04 Numerical Optimization

June 3, 2021

Information: *Brief introduction to convexity, optimization, and gradient descent*

Written by: *Zihao Xu*

Last update date: *06.03.2021*

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 $0 \leq \theta \leq 1$:
$$\theta x + (1 - \theta) y \in C$$
  - 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

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

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

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

  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 Karush-Kuhn-Tucker (KKT) approach

- **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
- **Generalized Lagrangian**
  - Also called **generalized Lagrange function**
  - Describe $\mathbb{S}$ in terms of m functions g_i and n functions h_j:

$$\mathbb{S} = \{\mathbf{x} | \forall i, g_i(\mathbf{x}) = 0 \text{ and } \forall j, h_j(\mathbf{x}) \leq 0\}$$

Equations involving g_i are called the **equality constraints** and the inequalities involving h_j are called **inequality constraints**

  - Introduce new variables λ_i and α_i for each constraint, which are called the **KKT multipliers**, then the generalized Lagrangian is then defined as

$$L(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\alpha}) = f(\mathbf{x}) + \sum_{i=1}^{m} \lambda_i g_i(\mathbf{x}) + \sum_{j}^{n} \alpha_j h_j(\mathbf{x})$$

  - The generalized Lagrangian enables us to solve a constrained minimization problem using unconstrained optimization of the generalized Lagrangian. As long as at least one feasible point exists and $f(\mathbf{x})$ is not permitted to have value ∞, then

$$\min_{\mathbf{x}} \max_{\boldsymbol{\lambda}} \max_{\boldsymbol{\alpha}, \boldsymbol{\alpha} \geq 0} L(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\alpha})$$

    has the same optimal objective function value ans set of optimal points $\mathbf{x}$ as

$$\min_{\mathbf{x} \in \mathbb{S}} f(x)$$

  - Any time the constraints are satisfied

$$\max_{\boldsymbol{\lambda}} \max_{\boldsymbol{\alpha}, \boldsymbol{\alpha} \geq 0} L(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\alpha}) = f(\mathbf{x})$$

    while any time a constraint is violated

$$\max_{\boldsymbol{\lambda}} \max_{\boldsymbol{\alpha}, \boldsymbol{\alpha} \geq 0} L(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\alpha}) = \infty$$

    These properties guarantee that no infeasible point can be optimal, and that the optimum within the feasible points is unchanged

- **KKT Conditions**:
  - A simple set of properties describe the optimal properties of constrained optimization problems
  - **Necessary** conditions but **not always sufficient** conditions
  - The conditions are:
    - ∗ The gradient of the generalized Lagrangian is zero
    - ∗ All constraints on both $\mathbf{x}$ and the KKT multipliers are satisfied
    - ∗ The inequality constraints exhibit "complementary slackness": $\boldsymbol{\alpha} \odot \mathbf{h}(\mathbf{x}) = \mathbf{0}$

2.5.3 Example: Linear Least Squares

To find the value of $\mathbf{x}$ that minimizes

$$f(\mathbf{x}) = \frac{1}{2} \|\mathbf{A}\mathbf{x} - \mathbf{b}\|_2^2$$

subject to the constraint $\mathbf{x}^T\mathbf{x} \leq 1$, introduce the Lagrangian

$$L(\mathbf{x}, \lambda) = f(\mathbf{x}) + \lambda(\mathbf{x}^T\mathbf{x} - 1)$$

then solve the problem

$$\min_{\mathbf{x}} \max_{\lambda, \lambda \geq 0} L(\mathbf{x}, \lambda)$$

By differentiating the Lagrangian with respect to $\mathbf{x}$, obtain

$$\mathbf{A}^T\mathbf{A}\mathbf{x} - \mathbf{A}^T\mathbf{b} + 2\lambda\mathbf{x} = 0$$

which means the solution would take the form

$$\mathbf{x} = (\mathbf{A}^T\mathbf{A} + 2\lambda\mathbf{I})^{-1}\mathbf{A}^T\mathbf{b}$$

To decide the magnitude of λ which makes the result obeys the constraint, perform gradient ascent on λ:

$$\frac{\partial}{\partial\lambda}L(\mathbf{x}, \lambda) = \mathbf{x}^T\mathbf{x} - 1$$

When the norm of $\mathbf{x}$ exceeds 1, the derivative is positive and we need to increase λ. Because the coefficient on the $\mathbf{x}^T\mathbf{x}$ penalty has increased, solving the linear equation for $\mathbf{x}$ will now yield a solution with smaller norm. Repeat the process of solving the linear equation and adjusting λ continues until $\mathbf{x}$ has the correct norm and the derivative on λ is 0

3 Gradient Descent

3.1 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

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

    for a $\eta \in \mathbb{R}_+$ 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
     - $\mathbf{d} \leftarrow -\nabla L(\boldsymbol{\theta})$
     - $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta \mathbf{d}^T$
  - η is called **learning rate** or **step size**

3.2 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**

3.2.1 Select appropriate learning rate

- Too large η leads to instability and even divergence
- Too small η leads to slow convergence
- **Steepest gradient descent** use **line search** to compute the best η
  1. Start at random parameter $\boldsymbol{\theta}$
  2. Repeat until converged
     - $\mathbf{d} \leftarrow -\nabla L(\boldsymbol{\theta})$
     - $\eta^* \leftarrow \underset{\eta}{\operatorname{argmin}}\{L(\boldsymbol{\theta} + \eta \mathbf{d}^T)\}$
     - $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta^* \mathbf{d}^T$
- **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

3.3 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

3.4 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 Stochastic Gradient Descent

4.1 Motivation

In Machine Learning and Deep Learning, the objective function is usually the average of the loss functions for each example in the training dataset. Given a training dataset of n examples, we assume that $f_i(\mathbf{x})$ is the loss function with respect to the training example of index i, where $\mathbf{x}$ is the parameter vector. The objective function is

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

Then the gradient of the objective function at $\mathbf{x}$ is computed as

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

It's obvious that when the training dataset is larger and larger, the computation cost for one iteration would become higher and higher, which leads to slow convergence

4.2 Stochastic Gradient Updates

At each iteration, uniformly sample an index $i \in \{1, 2, \cdots, n\}$ for data examples at random, and compute the gradient $\nabla f_i(\mathbf{x})$ to update $\mathbf{x}$:

$$\mathbf{x} \leftarrow \mathbf{x} - \eta \nabla f_i(\mathbf{x})$$

where α is the learning rate

4.2.1 Advantages

- Reduce the computation cost for each iteration from $\mathcal{O}(n)$ to $\mathcal{O}(1)$
- $\nabla f_i(\mathbf{x})$ is an unbiased estimate of the full gradient $\nabla f(\mathbf{x})$:

  $$\mathbb{E}_i[\nabla f_i(\mathbf{x})] = \frac{1}{n} \sum_{i=1}^{n} \nabla f_i(\mathbf{x}) = \nabla f(\mathbf{x})$$

  which means the stochastic gradient is a good estimate of the gradient on average

