04 Numerical Optimization

May 29, 2021

Information: *Brief introduction to convexity, optimization, gradient descent, and automatic differentiation supported by PyTorch*

Written by: *Zihao Xu*

Last update date: *05.29.2021*

1 Convexity

1.1 Introduction

- Convexity plays a vital rule in the design of optimization algorithms, which is largely due to fact that it is much easier to analyze and test algorithms in such a context.
- If the algorithm performs poorly even in the convex setting, typically we should not hope to see great results otherwise.
- Even though the optimization problems in ML/DL are generally non-convex, they often exhibit some properties of convex ones near local minimum.

1.2 Open and Closed Sets

- Define
$$A \subset \mathbb{R}^n$$
and open ball of diameter ϵ is $B(r, \epsilon) = \{r \in \mathbb{R}^n : \|r - r_o\| < \epsilon\}$
- A set A is **open** if
  - At every point, there is an open ball contained in A
  - $\forall r \in A,\ \exists \epsilon > 0$ s.t. $B(r, \epsilon) \subset A$
- A set A is **closed** if $A^c = \mathbb{R}^n - A$ is open
- A set A is compact if it is closed and bounded
- Facts:
  - $\mathbb{R}^N$ is both open and closed, but it is not compact
  - If A is compact, then every sequence in A has a limit point in A

1.3 Convex Sets

1.3.1 Definition

- A set C is convex if, for any $x, y \in C$ and $\theta \in \mathbb{R}$ with $0 \le \theta \le 1$:
$$\theta x + (1 - \theta) y \in C$$

  - Intuitively, it means if we take any two elements in C and draw a line segment between these two elements, then every point on that line segment also belongs to C
- The point $\theta x + (1-\theta)y$ is called a **convex combination** of the points x and y

1.3.2 Examples

- **All of $\mathbb{R}^n$.**
  - Given any $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$,
$$\theta\mathbf{x} + (1-\theta)\mathbf{y} \in \mathbb{R}^n$$
- **The non-negative orthant $\mathbb{R}^n_+$.**
  - $\mathbb{R}^n_+$ consists of all vectors in $\mathbb{R}^n$ whose elements are all non-negative
$$\mathbb{R}^n_+ = \{\mathbf{x} : x_i \geq 0 \ \forall i = 1, \cdots, n\}$$
  - Given any $\mathbf{x}, \mathbf{y} \in \mathbb{R}^N_+$ and $0 \leq \theta \leq 1$,
$$(\theta\mathbf{x} + (1-\theta)\mathbf{y})_i = \theta x_i + (1-\theta)y_i \geq 0 \ \forall i$$
- **Norm balls**
  - Let $\|\cdot\|$ be some norm on $\mathbb{R}^n$ (e.g., the Euclidean norm $\|\mathbf{x}\|_2 = \sqrt{\Sigma_{i=1}^n x_i^2}$). Then the set $\{\mathbf{x} : \|\mathbf{x}\| \leq 1\}$ is a convex set.
  - Given $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ with $\|\mathbf{x}\| \leq 1, \|\mathbf{y}\| \leq 1$ and $0 \leq \theta \leq 1$. Then
$$\|\theta\mathbf{x} + (1-\theta)\mathbf{y}\| \leq \|\theta\mathbf{x}\| + \|(1-\theta)\mathbf{y}\| = \theta\|\mathbf{x}\| + (1-\theta)\|\mathbf{y}\| \leq 1$$
    where the **triangle inequality** and the **positive homogeneity** of norms are used
- **Affine subspaces and polyhedra**
  - Given a matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ and a vector $\mathbf{b} \in \mathbb{R}^m$, an affine subspace is the set $\{\mathbf{x} \in \mathbb{R}^n : \mathbf{A}\mathbf{x} = \mathbf{b}\}$ (note this could possible be empty if $\mathbf{b}$ is not in range of $\mathbf{A}$).
  - Given $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ s.t. $\mathbf{A}\mathbf{x} = \mathbf{A}\mathbf{y} = \mathbf{b}$, then for $0 \leq \theta \leq 1$:
$$\mathbf{A}(\theta\mathbf{x} + (1-\theta)\mathbf{y}) = \theta\mathbf{A}\mathbf{x} + (1-\theta)\mathbf{A}\mathbf{y} = \theta\mathbf{b} + (1-\theta)\mathbf{b} = \mathbf{b}$$
  - Similarly, a polyhedron is the set $\{\mathbf{x} \in \mathbb{R}^n : \mathbf{A}\mathbf{x} \preceq \mathbf{b}\}$ (also possibly empty), where $\preceq$ denotes componentwise inequality
    * All the entries of $\mathbf{A}\mathbf{x}$ are less than or equal to their corresponding element in $\mathbf{b}$
  - Given $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ that satisfy $\mathbf{A}\mathbf{x} \leq \mathbf{b}$ and $\mathbf{A}\mathbf{y} \leq \mathbf{b}$ and $0 \leq \theta \leq 1$:
$$\mathbf{A}(\theta\mathbf{x} + (1-\theta)\mathbf{y}) \leq \theta\mathbf{b} + (1-\theta)\mathbf{b} = \mathbf{b}$$
- **Intersection of convex sets**
  - Suppose $C_1, C_2, \cdots, C_k$ are convex sets. Then their intersection
$$\bigcap_{i=1}^{k} C_i = \{x : x \in C_i \ \forall i = 1, \cdots, k\}$$
    is also a convex set

  - Given $x, y \in \bigcap_{i=1}^{k} C_i$ and $0 \leq \theta \leq 1$. Then

$$\theta x + (1-\theta)y \in C_i \ \forall i = 1, \cdots, k$$

    by the definition of a convex set. Therefore

$$\theta x + (1-\theta)y \in \bigcap_{i=1}^{k} C_i$$

  - Note that the *union* of convex sets in general will not be convex
- **Positive semidefinite matrices**
  - The set of all symmetric positive semidefinite matrices, often times called the *positive semidefinite cone* and denoted $\mathbb{S}_+^n$ is a convex set (in general, $\mathbb{S}^n \subset \mathbb{R}^{n\times n}$ denotes the set of symmetric $n \times n$ matrices).
  - A matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$ is symmetric positive semidefinite if and only if $\mathbf{A} = \mathbf{A}^T$ and for all $\mathbf{x} \in \mathbb{R}^n, \mathbf{x}^T\mathbf{A}\mathbf{x} \leq 0$
  - Given two symmetric positive semidefinite matrices $\mathbf{A}, \mathbf{B} \in \mathbb{S}_+^n$ and $0 \leq \theta \leq 1$, then for any $\mathbf{x} \in \mathbb{R}^n$,

$$\mathbf{x}^T(\theta\mathbf{A} + (1-\theta)\mathbf{B})\mathbf{x} = \theta\mathbf{x}^T\mathbf{A}\mathbf{x} + (1-\theta)\mathbf{x}^T\mathbf{B}\mathbf{x} \geq 0$$

  - The logic to show that all **positive definite**, **negative definite**, and **negative semidefinite** matrices are each also convex

1.4 Convex Functions

1.4.1 Definition

- A function $f : \mathbb{R}^n \to \mathbb{R}$ is convex if its domain (denoted $\mathcal{D}(f)$) is a *convex set*, and if, for all $\mathbf{x}, \mathbf{y} \in \mathcal{D}(f)$ and $\theta \in \mathbb{R}, 0 \leq \theta \leq 1$:

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

  - Intuitively, it means if we pick any two points on the graph pf a convex function and draw a straight line between them, then the portion of function between these two points will lie below this straight line.
- A function is called **strictly convex** if the definition holds with strict inequality for $\mathbf{x} \neq \mathbf{y}$ and $0 < \theta < 1$
- A function f is called **concave** if $-f$ is convex
- A function f is called **strictly concave** if $-f$ is strictly convex

1.4.2 First Order Condition for Convexity

- Suppose a function $f : \mathbb{R}^n \to \mathbb{R}$ is differentiable. Then f is convex if and only if $\mathcal{D}(f)$ is a convex set and for $\mathbf{x}, \mathbf{y} \in \mathcal{D}(f)$,

$$f(\mathbf{y}) \geq f(\mathbf{x}) + \nabla_x f(\mathbf{x})^T(\mathbf{y} - \mathbf{x})$$

  where the function $f(\mathbf{x}) + \nabla_x f(\mathbf{x})^T(\mathbf{y} - \mathbf{x})$ is called the **first-order approximation** to the function f at the point $\mathbf{x}$
  - Intuitively, this can be thought of as approximating f with its tangent line at the point $\mathbf{x}$.

- Similarly, f would be
  - strictly convex if this holds with strict inequality
  - concave if the inequality is reversed
  - strictly concave if the reverse inequality is strict

1.4.3 Second Order Condition for Convexity

- Suppose a function $f : \mathbb{R}^n \to \mathbb{R}$ is twice differentiable. Then f is convex if and only if $\mathcal{D}(f)$ is a convex set and its *Hessian* is positive semidefinite:

$$\forall x \in \mathcal{D}(f),\ \nabla_x^2 f(x) \succeq 0$$

  - Here the notation $\succeq$ refers to positive semidefiniteness
  - In one dimension, this is equivalent to the condition that the second derivative $f''(x)$ always be positive
- Similarly, f is
  - strictly convex if its Hessian is positive definite
  - concave if the Hessian is negative semidefinite
  - strictly concave if the Hessian is negative definite

1.4.4 Jensen's Inequality

- Start with the inequality in the basic definition of a convex function

$$f(\theta x + (1-\theta)y) \leq \theta f(x) + (1-\theta)f(y) \text{ for } 0 \leq \theta \leq 1$$

  Using induction, extend this definition to convex combinations of more than one point

$$f\left(\sum_{i=1}^{k} \theta_i x_i\right) \leq \sum_{i=1}^{k} \theta_i f(x_i) \ \text{ for } \sum_{i=1}^{k} \theta_i = 1,\ \theta_i \geq 0\ \forall i$$

  This can also extend to infinite sums or integrals. In the latter case, the inequality can be written as

$$f\left(\int p(x)x dx\right) \leq \int p(x)f(x)dx \quad \text{for} \quad \int p(x)dx = 1,\ p(x) \geq 0\ \ \forall x$$

  Since $\int p(x)dx = 1$, it is common to consider it a probability density, in which case the previous equation can be written in terms of expectations

