the optimization process.
- x_k: current guess.
- x^*: approximate minimum or the optimal solution is the point where the function is minimized.
- $\nabla f(x)$: gradient of the function is a vector that points in the direction of the greatest increase of the function at that point.

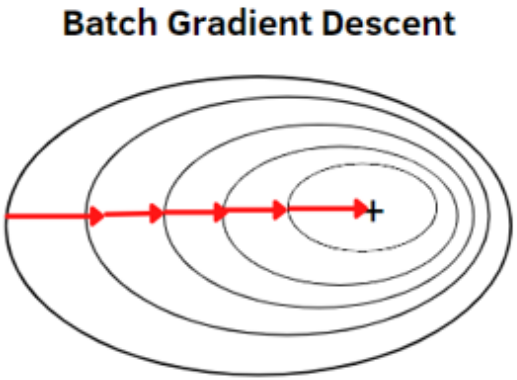
Gradient Descent - Example

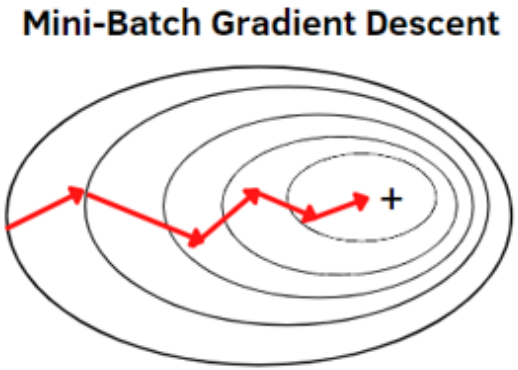
Consider the function $f(x) = x_1^2 + x_2^2$, and we want to find the minimum of this function using the stochastic gradient descent method. We set the learning rate to $\alpha = 0.1$, and the initial point to $x_0 = (1, 1)$. We'll run the algorithm for 100 iterations.

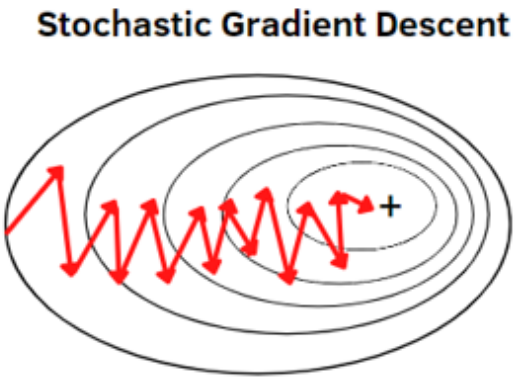
Iteration	Current Point x_t	Function Value $f(x_t)$	Gradient g_t
0	(1,1)	2	(2,2)
1	(0.6,0.6)	0.72	(1.2,1.2)
2	(0.36,0.36)	0.2592	(0.72,0.72)
...	...	...	...
99	(0,0)	0	(0,0)

After 100 iterations, the algorithm converges to the optimal point $x^* = (0, 0)$.

Types of Gradient Descent methods







Batch Gradient Descent Method

- **Basic Principle**: Batch Gradient Descent is the standard form of gradient descent where parameters are updated using the average gradient of the cost function computed over the entire training dataset.
- **Update Rule**: The update rule for Batch Gradient Descent is:

$$\theta_{t+1} = \theta_t - \alpha \nabla J(\theta_t)$$

  where:
  - θ_t is the parameter vector at iteration t.
  - α is the learning rate.
  - $\nabla J(\theta_t)$ is the gradient of the cost function J with respect to the parameters evaluated at θ_t using the entire training dataset.
- **Convergence**: Batch Gradient Descent typically converges smoothly to the global minimum of the cost function, but it can be computationally expensive, especially for large datasets.

Batch Gradient Descent Method

Algorithm Batch Gradient Descent Method

Input : Training set $\{(x^{(i)}, y^{(i)})\}_{i=1}^{m}$, learning rate α, number of iterations T
Output: Optimal parameters θ^*
Initialize θ randomly
for $t = 1$ **to** T **do**

Compute the gradient of the loss function $J(\theta)$ with respect to θ using the entire training set:

$$\nabla_\theta J(\theta; \{(x^{(i)}, y^{(i)})\}_{i=1}^{m}) = \frac{1}{m} \sum_{i=1}^{m} \nabla_\theta J(\theta; x^{(i)}, y^{(i)})$$

Update the parameters θ using the gradient:

$$\theta \leftarrow \theta - \alpha \nabla_\theta J(\theta; \{(x^{(i)}, y^{(i)})\}_{i=1}^{m})$$

Batch Gradient Descent - Definitions

Loss function ($J(\theta)$): A function that measures the difference between the model's predicted output and the actual output for a given input.

The goal of the stochastic gradient descent algorithm is to minimize this function over the training set by updating the model's parameters.