4.2.2 Disadvantages

- Usually the trajectory of the variables in SGD would be much more noisy than the one observed in gradient descent, due to the stochastic nature of the gradient
- Even when we arrive near the minimum, we are still subject to the uncertainty injected by the instantaneous gradient via $\eta \nabla f_i(\mathbf{x})$ and the performance usually won't improve
- No good choice of step size ⇒ *Dynamic Learning Rate*
  - Large step size always make us hanging around the minimum
  - Small step size prevents any meaningful progress initially

4.3 Dynamic Learning Rate

As mentioned above, we usually want a large step size at the beginning to accelerate the convergence and a small step size near minimum to get better performance. The intuition is to replace η with a time-dependent learning rate $\eta(t)$, which requires to figure out how rapidly η should decay. Decaying too quickly make us stop optimization prematurely and decaying to slowly will waste too much time on optimization. Several basic strategies that are used in adjusting η over time is:

- *Piecewise constant*: $\eta(t) = \eta_i$ if $t_i \leq t \leq t_{i+1}$. Common strategy for training deep networks
- *Exponential decay*: $\eta(t) = \eta_0 \cdot e^{-\lambda t}$. Often leads to premature stopping before the algorithm has converged
- *Polynomial decay*: $\eta(t) = \eta_0 \cdot (\beta t + 1)^{-\alpha}$ A popular choice is $\alpha = 0.5$

4.4 Stochastic Gradients and Finite Samples

- Usually, the we do not perform exactly **stochastic** gradient descent
- Instead of actually randomly select an instance from the training set with replacement, we iterated over all instances **exactly once**
- Consider sampling n observations from the discrete distribution with *replacement*. The probability of choosing an element i at random is $\frac{1}{n}$. Denote the probability of choose it x times in n samples is $P(X = x)$, then the probability of picking some sample **at least once** is

  $$P(X \geq 1) = 1 - \left(1 - \frac{1}{n}\right)^n$$

  When $n \to \infty$, it's easy to prove that

  $$\lim_{n\to\infty} P(X \geq 1) = 1 - \frac{1}{e} \approx 0.63$$

  Also consider the probability of picking some sample **exactly once** is

  $$P(X = 1) = C_n^1 \cdot \frac{1}{n}\left(1 - \frac{1}{n}\right)^{n-1}$$

  Similarly,

  $$\lim_{n\to\infty} P(X = 1) = \frac{1}{e} \approx 0.37$$

  That is to say, the sampling with replacement leads to an increased variance and decreased data efficiency relative to sampling without replacement. Therefore, in machine learning and deep learning context, we prefer **sampling without replacement**
- The common way is to iterating over the training dataset in different random orders

5 Minibatch Stochastic Gradient Descent

5.1 Motivation

- Gradient Descent is not particularly **data efficient** whenever data is very similar
- Stochastic Gradient Descent is not particularly **computationally efficient** since CPUs and GPUs cannot exploit the full power of **vectorization**
- There might be a happy medium between **GD** and **SGD**

5.2 Vectorization

- Vectorization is a basic method to increase computational efficiency
  - Due to reduced overhead arising from the deep learning framework and due to better memory locality and caching on CPUs and GPUs
- Use vectorization to replace **loops**, which usually leads high computation cost
- It is highly advisable to use vectorization and matrices whenever possible
- Consider matrix-matrix multiplication $\mathbf{A} = \mathbf{B}\mathbf{C}$. Common methods to compute $\mathbf{A}$ are:
  - $\mathbf{A}_{i,j} = \mathbf{B}_{i,:}\mathbf{C}^T_{:,j}$, compute it element-wise by means of dot product. In this way, we will need to copy one row and one column vector into the CPU each time we want to compute an element $\mathbf{A}_{i,j}$. In the meantime, we are required to access many disjoint locations for one of the two vectors as we read them from memory due to the fact that the matrix elements are aligned sequentially.
  - $\mathbf{A}_{:,j} = \mathbf{B}\mathbf{C}^T_{:,j}$, compute one column at a time. We are able to keep the column vector $\mathbf{C}_{:,j}$ in the CPU cache while we keep on traversing through $\mathbf{B}$
  - $\mathbf{A} = \mathbf{B}\mathbf{C}$ directly. Most desirable, but most matrices might not entirely fit into cache
  - Break $\mathbf{B}$ and $\mathbf{C}$ into smaller block matrices and compute $\mathbf{A}$ one block at a time. Offers a practically useful alternative: move blocks of the matrix into cache and multiply them locally
- Element-wise computation $\mathbf{A}_{i,j} = \mathbf{B}_{i,:}\mathbf{C}^T_{:,j}$

[1]:

```python
import numpy as np
import time
# Construct B,C
A = np.zeros((256, 256))
B = np.random.normal(0, 1, (256, 256))
C = np.random.normal(0, 1, (256, 256))
# Record the start time
start_time = time.time()
# Element-wise computation
for ii in range(256):
    for jj in range(256):
        A[:, jj] = np.dot(B[ii, :], C[:, jj])
# Record the end time
end_time = time.time()
# Show the time cost
print("Element-wise computation: %f ms" % (1000 * (end_time - start_time)))
```

```
Element-wise computation: 298.183203 ms
```

- Perform column-wise computation $\mathbf{A}_{:,j} = \mathbf{B}\mathbf{C}_{:,j}^{T}$ is faster

[2]:

```
# Record the start time
start_time = time.time()
# Column-wise computation
for jj in range(256):
    A[:, jj] = np.dot(B, C[:, jj])
# Record the end time
end_time = time.time()
# Show the time cost
print("Column-wise computation: %f ms" % (1000 * (end_time - start_time)))
```

```
Column-wise computation: 25.441408 ms
```

- The most effective manner is to perform the entire operation in one block $\mathbf{A} = \mathbf{B}\mathbf{C}$

[3]:

```
# Record the start time
start_time = time.time()
# Column-wise computation
A = np.dot(B, C)
# Record the end time
end_time = time.time()
# Show the time cost
print("Direct computation: %f ms" % (1000 * (end_time - start_time)))
```

```
Direct computation: 2.999067 ms
```

5.3 Minibatches

- Processing single observations requires us to perform many single matrix-vector (or even vector-vector) multiplications, which is quite expensive and which incurs a significant overhead on behalf of the underlying deep learning framework. This applies whenever we perform

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta_t \nabla_{\boldsymbol{\theta}} f_{\boldsymbol{\theta}}(\mathbf{x}_t)$$

- Taking the **vectorization** method into consideration, we can increase the **computational efficiency** of this operation by applying it to a minibatch of observations at a time. That is to say, approximate the full gradient by the gradient of a batch of samples