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

  which is called **Jensen's inequality**

1.4.5 Examples

- **Exponential**
  - Let $f : \mathbb{R} \to \mathbb{R}, f(x) = e^{ax}$ for any $a \in \mathbb{R}$.
  - $f''(x) = a^2 e^{ax}$ is positive for all x
- **Negative logarithm**
  - Let $f : \mathbb{R} \to \mathbb{R}, f(x) = -\text{log}x$ with domain $\mathcal{D}(f) = \mathbb{R}_{++} = \{x : x > 0\}$
  - $f''(x) = \frac{1}{x^2} > 0$ for all x

- **Affine functions**
  - Let $f : \mathbb{R}^n \to \mathbb{R}, f(\mathbf{x}) = \mathbf{b}^T\mathbf{x} + c$ for some $\mathbf{b} \in \mathbb{R}^n, c \in \mathbb{R}$
  - The Hessian $\nabla^2_{\mathbf{x}} f(\mathbf{x}) = 0$ for all $\mathbf{x}$
  - Affine functions of this form are the **only** functions that are **both convex and concave**
- **Quadratic function**
  - Let $f : \mathbb{R}^n \to \mathbb{R}, f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\mathbf{A}\mathbf{x} + \mathbf{b}^T\mathbf{x} + c$ for a symmetric matrix $\mathbf{A} \in \mathbb{S}^n, b \in \mathbb{R}^n$ and $c \in \mathbb{R}$
  - The Hessian for this function is $\nabla^2_{\mathbf{x}} f(\mathbf{x}) = \mathbf{A}$
  - The convexity or non-convexity of f is determined entirely by whether or not $\mathbf{A}$ is positive semidefinite
  - The **squared Euclidean norm** $f(\mathbf{x}) = \|\mathbf{x}\|_2^2 = \mathbf{x}^T\mathbf{x}$ is a special case of quadratic functions where $\mathbf{A} = \mathbf{I}, \mathbf{b} = \mathbf{0}, c = 0$, so it is therefore a **strictly convex function**
- **Norms**
  - Let $f : \mathbb{R}^n \to \mathbb{R}$ be some norm on $\mathbb{R}^n$.
  - By the **triangle inequality** and **positive homogeneity** of norms, for $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n, 0 \leq \theta \leq 1$,
$$f(\theta\mathbf{x} + (1-\theta)\mathbf{y}) \leq f(\theta\mathbf{x}) + f((1-\theta)\mathbf{y}) = \theta f(\mathbf{x}) + (1-\theta) f(\mathbf{y})$$
  - Not possible to prove convexity based on the first or second order conditions because norms are not generally differentiable
- **Nonnegative weighted sums of convex functions**
  - Let $f_1, f_2, \cdots, f_k$ be convex functions and $w_1, w_2, \cdots, w_k$ be nonnegative real numbers. Then
$$f(x) = \sum_{i=1}^{k} w_i f_i(x)$$
    is a convex function, since
$$\begin{aligned} f(\theta x + (1-\theta)y) &= \sum_{i=1}^{k} w_i f_i(\theta x + (1-\theta)y) \\ &\leq \sum_{i=1}^{k} w_i(\theta f_i(x) + (1-\theta) f_i(y)) \\ &= \theta \sum_{i=1}^{k} w_i f_i(x) + (1-\theta) \sum_{i=1}^{k} w_i f_i(y) \\ &= \theta f(x) + (1-\theta) f(x) \end{aligned}$$

2 Optimization

2.1 Motivation

- Most ML/DL algorithms involve **optimization** of some sort.
  - Optimization refers to the task of either **minimizing** or maximizing some function $f(\mathbf{x})$ by altering $\mathbf{x}$
  - Usually phrase most optimization problems in terms of minimizing $f(\mathbf{x})$
  - Maximization may be accomplished via s minimization algorithm by minimizing $-f(\mathbf{x})$
- Usually the function we want to minimize is called the **objective function**, or **criterion**. In ML/DL contexts, the name **loss function** is often used.
  - As mentioned in introduction, a loss function quantifies the *distance* between the **real** and **predicted** value of the target.
  - Usually be a non-negative number where smaller values are better and perfect predictions incur a loss of 0
  - Usually denoted as $L(\boldsymbol{\theta})$ where $\boldsymbol{\theta}$ is usually the parameter of ML/DL models
- Usually denote the value that minimizes a function with a superscript $*$
  - $\boldsymbol{\theta}^* = \underset{\boldsymbol{\theta}}{\operatorname{argmin}} L(\boldsymbol{\theta})$
- Most ML/DL algorithms are so complex that it is difficult or impossible to find the closed form solution for the optimization problem
  - Use numerical optimization method instead
- One common algorithm is **gradient descent**, other optimization algorithms are
  - Expectation Maximization
  - Sampling-based optimization
  - Greedy optimization

2.2 Local Minimum and Global Minimum

2.2.1 Local Minimum

- Let $f : A \to \mathbb{R}$ where $A \subset \mathbb{R}^N$, a point x is locally minimal if it is available and if there exists some $R < 0$ such that all feasible points z with $\|x - z\|_2 \leq R$, satisfy $f(x) \leq f(z)$
- **Necessary** condition for local minimum
  - Let f be continuously differentiable and let $x \in A$ be a local minimum, then $\nabla f(x) = 0$
- **Saddle Point**:
  - We say that $x \in A$ is a saddle point of f if $\nabla f(x) = 0$ and r_o is not a local minimum

2.2.2 Global Minimum

- A point x is globally minimal if it is available and for all feasible points z, $f(x) \leq f(z)$
  - A global minimum must also be a local minimum

2.3 Optimization Theorems

- Let $f : A \to \mathbb{R}$ where $A \subset \mathbb{R}^N$
  - If f is continuous and A is **compact**, then f takes on a global minimum in A
  - If f is **convex** on A, then any local minimum is a global minimum
  - If f is continuously differentiable and convex on A, then $\nabla f(x) = 0$ implies the $x \in A$ is a global minimum of f

- **Important Facts**:
  - Global minimum **may not be unique**
  - If A is closed but not bounded, then f may not take on a global minimum
  - Most interesting functions in ML/DL are **not** convex

2.4 Convex Optimization

- Formally, a convex optimization problem is an optimization problem of the form

$$\begin{aligned} &\text{minimize} && f(x) \\ &\text{subject to} && x \in C \end{aligned}$$

  where f is a convex function, C is a convex set, and x is the optimization variable
- Often written as

$$\begin{aligned} &\text{minimize} && f(x) && \\ &\text{subject to} && g_i(x) \le 0 && i = 1, \cdots, m \\ & && h_i(x) = 0 && i = 1, \cdots, p \end{aligned}$$

  where f is a convex function, g_i are convex functions and h_i are affine functions and x is the optimization variable

2.5 Constrained Optimization

2.5.1 Definition

- Sometimes we wish not only to maximize or minimize a function $f(\mathbf{x})$ over all possible values of $\mathbf{x}$. Instead the maximal or minimal value of $f(\mathbf{x})$ for values of $\mathbf{x}$ in some set $\mathbb{S}$. This is known as **constrained optimization**
- Points $\mathbf{x}$ that lies within the set $\mathbb{S}$ are called **feasible** points in constrained optimization terminology

2.5.2 Solution

- **Intuition**: Design a different, **unconstrained** optimization problem whose solution can be converted into a solution to the original constrained optimization problem
  - For example, to minimize $f(\mathbf{x})$ for $\mathbf{x} \in \mathbb{R}^2$ with $\mathbf{x}$ constrained to have exactly unit L^2 norm, we can instead minimize

$$g(\theta) = f([\cos\theta, \sin\theta]^T)$$

    with respect to θ, then return $[\cos\theta, \sin\theta]$ as the solution to the original problem
  - Requires creativity
  - The transformation between optimization problems must be designed specifically for each case we counter
- **Karush-Kuhn-Tucker** (KKT) approach
  - Provides a very general solution to constrained optimization

3 Gradient Descent

3.1 Basic Concepts

3.1.1 Definition

- **Definition**:
  - A **first-order iterative** optimization algorithm for finding **local minimum** of a **differential** function.
    - ∗ The idea is to take *repeated steps* in the opposite direction of the *gradient* of the function at the current point, because this is the direction of steepest descent.
    - ∗ As it only calculates the *first-order* derivative, it requires the objective function to be *differential* and is called *first-order optimization algorithms*
      - · Some optimization algorithms that also use the Hessian matrix are called *second-order optimization algorithms*
    - ∗ Converge when first-order derivative is zero, which only ensures reaching **local minimum** for general functions
      - · That is to say, the start point will sometimes affect final convergence
    - ∗ Generally speaking, gradient descent algorithms converge to the **global minimum** of continuously differentiable **convex** functions
- **Theory**:
  - Based on the observation that if the multi-variable function $F(\mathbf{x})$ is defined and differentiable in a neighborhood of a point $\mathbf{a}$, then $F(\mathbf{x})$ decreases **fastest** if one goes from $\mathbf{a}$ in the direction of the negative gradient of F at $\mathbf{a}$, which is $-\nabla F(\mathbf{a})$. It follows that if

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

    for a $\gamma \in \mathbb{R}_+$ small enough, then

$$F(\mathbf{a}_n) \geq F(\mathbf{a}_{n+1})$$

- Simple form of **vanilla gradient descent** (GD):
  1. Start at random parameter $\boldsymbol{\theta}$
  2. Repeat until converged
     - $\mathbf{d} \leftarrow -\nabla L(\boldsymbol{\theta})$
     - $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \mathbf{d}^T$
  - α is called **learning rate** or **step size**

3.1.2 Compute Loss Gradient

- Take the **mean square error** as an example:

$$\begin{aligned}\nabla_{\boldsymbol{\theta}} L_{MSE}(\boldsymbol{\theta}) &= \nabla_{\boldsymbol{\theta}} \left\{ \frac{1}{N} \sum_{i=1}^{N} \|\mathbf{y}_i - f_{\boldsymbol{\theta}}(\mathbf{x}_i)\|^2 \right\} \\ &= \frac{1}{N} \sum_{i=1}^{N} \nabla_{\boldsymbol{\theta}} \left\{ (\mathbf{y}_i - f_{\boldsymbol{\theta}}(\mathbf{x}_i))^T (\mathbf{y}_i - f_{\boldsymbol{\theta}}(\mathbf{x}_i)) \right\}\end{aligned}$$