Gradient ($\nabla_\theta J(\theta; x^{(i)}, y^{(i)})$): A vector that points in the direction of steepest increase of the loss function with respect to the model's parameters.

It is computed for each training example $(x^{(i)}, y^{(i)})$ and used to update the parameters of the model.

Optimal parameters (θ^*): The set of parameters that minimize the loss function $J(\theta)$ over the training set.

These parameters are learned through the process of stochastic gradient descent.

Learning rate (α): A hyperparameter that controls the step size of each update to the model's parameters.

A larger learning rate leads to larger updates, but can cause the model to overshoot the optimum. A smaller learning rate can be slower but more stable.

Number of iterations (T): The number of times the entire training set is used to update the model's parameters.

Each iteration involves passing through the entire training set once and updating the parameters based on the gradients computed for each example.

Stochastic Gradient Descent (SGD)

- **Basic Principle**: Stochastic gradient descent is a variant of gradient descent that updates the parameters using an estimate of the gradient computed on a subset of the training data rather than the entire dataset.
- **Update Rule**: The update rule for stochastic gradient descent is similar to that of gradient descent, but it uses a stochastic estimate of the gradient:

$$\theta_{t+1} = \theta_t - \alpha \nabla J(\theta_t; x_i, y_i)$$

  where:
  - (x_i, y_i) is a data point from the training set.
  - $\nabla J(\theta_t; x_i, y_i)$ is the stochastic estimate of the gradient of the cost function J with respect to the parameters evaluated at θ_t using the data point (x_i, y_i).
- **Convergence**: Stochastic gradient descent may converge faster due to its higher variance updates but may not converge as smoothly as standard gradient descent.

Stochastic Gradient Descent Method

Algorithm Stochastic Gradient Descent Method

Input : Training set $\{(x^{(i)}, y^{(i)})\}_{i=1}^{m}$, learning rate α, number of iterations T
Output: Optimal parameters θ^*
Initialize θ randomly
for $t = 1$ **to** T **do**
Randomly shuffle the training set
for $i = 1$ **to** m **do**
Select a single example from the shuffled training set:

$$x^{(i_{\text{rand}})}, y^{(i_{\text{rand}})} \leftarrow \text{Randomly chosen example}$$

Compute the gradient of the loss function $J(\theta)$ with respect to θ using the single example:

$$\nabla_\theta J(\theta; x^{(i_{\text{rand}})}, y^{(i_{\text{rand}})}) \leftarrow \text{Gradient of } J(\theta) \text{ using } x^{(i_{\text{rand}})}, y^{(i_{\text{rand}})}$$

Update the parameters θ using the gradient:

$$\theta \leftarrow \theta - \alpha \nabla_\theta J(\theta; x^{(i_{\text{rand}})}, y^{(i_{\text{rand}})})$$

Summary of Stochastic Gradient Descent (SGD)

Advantages:

- Speed: SGD is faster than other variants of Gradient Descent such as Batch Gradient Descent and Mini-Batch Gradient Descent since it uses only one example to update the parameters.
- Memory Efficiency: Since SGD updates the parameters for each training example one at a time, it is memory-efficient and can handle large datasets that cannot fit into memory.
- Avoidance of Local Minima: Due to the noisy updates in SGD, it has the ability to escape from local minima and converge to a global minimum.

Summary of Stochastic Gradient Descent (SGD) cont.

Disadvantages:

- Noisy updates: The updates in SGD are noisy and have a high variance, which can make the optimization process less stable and lead to oscillations around the minimum.
- Slow Convergence: SGD may require more iterations to converge to the minimum since it updates the parameters for each training example one at a time.
- Sensitivity to Learning Rate: The choice of learning rate can be critical in SGD since using a high learning rate can cause the algorithm to overshoot the minimum, while a low learning rate can make the algorithm converge slowly.
- Less Accurate: Due to the noisy updates, SGD may not converge to the exact global minimum and can result in a suboptimal solution. This can be mitigated by using techniques such as learning rate scheduling and momentum-based updates.

Mini-Batch Gradient Descent Method

- **Basic Principle**: Mini-Batch Gradient Descent is a compromise between Batch Gradient Descent and SGD. It updates parameters using the gradient of the cost function computed over a small random subset of the training data at each iteration.
- **Update Rule**: The update rule for Mini-Batch Gradient Descent is similar to SGD, but it uses a small subset of the training data:

$$\theta_{t+1} = \theta_t - \alpha \nabla J(\theta_t; X_t, Y_t)$$

  where: - X_t and Y_t are randomly chosen mini-batches of the training data at iteration t.
- **Convergence**: Mini-Batch Gradient Descent combines the advantages of both Batch Gradient Descent and SGD. It converges faster than Batch Gradient Descent and has smoother convergence than SGD. The choice of mini-batch size affects the trade-off between convergence speed and computational efficiency.