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta_t \sum_{k=1}^{b} \frac{1}{b} \nabla_{\boldsymbol{\theta}} f_{\boldsymbol{\theta}}(\mathbf{x}_k)$$

- Since both $\mathbf{x}_t$ and also all elements of the minibatch are drawn uniformly at random from the training set, the **expectation** of the gradient remains unchanged
- The variance is reduced significantly (relative to SGD)
  - The standard deviation is reduced by a factor of $b^{-\frac{1}{2}}$, which means the updates are more reliably aligned with the full gradient
- Naively it indicates that choosing a large minibatch b would be universally desirable while the additional reduction in standard deviation in minimal after some point when compared to the linear increase in computational cost
  - In practice, usually pick a minibatch that is large enough to offer good computational efficiency while still fitting into the memory of a GPU

5.4 Minibatch SGD Summary

5.4.1 Performance

- In general, minibatch stochastic gradient descent is faster than stochastic gradient descent and gradient descent for convergence to a smaller risk, when measures in terms of clock time
- Balance the trade-off between **statistical efficiency** arising from stochastic gradient descent and **computational efficiency** arising from processing large batches of data at a time

5.4.2 Batch size

- Large batches: less *noise* in gradient
  - Disadvantages: Slower updates per iteration; less exploration
  - Advantages: Better local convergence
- Smaller batches: more *noise* in gradient
  - Disadvantages: Hunts around local minimum
  - Advantages: Faster updates per iteration; better exploration

5.4.3 Patch size

- Many algorithms train on image *patches*
- Smaller patches might fit better into GPU cache
- Whether smaller patches speed training is apocryphal

5.4.4 Learning rate

- Too large learning rate leads to oscillations around local minimum
- Too small learning rate leads to convergence
- It is advisable to use **adaptive learning rate** and decay the learning rates during training

6 Momentum

6.1 Motivation

When performing optimization where only a noisy variant of the gradient is available, we need to be extra cautious when it comes to choosing the learning rate in the face of noise. If we decrease it too rapidly, convergence stalls. If we are too lenient, we fail to converge to a good enough solution since noise keeps on driving us away from optimality

6.2 Leaky Averages

Consider the Minibatch SGD (Often called SGD directly), pay special attention to notations of time step:

$$\boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} - \eta_t \sum_{k=1}^{b} \frac{1}{b} \nabla_{\boldsymbol{\theta}} f(\mathbf{x}_k | \boldsymbol{\theta}_{t-1})$$

Denote the stochastic gradient descent as:

$$\mathbf{g}_{t,t-1} = \sum_{k=1}^{b} \frac{1}{b} \nabla_{\boldsymbol{\theta}} f(\mathbf{x}_k | \boldsymbol{\theta}_{t-1})$$

One option to *benefit from the effect of variance reduction* even beyond averaging gradients on a minibatch is to *replace the gradient computation by a leaky average*

$$\begin{aligned} \mathbf{v}_t &= \beta \mathbf{v}_{t-1} - \eta \mathbf{g}_{t,t-1} \\ \boldsymbol{\theta}_t &= \boldsymbol{\theta}_{t-1} + \mathbf{v}_t \end{aligned}$$

for some $\beta \in (0, 1)$. This effectively replace the instantaneous gradient by one that's been **averaged over multiple past gradients**. In some notations, it's in the equivalent form which might show the *average* process better

$$\begin{aligned} \mathbf{v}_t &= \beta \mathbf{v}_{t-1} + \mathbf{g}_{t,t-1} \\ \boldsymbol{\theta}_t &= \boldsymbol{\theta}_{t-1} - \eta \mathbf{v}_t \end{aligned}$$

- $\mathbf{v}$ is called **momentum**, whereas small β amounts to only a slight correction relative to a gradient method
- The new gradient replacement no longer points into the direction of steepest descent on a particular instance any longer but rather than in the direction of a weighted average of past gradients
- Allows us to realize most of the benefits of *averaging over a batch without the cost of actually computing the gradients on it*
- Accelerates convergence significantly
- Desirable for both noise-free gradient descent and noisy stochastic gradient descent

Above reasoning formed the basis for what is known as **accelerated gradient methods**, such as gradients with momentum.

- Enjoy the additional benefit of being much more effective in cases where the optimization problem is **ill-conditioned** (where there are some directions where progress is much slower that in others, resembling a narrow canyon)
- Allow us to average over subsequent gradients to *obtain more stable directions of descents of descent*

6.3 An ill-conditioned problem

As mentioned above, accelerated gradient methods enjoy the additional benefit of being much more effective in cases where the optimization problem is ill-conditioned. Here is an example to help understand what a ill-conditioned problem exactly is. Consider the problem

$$f(\mathbf{x}) = 0.1x_1^2 + 2x_2^2$$

This function is very *flat* in the direction of x_1:

$$\frac{\partial f}{\partial x_1} = 0.2x_1, \ \frac{\partial f}{\partial x_2} = 4x_2$$

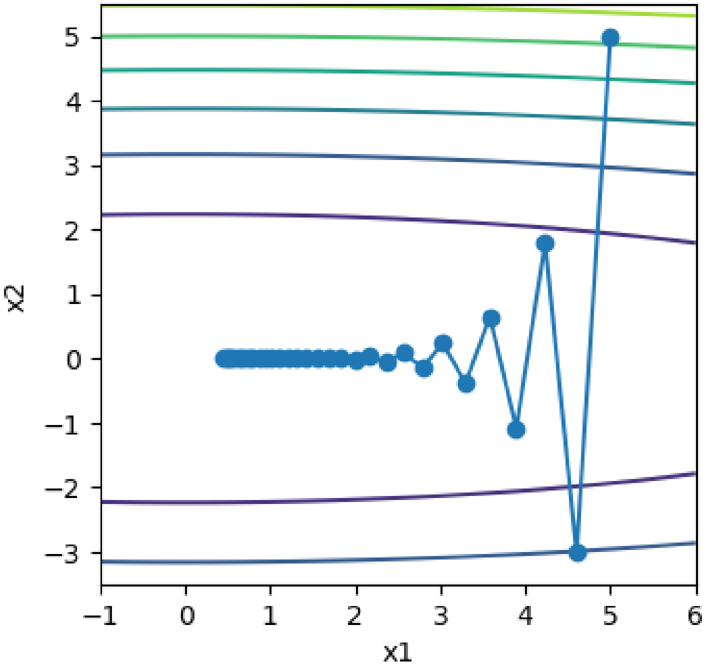
It means that a tiny change in x_2 can lead to great changes in f while changes in x_1 does not matter too much Let's check how GD performs on this function with a learning rate of 0.4. The expected convergence point is $\mathbf{x} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$