  Use the chain rule and scale-by-vector matrix calculus identity that

$$\frac{\partial \mathbf{x}^T \mathbf{x}}{\partial \mathbf{x}} = 2\mathbf{x}^T$$

We can get

$$\begin{aligned}\nabla_{\boldsymbol{\theta}} L_{MSE}(\boldsymbol{\theta}) &= \frac{2}{N}\sum_{i=1}^{N}(\mathbf{y}_i - f_{\boldsymbol{\theta}}(\mathbf{x}_i))^T \nabla_{\boldsymbol{\theta}}(\mathbf{y}_i - f_{\boldsymbol{\theta}}(\mathbf{x}_i)) \\ &= \frac{2}{N}\sum_{i=1}^{N}(\mathbf{y}_i - f_{\boldsymbol{\theta}}(\mathbf{x}_i))^T \nabla_{\boldsymbol{\theta}}(-f_{\boldsymbol{\theta}}(\mathbf{x}_i)) \\ &= -\frac{2}{N}\sum_{i=1}^{N}(\mathbf{y}_i - f_{\boldsymbol{\theta}}(\mathbf{x}_i))^T \nabla_{\boldsymbol{\theta}}(f_{\boldsymbol{\theta}}(\mathbf{x}_i))\end{aligned}$$

- The result of the gradient usually includes three parts:
  - Sum over training data. It consists of a lot of computations but the way of computation is relatively easy and straight forward
  - Prediction error term such as $\mathbf{y}_i - f_{\boldsymbol{\theta}}(\mathbf{x}_i)$ in MSE, which is usually easy to get
  - Gradient of inference function $\nabla_{\boldsymbol{\theta}}(f_{\boldsymbol{\theta}}(\mathbf{x}_i))$, which is difficult to solve
    - Enabled by **automatic differentiation** built into modern domain specific languages such as Pytorch, Tensorflow, ...
    - For neural networks, this is known as **back propagation**

3.1.3 Select appropriate learning rate

- Too large α leads to instability and even divergence
- Too small α leads to slow convergence
- **Steepest gradient descent** use **line search** to compute the best α
  1. Start at random parameter $\boldsymbol{\theta}$
  2. Repeat until converged
     - $\mathbf{d} \leftarrow -\nabla L(\boldsymbol{\theta})$
     - $\alpha^* \leftarrow \underset{\alpha}{\operatorname{argmin}}\{L(\boldsymbol{\theta} + \alpha\mathbf{d}^T)\}$
     - $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^*\mathbf{d}^T$
- **Adaptive learning rates** may help, but not always
  - $\alpha = \frac{1}{t}$, approaches 0 but can cover an infinite distance since $\lim_{a\to\infty}\sum_{t=1}^{a}\frac{1}{t} = \infty$
- **Coordinate Descent** update one parameter at a time
  - Removes problem of selecting step size
  - Each update can be very fast, but lots of updates

3.1.4 Slow convergence due to Poor Conditioning

- **Conditioning** refers to how rapidly a function changes with respect to small changes in its inputs.
- Consider the function

  $$f(x) = \mathbf{A}^{-1}\mathbf{x}$$

  When $\mathbf{A} \in \mathbb{R}^{n\times n}$ has an eigenvalue decomposition, its **condition number** is

  $$\max_{i,j}\left|\frac{\lambda_i}{\lambda_j}\right|$$

  This is the ratio of the magnitude of the largest and smallest eigenvalue
- A problem with a **low condition number** is said to be **well-conditioned**, while a problem with a high condition number is said to be ill-conditioned

  - In non-mathematical terms, an ill-conditioned problem is one where, for a small change in the inputs there is a large change in the answer or dependent variable, which means the correct solution to the equation becomes hard to find
  - Condition number is a property of the problem
- **Gradient descent** is very sensitive to **condition number** of the problem
  - No good choice of step size. Tiny change in one variable could lead to great change in dependent variable.
- **Solutions:**
  - **Newton's method:** Correct for local second derivative.
    * Too much computation and too difficult to implement
    * Harmful when near saddle points
  - **Alternative methods**:
    * Preconditioning: Easy, but tends to be ad-hoc, not so robust
    * Momentum

3.1.5 Vanishing Gradients

- The most insidious problem to encounter
- Some function leads to almost zero gradients far away from local minimums, which makes the optimization stuck for a long time or even stop.
- For example, assume that we want to minimize the function

  $$f(x) = \tanh(x)$$

  The derivative is

  $$f'(x) = 1 - \tanh^2(x)$$

  If we happen to get started at $x = 4$ then the derivative at that point is

  $$f'(4) = 0.0013$$

  The gradient is close to nil. Consequently, optimization will get stuck for a long time before we make progress
- **Possible Solutions**:
  - Reparameterize the problem
  - Good initialization of the parameter
  - Reconstruct the objective function (e.g., change activation function in neural networks)

4 Automatic Differentiation via Pytorch

4.1 Data Manipulation via Pytorch

- The *n*-dimensional array is usually called the tensor
- *tensor* class in Pytorch is similar to *NumPy*'s *ndarray* with several additional features
  - GPU is well-supported to accelerate the computation whereas *NumPy* only supports CPU computation
  - *tensor* class supports automatic differentiation
- The Pytorch documentation shows the full attributes

4.1.1 Create a tensor

- To get started, import **torch**. Although it's called Pytorch, we should import **torch** instead of pytorch

[1]:
```
import torch
# Check the version of a module
print(torch.__version__)
```

```
1.8.1
```

- Common ways to creating a tensor
  - torch.arange(start,end,step)
  - torch.zeros(shape)
  - torch.ones(shape)
  - torch.randn(shape)
  - torch.tensor(elements)

[2]:
```
torch.arange(0, 12, 1)
```

[2]:
```
tensor([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
```

[3]:
```
torch.zeros((3, 4))
```

[3]:
```
tensor([[0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.]])
```

[4]:
```
torch.ones((2, 5))
```

[4]:
```
tensor([[1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1.]])
```

[5]:
```
torch.randn(10)
```

[5]:
```
tensor([-1.9015, -0.5008,  1.2312,  0.2638, -0.3826, -0.5280, -0.1467, -1.3956,
        -0.0751,  0.6405])
```

[6]:
```
torch.tensor([[1, 2, 3], [4, 5, 6]])
```

```
[6]: tensor([[1, 2, 3],
             [4, 5, 6]])
```

- One can access a tensor's shape by viewing the **shape** attribute

```
[7]: torch.ones((4, 5)).shape
```

```
[7]: torch.Size([4, 5])
```

- **reshape** method can change the shape of a tensor without altering either the number of elements or their values.
  - No need to manually specify every dimension
  - Can place -1 for the dimension that we would like tensors to automatically infer

```
[8]: torch.arange(12).reshape(3, 4)
```

```
[8]: tensor([[ 0,  1,  2,  3],
             [ 4,  5,  6,  7],
             [ 8,  9, 10, 11]])
```

```
[9]: torch.arange(12).reshape(3, -1)
```

```
[9]: tensor([[ 0,  1,  2,  3],
             [ 4,  5,  6,  7],
             [ 8,  9, 10, 11]])
```

4.1.2 Type of a tensor

- Usually, a tensor is created as **tensor.float32** (32-bit floating point) by default. One can view its type in the **dtype** attribute
  - When creating a tensor using **torch.tensor**, tensor with all integers would be created as **torch.int64** (64-bit signed integer)
- Full tensor types can be viewed in the documentation

```
[10]: torch.ones(10).dtype
```

```
[10]: torch.float32
```

```
[11]: torch.tensor([[1, 2, 3], [4, 5, 6]]).dtype
```

```
[11]: torch.int64
```

```
[12]: # Only add one dot after the first element
      torch.tensor([[1., 2, 3], [4, 5, 6]]).dtype
```

```
[12]: torch.float32
```

- One can assign the wanted type when creating the tensor by setting the **dtype** attribute to
  - *torch.float32*, 32-bit floating point
  - *torch.float64*, 64-bit floating point

  - *torch.uint8*, 8-bit unsigned integer
  - *torch.int8*, 8-bit signed integer
  - *torch.int32*, 32-bit signed integer
  - *torch.int64*, 64-bit signed integer
  - *torch.bool*, Boolean

```
[13]: torch.ones(10, dtype=torch.float64).dtype
```

```
[13]: torch.float64
```

```
[14]: torch.ones(10, dtype=torch.uint8).dtype
```

```
[14]: torch.uint8
```

```
[15]: torch.ones(10, dtype=torch.int32).dtype
```

```
[15]: torch.int32
```

- One can also construct the type of a tensor from list or numpy array using the method:
  - *FloatTensor*, 32-bit floating point
  - *DoubleTensor*, 64-bit floating point
  - *ByteTensor*, 8-bit unsigned integer
  - *CharTensor*, 8-bit signed integer
  - *IntTensor*, 32-bit signed integer
  - *LongTensor*, 64-bit signed integer
  - *BoolTensor*, Boolean

```
[16]: torch.DoubleTensor([1, 2, 3]).dtype
```

```
[16]: torch.float64
```

```
[17]: torch.ByteTensor([1, 2, 3]).dtype
```

```
[17]: torch.uint8
```

4.1.3 Use GPU for tensor computation

- Unless otherwise specified, a new tensor will be stored in main memory and designated for CPU-based computation
- One can check which device the tensor is designated for by viewing the **device** attribute

```
[18]: torch.ones(10).device
```

```
[18]: device(type='cpu')
```

- One can always set the create a device if a GPU supporting cuda is available and use **to(device)** method to determine the device on which a tensor is or will be allocated
  - Assign the **device** parameter when creating a tensor also works

```
[19]: cuda0 = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
      cuda0
```

```
[19]: device(type='cuda', index=0)
```

```
[20]: torch.ones(5, device=cuda0)
```

```
[20]: tensor([1., 1., 1., 1., 1.], device='cuda:0')
```

```
[21]: # If available, indexing the cuda also works
      torch.ones(5, device=0)
```

```
[21]: tensor([1., 1., 1., 1., 1.], device='cuda:0')
```

```
[22]: # If avaiable, the string also works
      torch.ones(5, device="cuda:0")
```

```
[22]: tensor([1., 1., 1., 1., 1.], device='cuda:0')
```

```
[23]: # Use the to method the move a tensor
      x = torch.ones(5).to(cuda0)
      x
```

```
[23]: tensor([1., 1., 1., 1., 1.], device='cuda:0')
```

```
[24]: # One can also move a tensor from GPU to CPU
      x.to("cpu").device
```

```
[24]: device(type='cpu')
```