Mini-Batch Gradient Descent Method

Algorithm Mini-Batch Stochastic Gradient Descent Method

Input : Training set $\{(x^{(i)}, y^{(i)})\}_{i=1}^{m}$, learning rate α, number of iterations T
Output: Optimal parameters θ^*
Initialize θ randomly
for $t = 1$ **to** T **do**
  Randomly shuffle the training set
  for $i = 1$ **to** $\lfloor \frac{m}{b} \rfloor$ **do**
    Select a batch of b examples from the shuffled training set:

$$\mathcal{B} = \{(x^{(i_j)}, y^{(i_j)})\}_{j=1}^{b}, \quad i_j \in \{1, \ldots, m\}$$

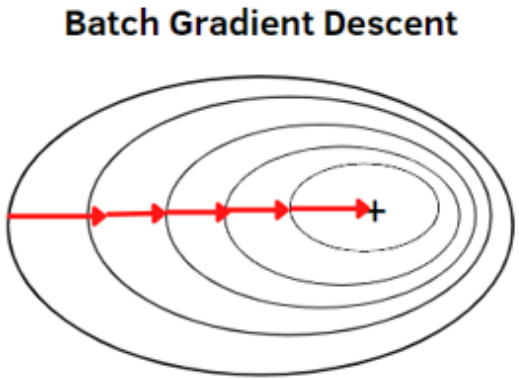
    Compute the gradient of the loss function $J(\theta)$ with respect to θ using the batch $\mathcal{B}$:

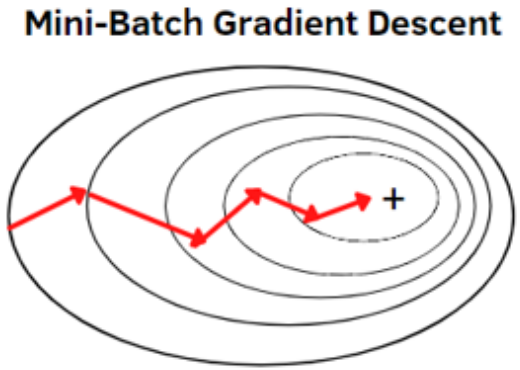
$$\nabla_\theta J(\theta; \mathcal{B}) = \frac{1}{b} \sum_{j=1}^{b} \nabla_\theta J(\theta; x^{(i_j)}, y^{(i_j)})$$

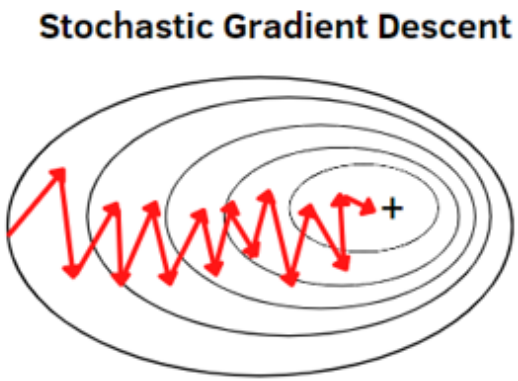
    Update the parameters θ using the gradient:

$$\theta \leftarrow \theta - \alpha \nabla_\theta J(\theta; \mathcal{B})$$

Summary of Gradient Descent methods







Main AI Optimization Algorithms - summary

- Newton's Method:
  - Uses a second-order approximation to the objective function.
  - Can converge faster than gradient descent, but requires computing the Hessian matrix.
  - Can get stuck in saddle points and other non-convex regions.
- Levenberg-Marquardt Algorithm:
  - Uses a combination of the gradient and Hessian matrix to update parameters.
  - Can handle non-linear least squares problems and can converge faster than gradient descent.
  - Can be slower than Newton's method and may require tuning of the damping parameter.
- Gradient Descent:
  - Update parameters in the direction of the negative gradient of the objective function.
  - Can get stuck in local minima.
  - Works well for convex functions and when the objective function is differentiable.

Overview

- Introduction optimisation methods in AI
- Applications in AI
- "Classic" Local Optimization Methods
  - Newton's Method
  - Levenberg Method
  - Gradient Descent Method
  - Stochastic Gradient Descent Method
  - Mini-batch Gradient Descent Method
- Comparison of Methods

References

1. S. J. Russell, P. Norvig, Artificial Intelligence: A Modern Approach", Financial Times Prentice Hall, 2019.
2. M. Flasiński, Introduction to Artificial Intelligence", Springer Verlang, 2016
3. M. Muraszkiewicz, R Nowak (ed.), Sztuczna Inteligencja dla inżynierów", Oficyna Wydawnicza PW, 2022
4. J. Prateek , Artificial Intelligence with Python", Packt 2017