[4]:

```
import matplotlib.pyplot as plt

eta = 0.4


def f(x_1, x_2):
    return 0.2 * x_1**2 + 2 * x_2**2


def gd(x_1, x_2):
    return (x_1 - eta * 0.2 * x_1, x_2 - eta * 4 * x_2)


# Number of iteration steps
iter = 30
# Record the iteration trace
x_arr = np.zeros((2, iter))
x_arr[:, 0] = np.array([5, 5])
f_arr = np.zeros((1, iter))
# Iteration
for ii in range(iter - 1):
    f_arr[0, ii] = f(x_arr[0, ii], x_arr[1, ii])
    (x_arr[0, ii + 1], x_arr[1, ii + 1]) = gd(x_arr[0, ii], x_arr[1, ii])
f_arr[0, iter - 1] = f(x_arr[0, iter - 1], x_arr[1, iter - 1])
print("After %d epoches, x1 is %f, x2 is %f" %
      (iter, x_arr[0, iter - 1], x_arr[1, iter - 1]))
# Visualization
fig, ax = plt.subplots(figsize=(4, 4), dpi=100)
# Plot the contour
vis_arr_x = np.linspace(-1, 6, 100)
vis_arr_y = np.linspace(-3.5, 5.5, 100)
vis_X, vis_Y = np.meshgrid(vis_arr_x, vis_arr_y)
```

```
vis_Z = 0.1 * vis_X**2 + 2 * vis_Y**2
ax.contour(vis_X, vis_Y, vis_Z)
ax.plot(x_arr[0, :], x_arr[1, :],'o-')
ax.set_xlabel("x1")
ax.set_ylabel("x2")
plt.show()
```

```
After 30 epoches, x1 is 0.445468, x2 is -0.000002
```

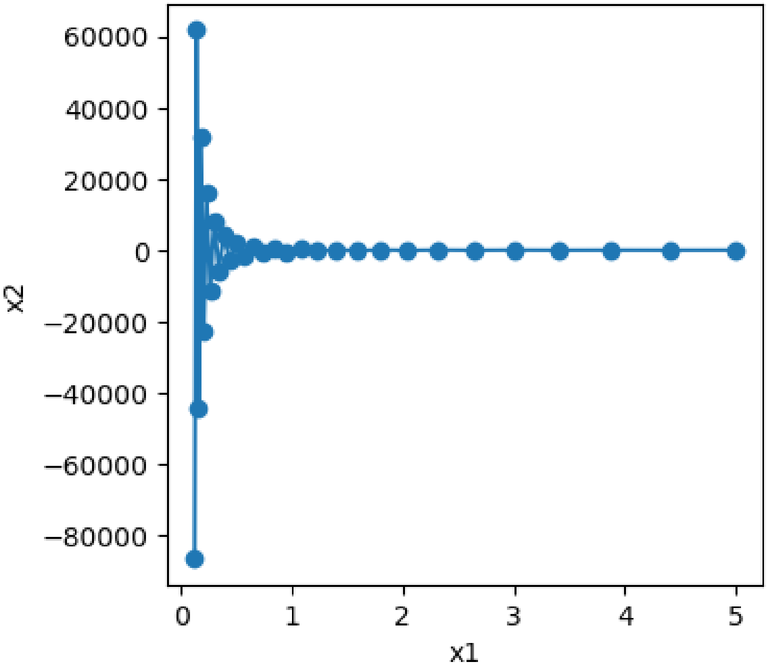


The result indicates that when x_2 has converged to its true value, x_1 is far away its true value and its converging very slowly, due to the gradient in x_2 is *much* higher and change much more rapidly than in the horizontal x_1 direction. If we pick a small learning rate we ensure that the solution does not diverge in the x_2 direction but we are saddled with slow convergence in the x_1 direction. Conversely, with a large learning rate we progress rapidly in the x_1 direction but diverge in x_2. The result of trying with the learning rate of 0.6 is shown below.

[5]:
```
eta = 0.6


def f(x_1, x_2):
    return 0.2 * x_1**2 + 2 * x_2**2


def gd(x_1, x_2):
    return (x_1 - eta * 0.2 * x_1, x_2 - eta * 4 * x_2)
```

```python
# Number of iteration steps
iter = 30
# Record the iteration trace
x_arr = np.zeros((2, iter))
x_arr[:, 0] = np.array([5, 5])
f_arr = np.zeros((1, iter))
# Iteration
for ii in range(iter - 1):
    f_arr[0, ii] = f(x_arr[0, ii], x_arr[1, ii])
    (x_arr[0, ii + 1], x_arr[1, ii + 1]) = gd(x_arr[0, ii], x_arr[1, ii])
f_arr[0, iter - 1] = f(x_arr[0, iter - 1], x_arr[1, iter - 1])
print("After %d epoches, x1 is %f, x2 is %f" %
      (iter, x_arr[0, iter - 1], x_arr[1, iter - 1]))
# Visualization
fig, ax = plt.subplots(figsize=(4, 4), dpi=100)
# Plot the contour
ax.plot(x_arr[0, :], x_arr[1, :],'o-')
ax.set_xlabel("x1")
ax.set_ylabel("x2")
plt.show()
```

```
After 30 epoches, x1 is 0.122735, x2 is -86433.686984
```



6.4 Momentum Method Example

The momentum method allows us to solve the gradient descent problem described above.

- In the x_1 direction, it will aggregate well-aligned gradients, thus increasing the distance we cover with each step
- In the x_2 direction where gradients oscillate, an aggregate gradient will reduce step size due to oscillations that cancel with each other out.

Using $\mathbf{v}_t$ instead of the gradient $\mathbf{g}_t$ yields the following update equations

$$\mathbf{v}_t \leftarrow \beta \mathbf{v}_{t-1} - \eta_t \mathbf{g}_{t,t-1}$$
$$\mathbf{x}_t \leftarrow \mathbf{x}_{t-1} + \mathbf{v}_t$$

For $\beta = 0$ we recover regular gradient descent.

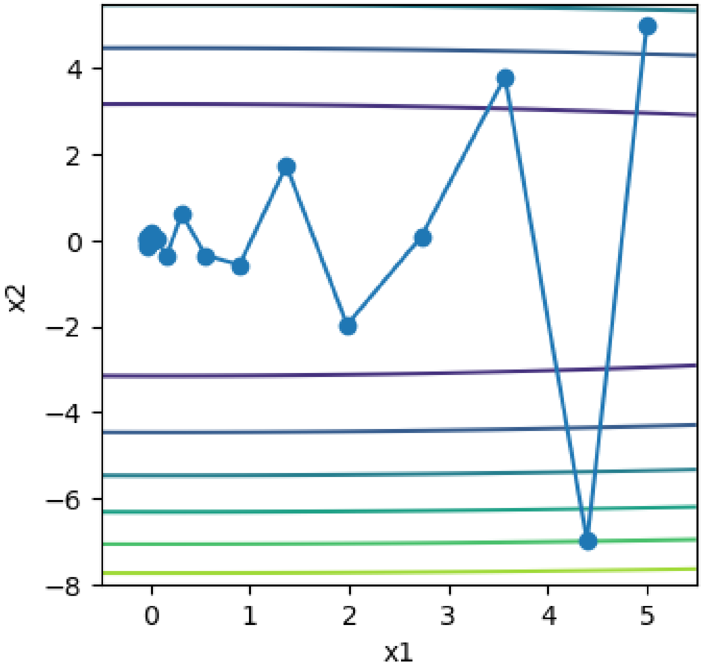
The following is the performance of GD with gradient on the ill-conditioned problem mentioned above.