4.1.4 Operations

- Common standard arithmetic operators have all been lifted to element-wise operations

```
[25]: x = torch.tensor([1., 2., 4., 8.])
      c = 1.
      x + c, x * c, x**c
```

```
[25]: (tensor([2., 3., 5., 9.]), tensor([1., 2., 4., 8.]), tensor([1., 2., 4., 8.]))
```

```
[26]: x = torch.tensor([1., 2., 4., 8.])
      y = torch.tensor([4., 3., 2., 1.])
      x + y, x * y, x**y
```

```
[26]: (tensor([5., 5., 6., 9.]),
       tensor([4., 6., 8., 8.]),
       tensor([ 1.,  8., 16.,  8.]))
```

```
[27]: torch.exp(x)
```

```
[27]: tensor([2.7183e+00, 7.3891e+00, 5.4598e+01, 2.9810e+03])
```

- Matrix multiplication is also supported

```
[28]: A = torch.randn((4, 3))
      B = torch.randn((3, 5))
      # Two ways of matrix multiplication
      torch.mm(A, B), A @ B
```

```
[28]: (tensor([[ 1.2511,  0.0413,  1.0971,  1.8404, -1.1420],
               [ 1.5384, -0.7219,  1.2509,  0.2686,  0.7976],
               [ 0.9634,  1.2131,  0.4805,  1.0209, -1.1431],
               [ 3.6579, -3.2721,  3.2438, -0.2472,  3.5113]]),
       tensor([[ 1.2511,  0.0413,  1.0971,  1.8404, -1.1420],
               [ 1.5384, -0.7219,  1.2509,  0.2686,  0.7976],
               [ 0.9634,  1.2131,  0.4805,  1.0209, -1.1431],
               [ 3.6579, -3.2721,  3.2438, -0.2472,  3.5113]]))
```

```
[29]: A = torch.randn((5, 4))
      B = torch.randn((5, 4))
      # Two ways of elementwise multiplication
      torch.mul(A, B), A * B
```

```
[29]: (tensor([[-0.3360,  1.1862, -0.0905,  0.2306],
               [ 0.0185, -0.5806,  1.0578,  0.1829],
               [ 0.9390,  1.4462,  0.3555, -0.1335],
               [ 2.0323, -0.4050, -0.7776, -0.6367],
               [ 0.2002,  0.1821, -0.1532, -0.2998]]),
       tensor([[-0.3360,  1.1862, -0.0905,  0.2306],
               [ 0.0185, -0.5806,  1.0578,  0.1829],
               [ 0.9390,  1.4462,  0.3555, -0.1335],
               [ 2.0323, -0.4050, -0.7776, -0.6367],
               [ 0.2002,  0.1821, -0.1532, -0.2998]]))
```

- We can also **concatenate** multiple tensors together, stacking them end-to-end to form a larger tensor. We just need to provide a list of tensors and tell the system along which axis to concatenate.

```
[30]: A = torch.arange(0, 12, 1).reshape((3, 4))
      B = torch.ones((3, 4))
      # dim stands for the index of dimension in which the tensors are concatenated
      torch.cat((A, B), dim=0), torch.cat((A, B), dim=1)
```

```
[30]: (tensor([[ 0.,  1.,  2.,  3.],
               [ 4.,  5.,  6.,  7.],
               [ 8.,  9., 10., 11.],
               [ 1.,  1.,  1.,  1.],
               [ 1.,  1.,  1.,  1.],
               [ 1.,  1.,  1.,  1.]]),
       tensor([[ 0.,  1.,  2.,  3.,  1.,  1.,  1.,  1.],
               [ 4.,  5.,  6.,  7.,  1.,  1.,  1.,  1.],
               [ 8.,  9., 10., 11.,  1.,  1.,  1.,  1.]]))
```

- Also, we can construct a binary tensor via logical statements

[31]:
```
A = torch.arange(0, 12, 1).reshape((3, 4))
B = 5 * torch.ones((3, 4))
A == B, A > B
```

[31]:
```
(tensor([[False, False, False, False],
         [False,  True, False, False],
         [False, False, False, False]]),
 tensor([[False, False, False, False],
         [False, False,  True,  True],
         [ True,  True,  True,  True]]))
```

4.1.5 Broadcasting Mechanism

- Under certain conditions, even shapes differ, we can still perform **element-wise** operations by invoking the **broadcasting mechanism**.
  - First, expand one or both arrays by copying elements appropriately so that after this transformation, the two tensors have the same shape.
  - Second, carry out the element-wise operation on the resulting arrays
- In most cases, we broadcast along an axis where an array initially only has length 1

[32]:
```
a = torch.arange(3).reshape(3, 1)
b = torch.arange(2).reshape(1, 2)
a, b, a + b
```

[32]:
```
(tensor([[0],
         [1],
         [2]]),
 tensor([[0, 1]]),
 tensor([[0, 1],
         [1, 2],
         [2, 3]]))
```

4.1.6 Indexing and Slicing

- As in standard Python lists, we can access elements according to their relative position to the end of the list by using negative indices

[33]:
```
X = torch.arange(16).reshape(4, 4)
X, X[-1], X[1:3]
```

[33]:
```
(tensor([[ 0,  1,  2,  3],
         [ 4,  5,  6,  7],
         [ 8,  9, 10, 11],
         [12, 13, 14, 15]]),
 tensor([12, 13, 14, 15]),
 tensor([[ 4,  5,  6,  7],
         [ 8,  9, 10, 11]]))
```

- Can also index using binary tensor

```
[34]: A = torch.arange(0, 12, 1).reshape((3, 4))
      B = 5 * torch.ones((3, 4))
      A, B, A[A > B]
```

```
[34]: (tensor([[ 0,  1,  2,  3],
               [ 4,  5,  6,  7],
               [ 8,  9, 10, 11]]),
       tensor([[5., 5., 5., 5.],
               [5., 5., 5., 5.],
               [5., 5., 5., 5.]]),
       tensor([ 6,  7,  8,  9, 10, 11]))
```

4.2 Automatic Calculation of Gradients

- In practice, based on our designed model, the system builds a **computational graph**, tracking which data combined through which operations to produce the output. Automatic differentiation enables the system to subsequently **backpropagate gradients**.
  - Here *backpropagate* simply means to trace through the computational graph, filling in the partial derivatives with respect to each parameter

4.2.1 Intuition

- All computation can be broken into simple components
  - sum
  - multiply
  - exponential
  - convolution
  - …
- Derivatives for each simple component can be derived mathematically
- Derivatives for **any composition** can be derived via **chain rule**

4.2.2 Automatic Differentiation - Forward Mode

- **Notice:** Here is only my naive understanding about automatic differentiation since I haven't found some detailed mathematical explanation about it.
- One naive way of find the partial derivative respect to i-th variable $\frac{\partial f}{\partial x_i}$ in the function

$$f = f(x_1, x_2, \cdots, x_n)$$

the simplest way is to **construct** the chain rule ($\frac{\partial f}{\partial t}$ is actually meaningless in this context)

$$\frac{\partial f}{\partial t} = \sum_{j=1}^{n} \frac{\partial f}{\partial x_j} \frac{\partial x_j}{\partial t}$$

by setting

$$\frac{\partial x_j}{\partial t} = \begin{cases} 1 & j = i \\ 0 & j \neq i \end{cases}$$

we get

$$\frac{\partial f}{\partial t} = \frac{\partial f}{\partial x_i}$$

- That is to say, for each **decomposed simple operation**, the computer does not only calculate the **output**, but also tracks the **derivatives of output respect to time** based on the input value and input derivative, thus **step by step** getting the final derivative respect to time, which is equal to the partial derivative
- Since the computation is **straight forward**, this method to calculate partial derivative is called **forward mode**.
- For example, for a function

$$f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2)$$

Decompose the function into simple operation nodes

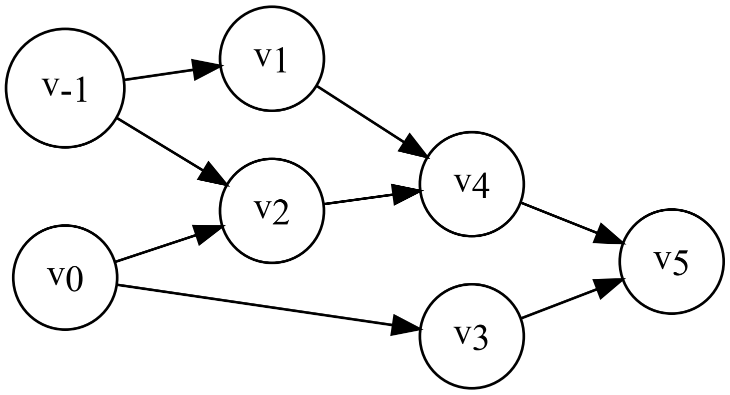
$$\begin{aligned} v_{-1} &= x_1 \\ v_0 &= x_2 \\ v_1 &= \ln v_{-1} \\ v_2 &= v_{-1} \times v_0 \\ v_3 &= \sin v_0 \\ v_4 &= v_1 + v_2 \\ v_5 &= v_4 - v_3 \\ y &= v_5 \end{aligned}$$

The forward mode computation graph is shown below

[35]:
```python
from graphviz import Digraph

f = Digraph('ComputationGraph')
f.attr(rankdir='LR')
f.attr('node', shape='circle')
f.node('v_-1', label='<v<sub>-1</sub>>')
f.node('v_0', label='<v<sub>0</sub>>')
f.node('v_1', label='<v<sub>1</sub>>')
f.node('v_2', label='<v<sub>2</sub>>')
f.node('v_3', label='<v<sub>3</sub>>')
f.node('v_4', label='<v<sub>4</sub>>')
f.node('v_5', label='<v<sub>5</sub>>')
f.edge('v_-1', 'v_1')
f.edge('v_-1', 'v_2')
f.edge('v_0', 'v_2')
f.edge('v_0', 'v_3')
f.edge('v_1', 'v_4')
f.edge('v_2', 'v_4')
```

```
f.edge('v_4', 'v_5')
f.edge('v_3', 'v_5')
f
```

[35]:



- To find the **partial derivate** $\frac{\partial f}{\partial x_1}$ when $x_1 = 2, x_2 = 5$, compute along the **forward evaluation trace** first:

$$\begin{aligned} v_{-1} &= & x_1 &= & 2 \\ v_0 &= & x_2 &= & 5 \\ v_1 &= & \ln v_{-1} &= & \ln 2 \\ v_2 &= & v_{-1} \times v_0 &= & 2 \times 5 \\ v_3 &= & \sin v_0 &= & \sin 5 \\ v_4 &= & v_1 + v_2 &= & 0.693 + 10 \\ v_5 &= & v_4 - v_3 &= & 10.693 + 0.959 \\ y &= & v_5 &= & 11.652 \end{aligned}$$

Then set $\dot{x_1} = 1, \ \dot{x_2} = 0$ and compute along the **forward derivative trace** (maybe also computed simultaneously along with the forward evaluation trace)

$$\begin{aligned} \dot{v}_{-1} &= & \dot{x}_1 &= & 1 \\ \dot{v}_0 &= & \dot{x}_2 &= & 0 \\ \dot{v}_1 &= & \dot{v}_{-1}/v_{-1} &= & 1/2 \\ \dot{v}_2 &= & \dot{v}_{-1} \times v_0 + \dot{v}_0 \times v_{-1} &= & 1 \times 5 + 0 \times 2 \\ \dot{v}_3 &= & \dot{v}_0 \times \cos v_0 &= & 0 \times \cos 5 \\ \dot{v}_4 &= & \dot{v}_1 + \dot{v}_2 &= & 0.5 + 5 \\ \dot{v}_5 &= & \dot{v}_4 - \dot{v}_3 &= & 5.5 - 0 \\ \dot{y} &= & \dot{v}_5 &= & 5.5 \end{aligned}$$

Therefore, the result is

$$\frac{\partial f}{\partial x_1} = \frac{\partial f}{\partial t} = 5.5$$

If $\frac{\partial f}{\partial x_2}$ is also interested, the **forward derivative trace** needs to be computed again when setting $\dot{x}_1 = 0, \dot{x}_2 = 1$

- **Comments**:
  - For a function $f : \mathbb{R}^n \Rightarrow \mathbb{R}^m$ with input $\mathbf{x} \in \mathbb{R}^n$ and output $\mathbf{y} \in \mathbb{R}^m$, needs to calculate n times along the **forward derivative trace** for the partial derivatives

    $$\frac{\partial \mathbf{y}}{\partial x_j},\ j = 1, 2, \cdots, n$$

    by sequentially setting

    $$\frac{\partial x_j}{\partial t} = \begin{cases} 1 & j = i \\ 0 & j \neq i \end{cases},\ i = 1, 2, \cdots, n$$

    and only one time along the **forward evaluation trace** for the m-dimension output values

    $$y_1, y_2, \cdots, y_m$$
  - Get the partial derivatives of **all outputs** respect to **one variable** after computing along the derivative trace once
  - **High efficiency** when $n << m$
  - **Low efficiency** when $n >> m$, which is common in machine learning and deep learning

4.2.3 Automatic differentiation - Reverse Mode

- Motivated by the **low efficiency** of *forward mode* when the derivatives of multiple variables are interested
  - In machine learning and deep learning, usually the dimension of input parameters are huge (such as 10^6) while the dimension of output is usually 1 (focusing on the loss function which produce a scalar)
- To find the partial derivative respect to i-th variable $\frac{\partial f}{\partial x_i}$ in the function

  $$f = f(x_1, x_2, \cdots, x_n)$$

  we can always decompose the function into simple operation nodes

  $$v_1, v_2, \cdots, v_p$$

  such that we can find some nodes $v_c, c \in [1, M]$ to compose the derivative

  $$\frac{\partial y}{\partial x_i} = \frac{\partial y}{\partial v_{c_q}} \frac{\partial v_{c_q}}{\partial v_{c_{q-1}}} \cdots \frac{\partial v_{c_2}}{\partial v_{c_1}} \frac{\partial v_{c_1}}{\partial x_i}$$

  Denote

  $$\bar{v}_c = \frac{\partial y}{\partial v_c}$$

  The idea is to first compute following the **forward evaluation trace** same as that in *forward mode* while recording the **relationship** (such as the differential equation) between nodes, then sequentially calculate

  $$\bar{v}_{c_q} = \frac{\partial y}{\partial v_{c_q}}, \bar{v}_{c_{q-1}} = \bar{v}_{c_q} \frac{\partial v_{c_q}}{\partial v_{c_{q-1}}}, \cdots, \frac{\partial f}{\partial x_i} = \bar{v}_{c_1} \frac{\partial v_{c_1}}{\partial x}$$

- The example is still to find the partial derivatives at $x_1 = 2, x_2 = 5$ of the function

$$f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2)$$

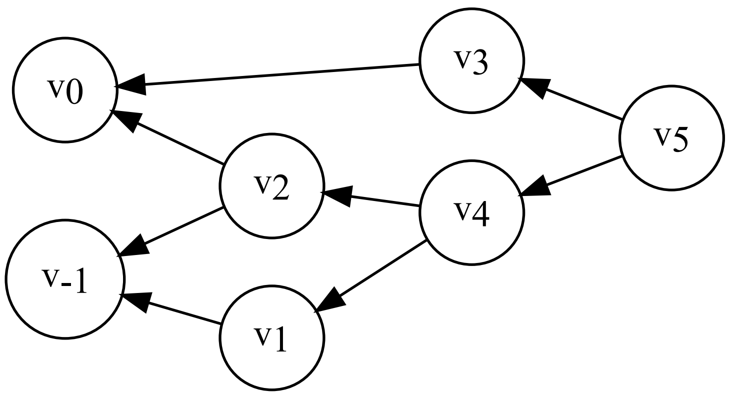
Similarly, decompose the function into simple operation nodes

$$\begin{aligned} v_{-1} &= x_1 \\ v_0 &= x_2 \\ v_1 &= \ln v_{-1} \\ v_2 &= v_{-1} \times v_0 \\ v_3 &= \sin v_0 \\ v_4 &= v_1 + v_2 \\ v_5 &= v_4 - v_3 \\ y &= v_5 \end{aligned}$$

The **forward evaluation computation graph** would be the same of that in *forward mode*. Here show the **reverse adjoint computation graph**

[36]:
```
f = Digraph('ComputationGraph')
f.attr(rankdir='RL')
f.attr('node', shape='circle')
f.node('v_-1', label='<v<sub>-1</sub>>')
f.node('v_0', label='<v<sub>0</sub>>')
f.node('v_1', label='<v<sub>1</sub>>')
f.node('v_2', label='<v<sub>2</sub>>')
f.node('v_3', label='<v<sub>3</sub>>')
f.node('v_4', label='<v<sub>4</sub>>')
f.node('v_5', label='<v<sub>5</sub>>')
f.edge('v_1', 'v_-1')
f.edge('v_2', 'v_-1')
f.edge('v_2', 'v_0')
f.edge('v_3', 'v_0')
f.edge('v_4', 'v_1')
f.edge('v_4', 'v_2')
f.edge('v_5', 'v_4')
f.edge('v_5', 'v_3')
f
```

[36]:



- First, compute along the **forward evaluation trace**

$$\begin{aligned}
v_{-1} &= & x_1 &= & 2\\
v_0 &= & x_2 &= & 5\\
v_1 &= & \ln v_{-1} &= & \ln 2\\
v_2 &= & v_{-1}\times v_0 &= & 2\times 5\\
v_3 &= & \sin v_0 &= & \sin 5\\
v_4 &= & v_1+v_2 &= & 0.693+10\\
v_5 &= & v_4-v_3 &= & 10.693+0.959\\
y &= & v_5 &= & 11.652
\end{aligned}$$

Then compute along the **reverse adjoint trace**

$$\begin{aligned}
\bar{v}_5 &= & \frac{\partial y}{\partial v_5} &= & 1 & \\
\bar{v}_4 &= & \bar{v}_5\frac{\partial v_5}{\partial v_4} &= & \bar{v}_5\times 1 = & \quad 1\\
\bar{v}_3 &= & \bar{v}_5\frac{\partial v_5}{\partial v_3} &= & \bar{v}_5\times(-1) = & \quad -1\\
\bar{v}_1 &= & \bar{v}_4\frac{\partial v_4}{\partial v_1} &= & \bar{v}_4\times 1 = & \quad 1\\
\bar{v}_2 &= & \bar{v}_4\frac{\partial v_4}{\partial v_2} &= & \bar{v}_4\times 1 = & \quad 1\\
\bar{v}_0 &= & \bar{v}_2\frac{\partial v_2}{\partial v_0}+\bar{v}_3\frac{\partial v_3}{\partial v_0} &= & \bar{v}_2\times v_{-1}+\bar{v}_3\times\cos v_0 = & \quad 1.716\\
\bar{v}_{-1} &= & \bar{v}_1\frac{\partial v_1}{\partial v_{-1}}+\bar{v}_2\frac{\partial v_2}{\partial v_{-1}} &= & \bar{v}_1\times\frac{1}{v_{-1}}+\bar{v}_2\times v_0 = & \quad 5.5
\end{aligned}$$

Therefore, the result is

$$\begin{aligned}
\frac{\partial f}{\partial x_1} &= & \bar{v}_{-1} = & \quad 5.5\\
\frac{\partial f}{\partial x_2} &= & \bar{v}_0 = & \quad 1.716
\end{aligned}$$

- **Comments**:
  - For a function $f : \mathbb{R}^n \Rightarrow \mathbb{R}^m$ with input $\mathbf{x} \in \mathbb{R}^n$ and output $\mathbf{y} \in \mathbb{R}^m$, needs to calculate m times along the **reverse adjoint trace**
  - Get the partial derivatives of **one output** respect to **all variables** after computing along the reverse adjoint trace once
    - Fits the idea of calculating **gradients**
  - **High efficiency** when $n >> m$
  - **Low efficiency** when $n << m$
  - Frequently used in ML/DL tasks and is supported by PyTorch, Tensorflow, ...