[19]:

```
eta = 0.6
beta = 0.5


def momentum_gd(x1, x2, v1, v2):
    v1 = beta * v1 - eta * 0.2 * x1
    v2 = beta * v2 - eta * 4 * x2
    return x1 + v1, x2 + v2, v1, v2


# Number of iteration steps
iter = 30
# Record the iteration trace
x_arr = np.zeros((2, iter))
x_arr[:, 0] = np.array([5, 5])
f_arr = np.zeros((1, iter))
v1 = 0.
v2 = 0.
# Iteration
for ii in range(iter - 1):
    f_arr[0, ii] = 0.1 * x_arr[0, ii]**2 + 2 * x_arr[1, ii]**2
    x_arr[0,ii + 1], x_arr[1,ii + 1], v1, v2 =\
    momentum_gd(x_arr[0, ii],x_arr[1, ii], v1, v2)
f_arr[0, iter - 1] = 0.1 * x_arr[0, iter - 1]**2 + 2 * x_arr[1, iter - 1]**2
print("After %d epoches, x1 is %f, x2 is %f" %
      (iter, x_arr[0, iter - 1], x_arr[1, iter - 1]))
# Visualization
fig, ax = plt.subplots(figsize=(4, 4), dpi=100)
# Plot the contour
vis_arr_x = np.linspace(-0.5, 5.5, 100)
vis_arr_y = np.linspace(-8, 5.5, 100)
vis_X, vis_Y = np.meshgrid(vis_arr_x, vis_arr_y)
```

```
vis_Z = 0.1 * vis_X**2 + 2 * vis_Y**2
ax.contour(vis_X, vis_Y, vis_Z)
ax.plot(x_arr[0, :], x_arr[1, :],'o-')
ax.set_xlabel("x1")
ax.set_ylabel("x2")
plt.show()
```

```
After 30 epoches, x1 is 0.000243, x2 is -0.000349
```



As we can see, gradient with momentum still converges well even with the same learning rate of 0.6. The same method can also to applied to stochastic gradient descent and in particular, minibatch stochastic gradient descent.

6.5 Effective Sample Weight

In momentum method

$$\sum_{\tau=0}^{t-1}\beta^{\tau}\mathbf{g}_{t-\tau,t-\tau-1}$$

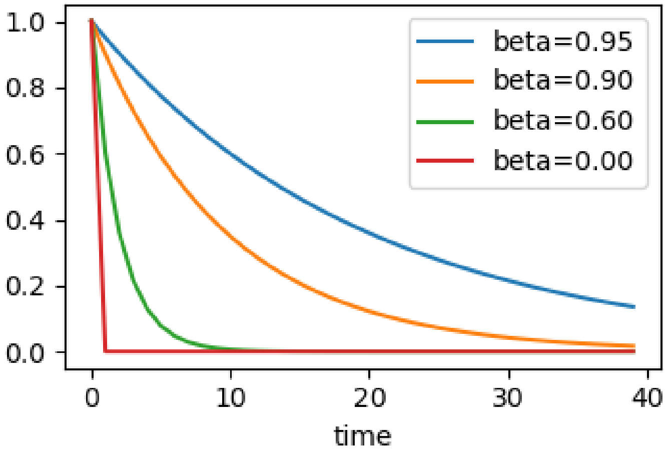
In the limit the terms add up to

$$\sum_{\tau=0}^{\infty}\beta^{\tau}=\frac{1}{1-\beta}$$

That is to say, rather than taking a step size of η in gradient descent or stochastic gradient descent we take a step of size $\frac{\eta}{1-\beta}$ while at the same time, dealing with a potentially much better behaved descent direction. The following are the behavior of different choices of β.

[7]:

```
fig, ax = plt.subplots(figsize=(4, 2.5), dpi=100)
betas = np.array([0.95, 0.9, 0.6, 0.])
for beta in betas:
    x = np.arange(40)
    ax.plot(x, beta**x, label='beta=%.2f' % beta)
ax.set_xlabel('time')
ax.legend()
plt.show()
```



6.6 Nesterov Momentum

- While the updates rules in vanilla momentum method is

$$\begin{aligned} \mathbf{v}_t &= \beta \mathbf{v}_{t-1} - \eta_t \frac{1}{b} \sum_{k=1}^{b} \nabla_{\boldsymbol{\theta}} \left[f(\mathbf{x}_k | \boldsymbol{\theta}_{t-1}) \right] \\ \boldsymbol{\theta}_t &= \boldsymbol{\theta}_{t-1} + \mathbf{v}_t \end{aligned}$$

  The updates rules in Nesrtov's accelerated gradient method are given by

$$\begin{aligned} \mathbf{v}_t &= \beta \mathbf{v}_{t-1} - \eta_t \frac{1}{b} \sum_{k=1}^{b} \nabla_{\boldsymbol{\theta}} \left[f(\mathbf{x}_k | \boldsymbol{\theta}_{t-1} + \beta \mathbf{v}_{t-1}) \right] \\ \boldsymbol{\theta}_t &= \boldsymbol{\theta}_{t-1} + \mathbf{v}_t \end{aligned}$$

  The difference between Nesterov momentum and standard momentum is where the gradient is evaluated. With Nesterov momentum, the gradient is evaluated after the current *velocity* is applied. Thus one can interpret Nesterov momentum as attempting to add a *correction factor* to the standard method of momentum.
- Unfortunately, in the stochastic gradient case, Nesterov momentum does **not** improve the rate of convergence.

7 Algorithms with Adaptive Learning Rates

7.1 Adagrad

7.1.1 Motivation

- When training a model, we typically want to decrease the learning rate as we keep on training to get good accuracy.
- For a model training on sparse feature (features that occur only infrequently), parameters associated with infrequent features only receive meaningful updates whenever there features occur.
- Given a decreasing learning rate we might end up in a situation where the parameters for common features converge rather quickly to their optimal values, whereas for infrequent features we are still short of observing them sufficiently frequently before their optimal values can be determined
- A possible hack to redress this issue would be to count the number of times we see a particular feature and to use this as a clock for adjusting learning rates

$$\eta_i = \frac{\eta_0}{\sqrt{s(i,t)+c}}$$

7.1.2 Algorithm

Adagrad addresses the issue by replacing the rather crude counter $s(i,t)$ by an aggregate of the squares of previously observed gradients. In particular, It uses

$$s(i,t+1) = s(i,t) + (\partial_i f(\mathbf{x})^2)$$

as a means to adjust the learning rate. It has two benefits:

- We no longer need to decide just when a gradient is large enough
- It scales automatically with the magnitude of the gradients.