4.2.4 Computation graph and AutoGrad in PyTorch

- As mentioned above, PyTorch supports **reverse mode automatic differentiation**
  - PyTorch automatically creates a computation graph if **requires_grad=True**
  - For a given variable with **requires_grad=True**, identify the operations and record the required information for reverse adjoint trace computation
  - Can be checked by viewing the **requires_grad** attributes of independent variables
  - Use the **make_dots** method from **torchviz** module to visualize the computation graph
    - Need support from **graphviz**
- Some comments:
  - PyTorch's computational graphs are **dynamic**, in the sense that a new graph is created in each forward pass. On the other hand, the computational graphs constructed by Tensorflow are **static**, in the sense that the same graph is used over and over in all iterations during training
  - In general, static graphs are more efficient because it only needs to be optimized once. Optimization generally consists of distributing the computations over the graph nodes across multiple GPUs if more than one GPU is available, or just fusing some nodes of the graph if the result logic won't be impacted by such fusion
  - However, static graphs do not lend themselves well to recurrent neural computations because the graph itself can change from iteration to iteration

[37]:
```
# The default setting is requires_grad = False
x = torch.tensor(5.)
y = 3*x**2+x
print(x,x.requires_grad)
print(y,y.requires_grad)
```

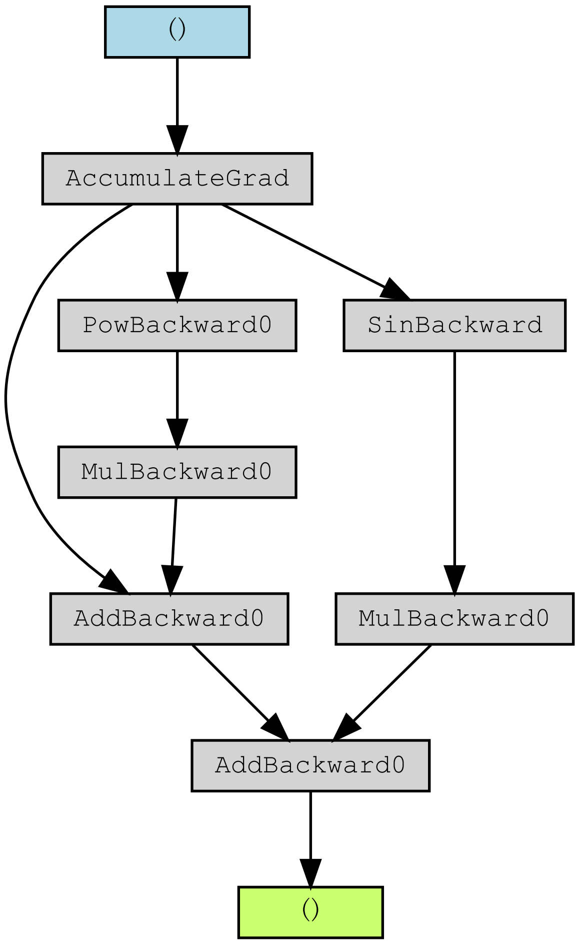
```
tensor(5.) False
tensor(80.) False
```

[38]:
```
# Use the torchviz module for visualization
from torchviz import make_dot
x = torch.tensor(5., requires_grad=True)
y = 3*x**2+x+4*torch.sin(x)
# Independent variable would show the requires_grad attribute
print(x,x.requires_grad)
# Dependent variable would show the grad_fn attribute
print(y,y.requires_grad)
```

```
make_dot(y)
```

```
tensor(5., requires_grad=True) True
tensor(76.1643, grad_fn=<AddBackward0>) True
```

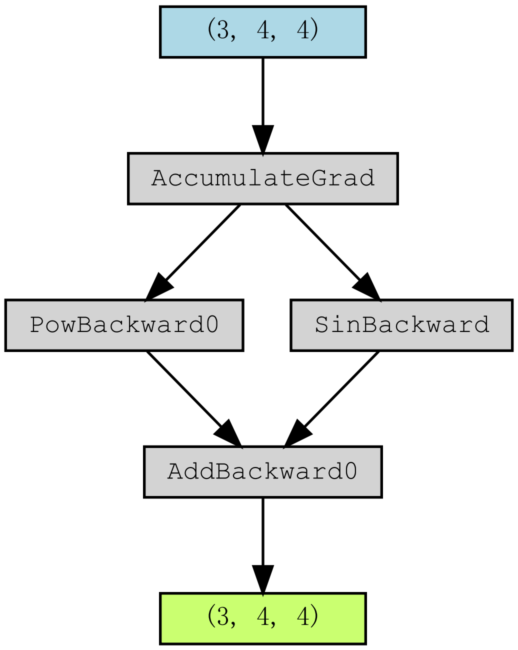
[38]:



- Notice the **grad__fn** attribute. Tensor use this attribute to do the backwards computation.
- When the dimension of input tensor is greater than one, the dimension of each step would also show on the graph
  - Very useful when building neural networks

[39]:
```
x = torch.randn((3,4,4),requires_grad=True)
y = x**2 + torch.sin(x)
make_dot(y)
```

[39]:



- One can access the gradient by calling **backward()** method
- After invoke **backward()** method on the dependent variable we are interested in, the partial derivative would automatically be returned to the **grad** attribute of variables contributed to that dependent variable
- For **independent** variables, we can directly call **backward()** method and see the gradients
- In the following case:

$$z = x_1^2 + x_2^2$$

  The partial derivatives at $x_1 = 5, x_2 = 2$ are

$$\frac{\partial z}{\partial x_1} = 2x_1 = 10, \frac{\partial z}{\partial x_2} = 2x_2 = 4$$

[40]:
```
x_1 = torch.tensor(5.,requires_grad=True)
x_2 = torch.tensor(2.,requires_grad=True)
z = x_1**2 + x_2**2
z.backward()
print(x_1,x_1.grad)
print(x_2,x_2.grad)
print(z)
```

```
tensor(5., requires_grad=True) tensor(10.)
tensor(2., requires_grad=True) tensor(4.)
tensor(29., grad_fn=<AddBackward0>)
```

- For **intermediate dependent** variables (*non-leaf tensor* in pytorch documentation) used in computation, the gradient value is usually not considered and would be cleared after the computation along reverse adjoint trace.

- If the value is indeed needed, the **retain_grad()** method should be invoked

[41]:
```
x_1 = torch.tensor(5.,requires_grad=True)
x_2 = torch.tensor(2.,requires_grad=True)
y = 0.5*x_2**2
y.retain_grad()
z = x_1**2 + y
z.backward()
print(x_1,x_1.grad)
print(x_2,x_2.grad)
print(y,y.grad)
print(z)
```

```
tensor(5., requires_grad=True) tensor(10.)
tensor(2., requires_grad=True) tensor(2.)
tensor(2., grad_fn=<MulBackward0>) tensor(1.)
tensor(27., grad_fn=<AddBackward0>)
```

- It needs to be noticed that the gradients accumulate when calling **backward()**

[42]:
```
x = torch.tensor(5.,requires_grad=True)
for ii in range(2):
    y = 3*x**2
    y.backward()
    print(x,x.grad)
    print(y)
```

```
tensor(5., requires_grad=True) tensor(30.)
tensor(75., grad_fn=<MulBackward0>)
tensor(5., requires_grad=True) tensor(60.)
tensor(75., grad_fn=<MulBackward0>)
```

- Therefore, when calculating the gradients repeated in loops, we usually need to **zero** the gradients before calling **backward()** method, by calling **zero_()** method

[43]:
```
x = torch.tensor(5.,requires_grad=True)
for ii in range(2):
    try:
        x.grad.zero_()
    except Exception as e:
        print(e)
    y = 3*x**2
    y.backward()
    print(x,x.grad)
    print(y)
```

```
'NoneType' object has no attribute 'zero_'
tensor(5., requires_grad=True) tensor(30.)
tensor(75., grad_fn=<MulBackward0>)
tensor(5., requires_grad=True) tensor(30.)
tensor(75., grad_fn=<MulBackward0>)
```

- Sometimes, we would need to call the **backward()** method multiple times, like using MSE loss and Cross-Entropy loss separately for localization and detection in CV. In this case, **retain_graph** should be set to **True** except in the last **backward()** method if any **intermediate dependent variables** are used.

[44]:
```
x = torch.tensor(5.,requires_grad=True)
y = x**2
z_1 = x + y
z_2 = x**2 + y
z_3 = x**3 + y
z_1.backward(retain_graph=True)
z_2.backward(retain_graph=True)
z_3.backward()
print(x,x.grad)
```

```
tensor(5., requires_grad=True) tensor(116.)
```

- Generally speaking, PyTorch can compute gradients for any number of parameters and any complex functions, as long as the decomposed operation is differentiable

[45]:
```
x = torch.arange(5.).requires_grad_(True)
y = torch.sum(x**2)
y.backward()
print(x)
print(y)
print(x.grad)
```

```
tensor([0., 1., 2., 3., 4.], requires_grad=True)
tensor(30., grad_fn=<SumBackward0>)
tensor([0., 2., 4., 6., 8.])
```

[46]:
```
x = torch.arange(5.).requires_grad_(True)
y = torch.mean(torch.log(x**2+1)+5*x)
y.backward()
print(x)
print(y)
print(x.grad)
```

```
tensor([0., 1., 2., 3., 4.], requires_grad=True)
tensor(11.4877, grad_fn=<MeanBackward0>)
tensor([1.0000, 1.2000, 1.1600, 1.1200, 1.0941])
```

4.3 Simple Gradient Descent example

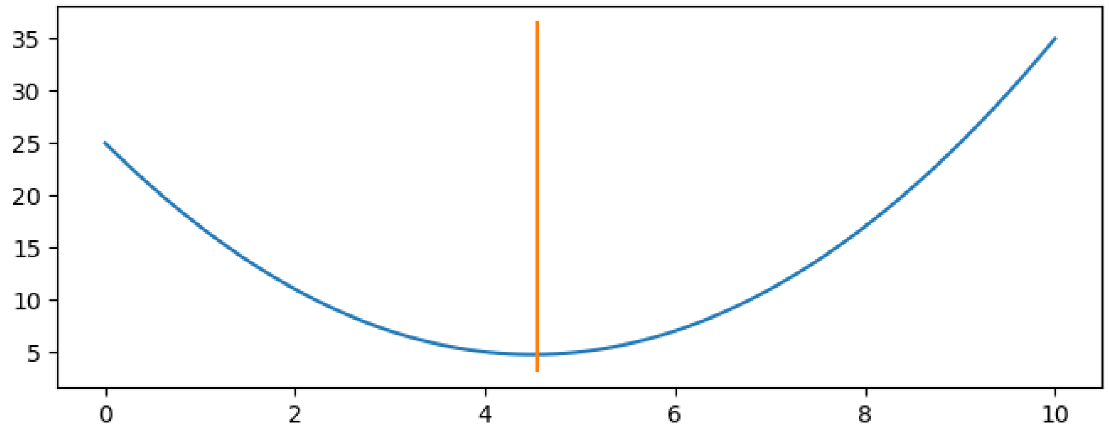
- To show how automatic gradient in PyTorch and the simple gradient descent algorithm work, here are two examples where the input and output are both 1-dimension variables
- For convenience, define the gradient descent algorithm as a function and the visualization process as a function

```
[47]: import matplotlib.pyplot as plt
import numpy as np

# Function of simple gradient descent
def gradient_descent(objective, step_size=0.05, max_iter=100, init=0):
    # Initialize
    x_hat = torch.tensor(init, dtype=torch.float32, requires_grad=True)
    # Record the value iteration process
    x_hat_arr = [x_hat.detach().numpy().copy()]
    obj_arr = [objective(x_hat).detach().numpy()]
    # Iterate
    for ii in range(max_iter):
        # Compute gradient
        if x_hat.grad is not None:
            x_hat.grad.zero_()
        out = objective(x_hat)
        out.backward()
        # Update x_hat
        # Stop tracking gradients
        with torch.no_grad():
            x_hat -= step_size * x_hat.grad
        x_hat_arr.append(x_hat.detach().numpy().copy())
        obj_arr.append(objective(x_hat).detach().numpy())
    return np.array(x_hat_arr), np.array(obj_arr)
# Function to visualize iteration process
def visualize_result(x_arr,obj_arr,objective,x_true=None, vis_arr=None):
    # The horizontal range of figure
    if vis_arr is None:
        vis_arr = np.linspace(np.min(x_arr),np.max(x_arr))
    fig, ax = plt.subplots(figsize=(8,3),dpi=100)
    ax.plot(vis_arr,[objective(torch.tensor(x)).numpy() for x in vis_arr],␣
 ↪label='Objective')
    ax.plot(x_arr,obj_arr,'o-', label='Gradient Steps')
    # If true minimum location is provided, plot it out
    if x_true is not None:
        ax.plot(np.ones(2)*x_true, plt.ylim(), label='True x')
    ax.plot(np.ones(2)*x_arr[-1], plt.ylim(),label='Final x')
    plt.legend(loc='lower right')
    plt.show()
```