In practice, coordinates that routinely correspond to large gradients are scaled down significantly while others with small gradients receive a much more gentle treatment. This leads to a very effective optimization procedure for computational advertising and related problems.

- Like the idea of *distorting* the space such that all eigenvalues are 1 in ill-conditioned problems while safely using the variance of the gradients as a cheap proxy for the scale of the Hessian, which is usually infeasible in complex learning problems due to memory and computational constraints

The formalized updating rule when optimizing some loss functions is

$$\begin{aligned}\mathbf{g}_t &= \nabla_{\boldsymbol{\theta}} L(y_t, f(\mathbf{x}_t, \boldsymbol{\theta}_{t-1})) \\ \mathbf{s}_t &= \mathbf{s}_{t-1} + \mathbf{g}_t^2 \\ \boldsymbol{\theta}_t &= \boldsymbol{\theta}_{t-1} - \frac{\eta_0}{\sqrt{\mathbf{s}_t + \epsilon}} \odot \mathbf{g}_t\end{aligned}$$

where the operation are applied coordinate wise. η_0 is the initial learning rate and ϵ is an additive constant that ensures that we do not divide by 0. Typically, we initialize $\mathbf{s}_0 = 0$

In deep learning, we might want to decrease the learning rate rather more slowly, which led to a number of Adagrad variants.

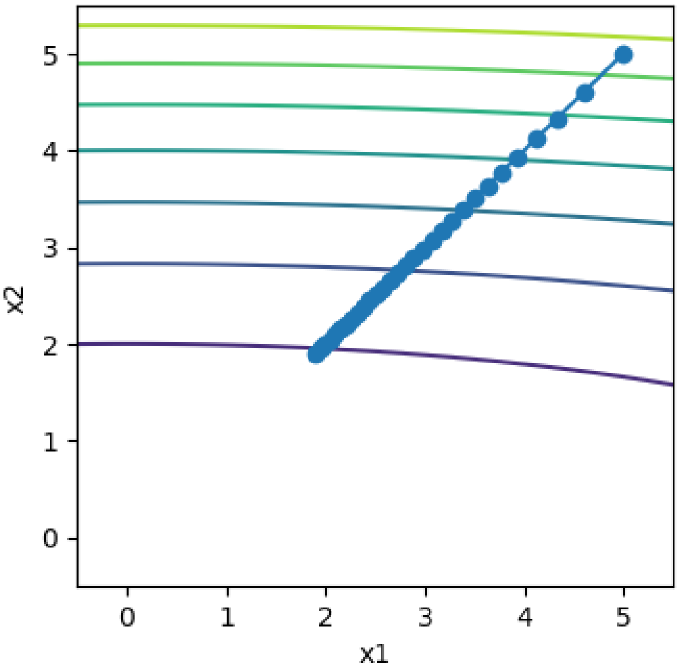
7.1.3 Adagrad Example

Still considering the ill-conditioned problem mentioned before and use an initial learning rate of 0.4. The iterative trajectory is much smoother while the independent variable does not move as much during later stages of iteration

[18]:

```
eta = 0.4


def Adagrad_gd(x1, x2, s1, s2):
    eps = 1e-16
    g1, g2 = 0.2 * x1, 4 * x2
    s1 += g1**2
    s2 += g2**2
    x1 -= eta / np.sqrt(s1 + eps) * g1
    x2 -= eta / np.sqrt(s2 + eps) * g2
    return x1, x2, s1, s2


# Number of iteration steps
iter = 30
# Record the iteration trace
x_arr = np.zeros((2, iter))
x_arr[:, 0] = np.array([5, 5])
f_arr = np.zeros((1, iter))
s1 = 0.
s2 = 0.
# Iteration
for ii in range(iter - 1):
    f_arr[0, ii] = 0.1 * x_arr[0, ii]**2 + 2 * x_arr[1, ii]**2
    x_arr[0,ii + 1], x_arr[1,ii + 1], s1, s2 =\
    Adagrad_gd(x_arr[0, ii],x_arr[1, ii], s1, s2)
f_arr[0, iter - 1] = 0.1 * x_arr[0, iter - 1]**2 + 2 * x_arr[1, iter - 1]**2
print("After %d epoches, x1 is %f, x2 is %f" %
      (iter, x_arr[0, iter - 1], x_arr[1, iter - 1]))
# Visualization
fig, ax = plt.subplots(figsize=(4, 4), dpi=100)
# Plot the contour
vis_arr_x = np.linspace(-0.5, 5.5, 100)
vis_arr_y = np.linspace(-0.5, 5.5, 100)
vis_X, vis_Y = np.meshgrid(vis_arr_x, vis_arr_y)
vis_Z = 0.1 * vis_X**2 + 2 * vis_Y**2
ax.contour(vis_X, vis_Y, vis_Z)
ax.plot(x_arr[0, :], x_arr[1, :],'o-')
ax.set_xlabel("x1")
ax.set_ylabel("x2")
plt.show()
```

```
After 30 epoches, x1 is 1.901750, x2 is 1.901750
```



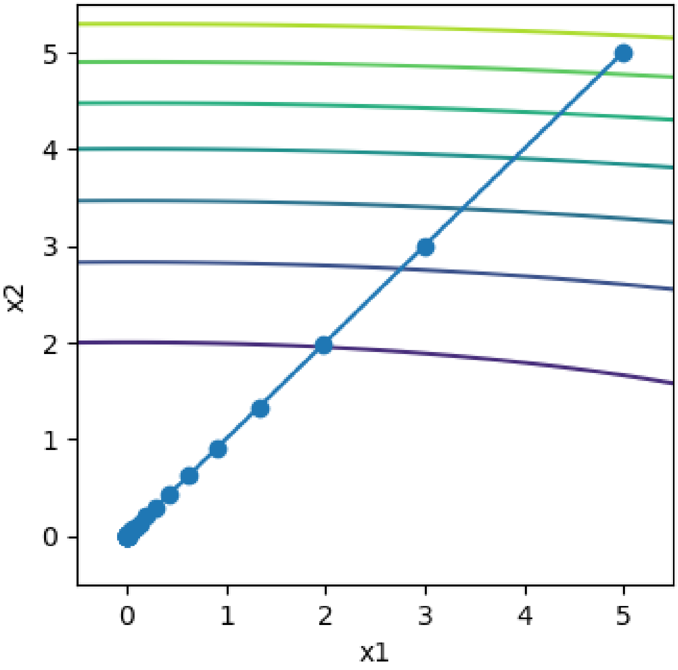
As we increase the learning rate to 2, we see much better behavior. It indicates that the decrease in learning rate might be rather aggressive, even in the noise-free case and we need to ensure that parameters converge appropriately.

[17]:

```
eta = 2


def Adagrad_gd(x1, x2, s1, s2):
    eps = 1e-16
    g1, g2 = 0.2 * x1, 4 * x2
    s1 += g1**2
    s2 += g2**2
    x1 -= eta / np.sqrt(s1 + eps) * g1
    x2 -= eta / np.sqrt(s2 + eps) * g2
    return x1, x2, s1, s2


# Number of iteration steps
iter = 30
# Record the iteration trace
x_arr = np.zeros((2, iter))
x_arr[:, 0] = np.array([5, 5])
f_arr = np.zeros((1, iter))
s1 = 0.
s2 = 0.
```

```
# Iteration
for ii in range(iter - 1):
    f_arr[0, ii] = 0.1 * x_arr[0, ii]**2 + 2 * x_arr[1, ii]**2
    x_arr[0,ii + 1], x_arr[1,ii + 1], s1, s2 =\
    Adagrad_gd(x_arr[0, ii],x_arr[1, ii], s1, s2)
f_arr[0, iter - 1] = 0.1 * x_arr[0, iter - 1]**2 + 2 * x_arr[1, iter - 1]**2
print("After %d epoches, x1 is %f, x2 is %f" %
      (iter, x_arr[0, iter - 1], x_arr[1, iter - 1]))
# Visualization
fig, ax = plt.subplots(figsize=(4, 4), dpi=100)
# Plot the contour
vis_arr_x = np.linspace(-0.5, 5.5, 100)
vis_arr_y = np.linspace(-0.5, 5.5, 100)
vis_X, vis_Y = np.meshgrid(vis_arr_x, vis_arr_y)
vis_Z = 0.1 * vis_X**2 + 2 * vis_Y**2
ax.contour(vis_X, vis_Y, vis_Z)
ax.plot(x_arr[0, :], x_arr[1, :],'o-')
ax.set_xlabel("x1")
ax.set_ylabel("x2")
plt.show()
```

```
After 30 epoches, x1 is 0.000080, x2 is 0.000080
```



7.2 RMSProp

7.2.1 Motivation

The decrease in learning rate in Adagrad might be rather aggressive, even in the noise-free case. In the meantime, the coordinate wise adaptivity of Adagrad is highly desirable as a preconditioner.

7.2.2 Algorithm

RMSProp is proposed to be a simple fix to decouple rate scheduling from coordinate-adaptive learning rates. This issue is that Adagrad accumulates the squares of the gradient $\mathbf{g}_t$ into a state vector $\mathbf{s}_t = \mathbf{s}_{t-1} + \mathbf{g}_t^2$. As a result $\mathbf{s}_t$ keeps on growing without bound due to the lack of normalization, essentially linearly as the algorithm converges.

RMSProp use a leaky average in the same way we used in momentum method

$$\mathbf{s}_t = \gamma \mathbf{s}_{t-1} + (1-\gamma)\mathbf{g}_t^2$$

for some parameter $\gamma > 0$ which determines how long the history is when adjusting the per-coordinate scale

The formalized updating rule for RMSProp when optimizing some loss function is

$$\begin{aligned} \mathbf{g}_t &= \nabla_{\boldsymbol{\theta}} L(y_t, f(\mathbf{x}_t, \boldsymbol{\theta}_{t-1})) \\ \mathbf{s}_t &= \gamma \mathbf{s}_{t-1} + (1-\gamma)\mathbf{g}_t^2 \\ \boldsymbol{\theta}_t &= \boldsymbol{\theta}_{t-1} - \frac{\eta_0}{\sqrt{\mathbf{s}_t + \epsilon}} \odot \mathbf{g}_t \end{aligned}$$

7.2.3 RMSProp Example

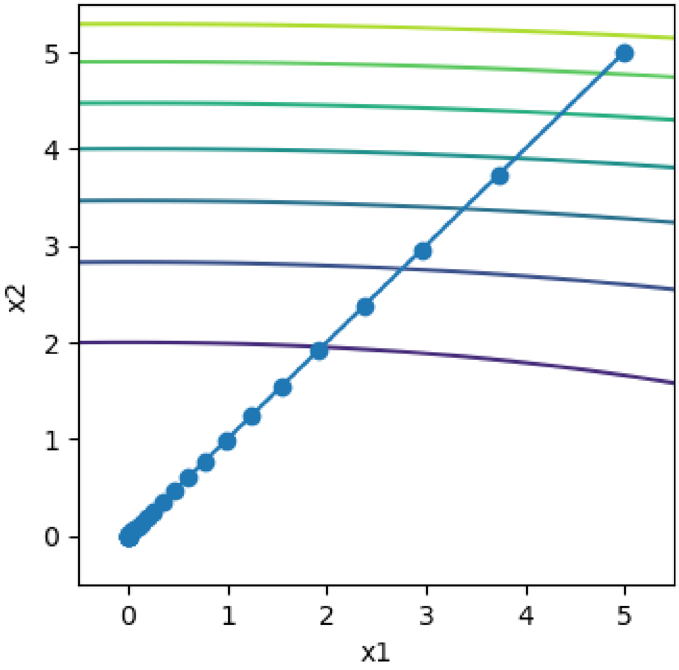
Still considering the ill-conditioned problem mentioned before and use an initial learning rate of 0.4. Recall that when we used Adagrad with a learning rate of 0.4, the variables moved only very slowly in the later stages of the algorithm since the learning rate decreased too quickly. Since η is controlled differently this does not happen with RMSProp.

[16]:

```
eta = 0.4
gamma = 0.9

def RMSProp_gd(x1, x2, s1, s2):
    eps = 1e-16
    g1, g2 = 0.2 * x1, 4 * x2
    s1 = gamma*s1+(1-gamma)*g1**2
    s2 = gamma*s2+(1-gamma)*g2**2
    x1 -= eta / np.sqrt(s1 + eps) * g1
    x2 -= eta / np.sqrt(s2 + eps) * g2
    return x1, x2, s1, s2


# Number of iteration steps
iter = 30
# Record the iteration trace
x_arr = np.zeros((2, iter))
x_arr[:, 0] = np.array([5, 5])
```

```
f_arr = np.zeros((1, iter))
s1 = 0.
s2 = 0.
# Iteration
for ii in range(iter - 1):
    f_arr[0, ii] = 0.1 * x_arr[0, ii]**2 + 2 * x_arr[1, ii]**2
    x_arr[0,ii + 1], x_arr[1,ii + 1], s1, s2 =\
    RMSProp_gd(x_arr[0, ii],x_arr[1, ii], s1, s2)
f_arr[0, iter - 1] = 0.1 * x_arr[0, iter - 1]**2 + 2 * x_arr[1, iter - 1]**2
print("After %d epoches, x1 is %f, x2 is %f" %
      (iter, x_arr[0, iter - 1], x_arr[1, iter - 1]))
# Visualization
fig, ax = plt.subplots(figsize=(4, 4), dpi=100)
# Plot the contour
vis_arr_x = np.linspace(-0.5, 5.5, 100)
vis_arr_y = np.linspace(-0.5, 5.5, 100)
vis_X, vis_Y = np.meshgrid(vis_arr_x, vis_arr_y)
vis_Z = 0.1 * vis_X**2 + 2 * vis_Y**2
ax.contour(vis_X, vis_Y, vis_Z)
ax.plot(x_arr[0, :], x_arr[1, :],'o-')
ax.set_xlabel("x1")
ax.set_ylabel("x2")
plt.show()
```

```
After 30 epoches, x1 is 0.000015, x2 is 0.000015
```



7.3 Adadelta

Adadelta is another variant of AdaGrad. The main differences lies in the fact that it decreases the amount by which the learning rate is adaptive to coordinates. Moreover, traditionally it referred to as not having a learning rate since it uses the amount of change itself as calibration for future changes.