4.3.1 Convex function - Converge to global minimum

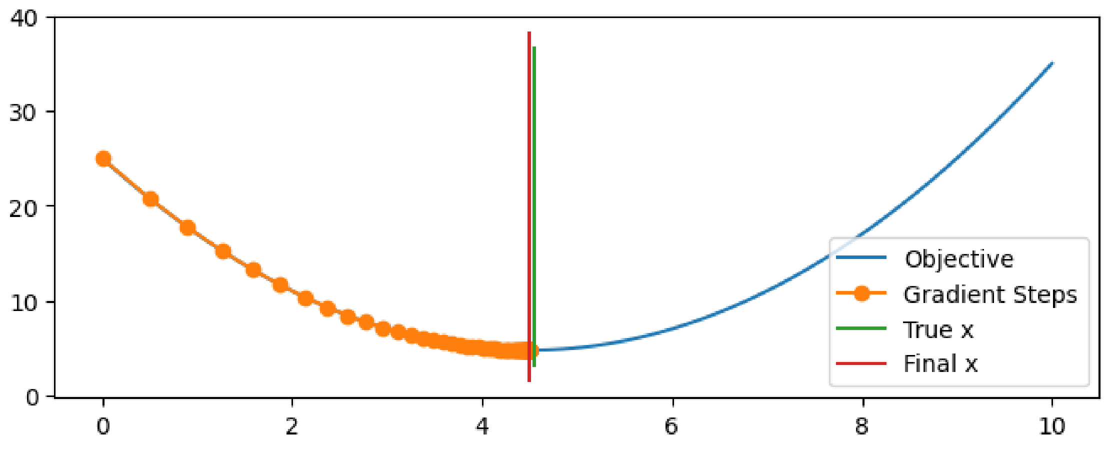
- Here is an example where the objective function is very simple and gradient descent can easily reach the global minimum
- First plot out the objective function

```
[48]: # Objective function
      def Objective(x):
          return torch.abs(x)+(x-5)**2


      x = torch.linspace(0,10,100)
      y = Objective(x)
      x_true = float(x[np.argmin(y)])
      fig, ax = plt.subplots(figsize=(8,3),dpi=100)
      # Need to convert the tensor to numpy arrays for plots
      ax.plot(x.numpy(),y.numpy())
      # Show the minimum location
      ax.plot(x_true*np.ones(2),plt.ylim())
      plt.show()
```



- Apply simple gradient descent on this function

```
[49]: # Call the functions to get the minimum
      x_hat_arr, obj_arr = gradient_descent(Objective, step_size=0.05, max_iter=150,␣
       ↪init=0)
      visualize_result(x_hat_arr, obj_arr,Objective, x_true=x_true, vis_arr=np.
       ↪linspace(0,10,num=100))
```



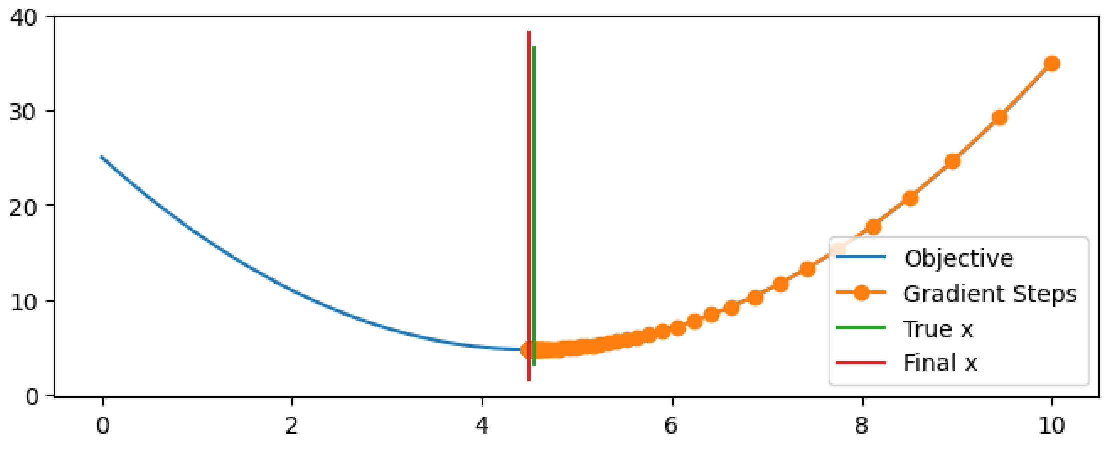
- It seems that the final value approximately reaches the global minimum. Let's check the result by printing them out

[50]:
```
print("The true minimum location is:              %f"%(x_true))
print("The final location got by GD is:           %f"%(x_hat_arr[-1]))
print("Errors between true and final location is: %f"%(x_true-x_hat_arr[-1]))
```

```
The true minimum location is:              4.545455
The final location got by GD is:           4.499998
Errors between true and final location is: 0.045456
```

- In this case, the start point does **not** matter

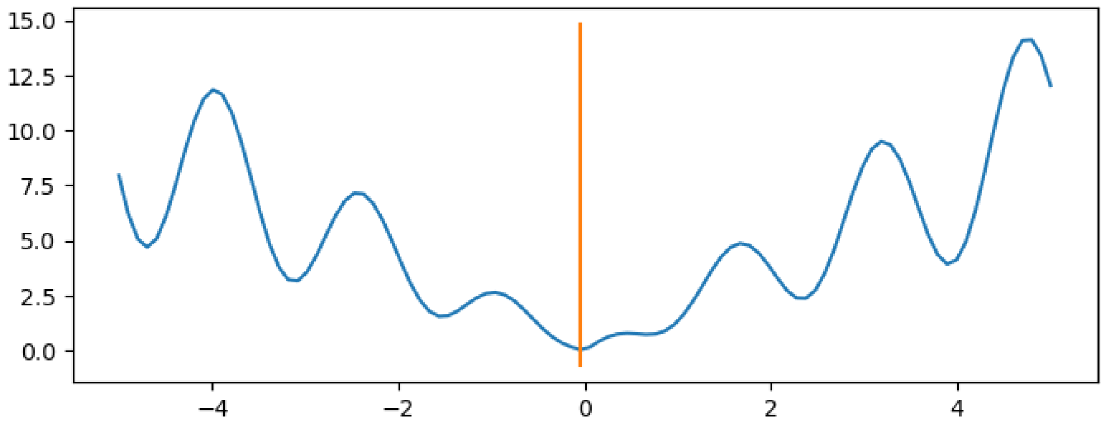
[51]:
```
# Call the functions to get the minimum
x_hat_arr, obj_arr = gradient_descent(Objective, step_size=0.05, max_iter=150,␣
 ↪init=10)
visualize_result(x_hat_arr, obj_arr,Objective, x_true=x_true, vis_arr=np.
 ↪linspace(0,10,num=100))
```



4.3.2 Non convex function - Converge to local minimum

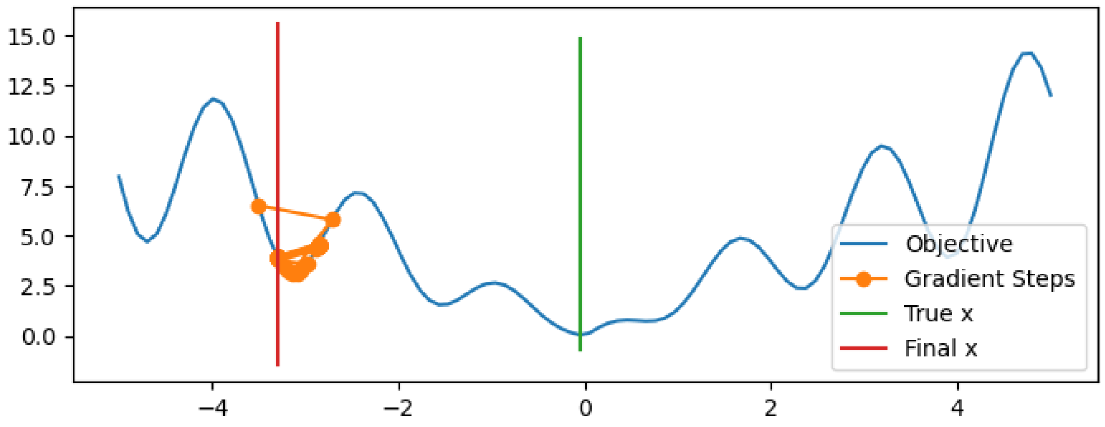
- Here is an example where the objective function is a bit complex and gradient descent falls into local minimum
- First plot out the objective function

[52]:
```
def Objective(x):
    return x*torch.cos(4*x) + 2*torch.abs(x)

x = torch.linspace(-5,5,100)
y = Objective(x)
x_true=float(x[np.argmin(y)])
fig, ax = plt.subplots(figsize=(8,3),dpi=100)
ax.plot(x.numpy(),y.numpy())
ax.plot(x_true*np.ones(2),plt.ylim())
plt.show()
```



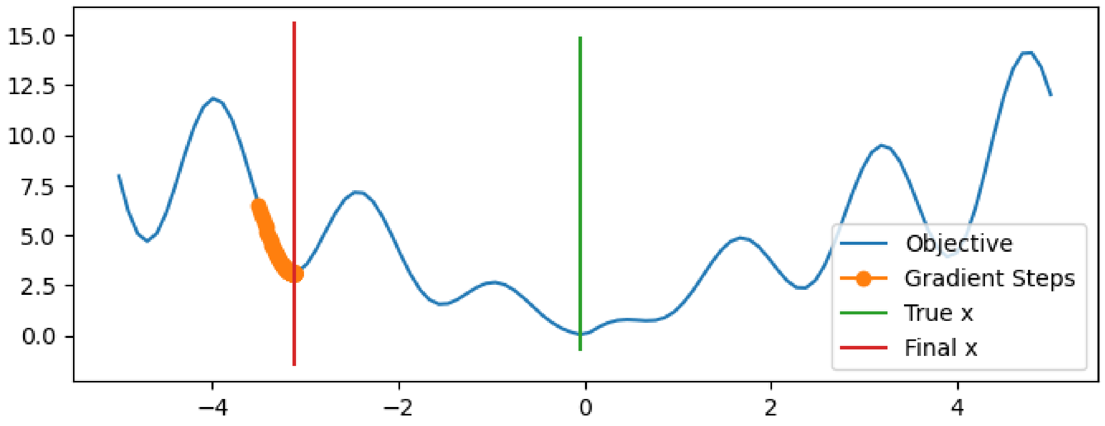
- As what was done before, apply simple gradient descent to it

[53]:
```
x_hat_arr, obj_arr = gradient_descent(Objective, step_size=0.05, max_iter=150,␣
 ↪init=-3.5)
visualize_result(x_hat_arr, obj_arr,Objective, x_true=x_true, vis_arr=np.
 ↪linspace(-5,5,num=100))
```



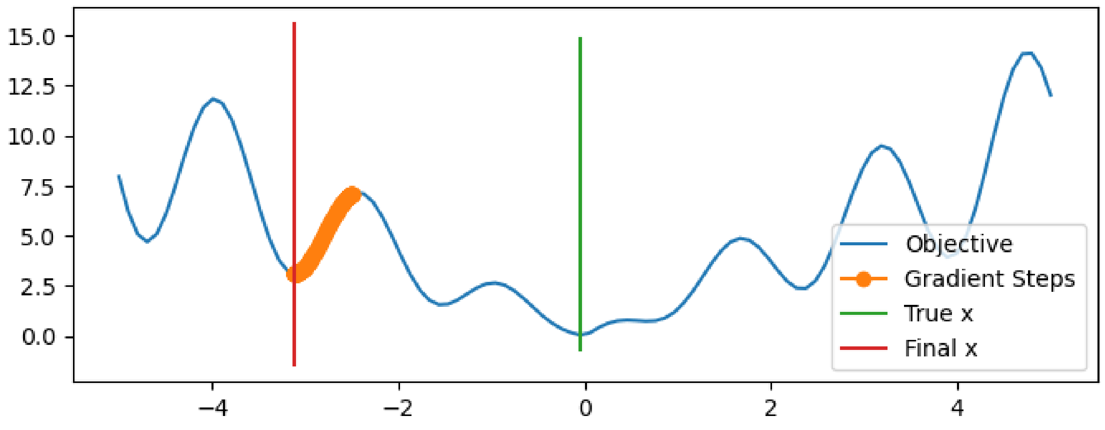
- In this case, it seems that the **step size** is so huge that the oscillation occurs and gradient descent even cannot reach a local minimum. Try with a smaller step size

[54]:
```
x_hat_arr, obj_arr = gradient_descent(Objective, step_size=0.001, max_iter=150,␣
↪init=-3.5)
visualize_result(x_hat_arr, obj_arr,Objective, x_true=x_true, vis_arr=np.
↪linspace(-5,5,num=100))
```

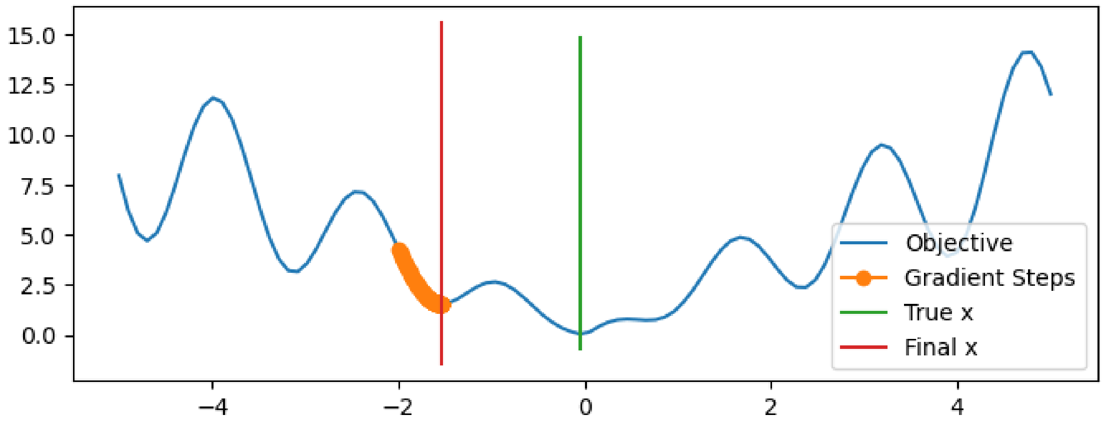


- With a smaller step size, it successfully converge to a local minimum. However, we would always look forward to a global minimum.
- In this case, the **start point will affect final convergence**. Try with different start points

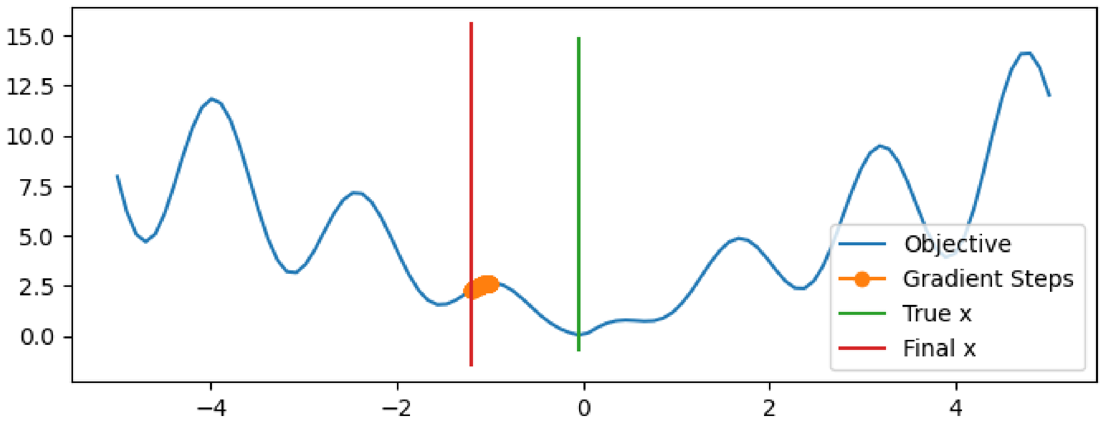
[55]:
```
x_hat_arr, obj_arr = gradient_descent(Objective, step_size=0.001, max_iter=150,␣
↪init=-2.5)
visualize_result(x_hat_arr, obj_arr,Objective, x_true=x_true, vis_arr=np.
↪linspace(-5,5,num=100))
```



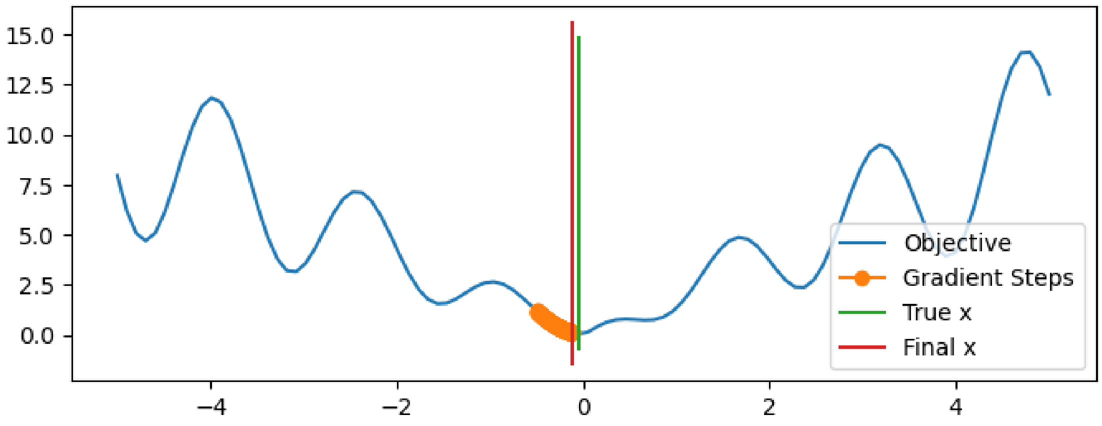
[56]:
```
x_hat_arr, obj_arr = gradient_descent(Objective, step_size=0.001, max_iter=150,␣
 ↪init=-2)
visualize_result(x_hat_arr, obj_arr,Objective, x_true=x_true, vis_arr=np.
 ↪linspace(-5,5,num=100))
```



[57]:
```
x_hat_arr, obj_arr = gradient_descent(Objective, step_size=0.001, max_iter=150,␣
 ↪init=-1)
visualize_result(x_hat_arr, obj_arr,Objective, x_true=x_true, vis_arr=np.
 ↪linspace(-5,5,num=100))
```



```
[58]: x_hat_arr, obj_arr = gradient_descent(Objective, step_size=0.001, max_iter=150,␣
       ↪init=-0.5)
      visualize_result(x_hat_arr, obj_arr,Objective, x_true=x_true, vis_arr=np.
       ↪linspace(-5,5,num=100))
```



- After several trials, we finally reaches the approximated location of global minimum. However, in complex optimization problems, we will not be able to select a best start point so easily by visualization, which **makes finding the global minimum very difficult**.