Adadelta uses two state variables $\mathbf{s}_t$ to store a leaky average of the second moment of the gradient and $\Delta\boldsymbol{\theta}_t$ to store a leaky average of the second moment of the change of parameters in the model itself.

The formalized updating rule when optimizing some loss function is

$$\begin{aligned}
\mathbf{g}_t &= \nabla_{\boldsymbol{\theta}} L(y_t, f(\mathbf{x}_t, \boldsymbol{\theta}_{t-1})) \\
\mathbf{s}_t &= \gamma \mathbf{s}_{t-1} + (1-\gamma)\mathbf{g}_t^2 \\
\mathbf{g}_t' &= \frac{\sqrt{\Delta\mathbf{x}_{t-1} + \epsilon}}{\sqrt{\mathbf{s}_t + \epsilon}} \odot \mathbf{g}_t \\
\Delta\mathbf{x}_t &= \gamma \Delta\mathbf{x}_{t-1} + (1-\gamma)\mathbf{g}_t'^2 \\
\boldsymbol{\theta}_t &= \boldsymbol{\theta}_{t-1} - \mathbf{g}_t'
\end{aligned}$$

7.4 Adam

7.4.1 Algorithm

Adam is one efficient learning algorithm which combines all the techniques mentioned above. It is an algorithm that has become rather popular as one of the more robust and effective optimization to use in deep learning. However, it is not without issues. There are situations where Adam can diverge due to poor variance control.

Adam uses leaky averaging to obtain an estimate of both the momentum and also the second moment of the gradient

$$\begin{aligned}
\mathbf{g}_t &= \nabla_{\boldsymbol{\theta}} L(y_t, f(\mathbf{x}_t, \boldsymbol{\theta}_{t-1})) \\
\mathbf{v}_t &= \beta_1 \mathbf{v}_{t-1} + (1-\beta_1)\mathbf{g}_t \\
\mathbf{s}_t &= \beta_2 \mathbf{s}_{t-1} + (1-\beta_2)\mathbf{g}_t^2
\end{aligned}$$

Here β_1 and β_2 are nonnegative weighting parameters. Common choices for them are $\beta_1 = 0.9$ and $\beta_2 = 0.999$. That is, the variance estimate moves *much more slowly* than the momentum term. Note that if we initialize $\mathbf{v}_0 = \mathbf{s}_0 = 0$ we have a **significant amount of bias** initially towards smaller values. This can be addressed by using the fact that

$$\sum_{i=1}^{t} \beta^i = \frac{1-\beta^t}{1-\beta}$$

to re-normalize terms. The normalized state variables are given by

$$\hat{\mathbf{v}}_t = \frac{\mathbf{v}_t}{1-\beta_1^t}, \ \hat{\mathbf{s}}_t = \frac{\mathbf{s}_t}{1-\beta_2^t}$$

Then rescale the gradient in a manner very similar to that of RMSProp to obtain

$$\mathbf{g}_t' = \frac{1}{\sqrt{\hat{\mathbf{s}}_t} + \epsilon} \odot \hat{\mathbf{v}}_t$$

Pay attention to the slight cosmetic difference as the rescaling happen using $\frac{1}{\sqrt{\hat{\mathbf{s}}_t}+\epsilon}$ instead of $\frac{1}{\sqrt{\hat{\mathbf{s}}_t+\epsilon}}$. The former works arguably slightly better in practice. Typically we pick $\epsilon = 10^{-6}$ or $\epsilon = 10^{-8}$ for a good trade-off between numerical stability and fidelity. In the end, we update the parameters in a simple form

$$\boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} - \eta \mathbf{g}'_t$$

The suggested default to the step size is $\eta = 0.001$.

In a nutshell, the algorithm is

$$\begin{aligned}
\mathbf{g}_t &= \nabla_{\boldsymbol{\theta}} L(y_t, f(\mathbf{x}_t, \boldsymbol{\theta}_{t-1})) \\
\mathbf{v}_t &= \beta_1 \mathbf{v}_{t-1} + (1-\beta_1)\mathbf{g}_t \\
\mathbf{s}_t &= \beta_2 \mathbf{s}_{t-1} + (1-\beta_2)\mathbf{g}_t^2 \\
\hat{\mathbf{v}}_t &= \frac{\mathbf{v}_t}{1-\beta_1^t} \\
\hat{\mathbf{s}}_t &= \frac{\mathbf{s}_t}{1-\beta_2^t} \\
\mathbf{g}'_t &= \frac{1}{\sqrt{\hat{\mathbf{s}}_t}+\epsilon} \odot \hat{\mathbf{v}}_t \\
\boldsymbol{\theta}_t &= \boldsymbol{\theta}_{t-1} - \eta \mathbf{g}'_t
\end{aligned}$$

with the suggested default is

$$\beta_1 = 0.9,\ \beta_2 = 0.999,\ \epsilon = 10^{-8},\ \eta = 0.001$$

7.4.2 Yogi

One of the problems of Adam it that it can fail to converge even in convex settings when the second moment estimate in $\mathbf{s}_t$ blows up. As a fix a refined update and initialization for $\mathbf{s}_t$ is proposed.

In Adam, the update rule for $\mathbf{s}_t$ is

$$\begin{aligned}
\mathbf{s}_t &= \beta_2 \mathbf{s}_{t-1} + (1-\beta_2)\mathbf{g}_t^2 \\
&= \mathbf{s}_{t-1} + (1-\beta_2)(\mathbf{g}_t^2 - \mathbf{s}_{t-1})
\end{aligned}$$

Whenever $\mathbf{g}_t^2$ has high variance or updates are sparse, $\mathbf{s}_t$ might forget past values too quickly. A possible fix for this is to replace $\mathbf{g}_t^2 - \mathbf{s}_{t-1}$ by $\mathbf{g}_t^2 \odot \operatorname{sgn}(\mathbf{g}_t^2 - \mathbf{s}_{t-1})$, which makes the magnitude of the update no longer depends on the amount of deviation. This is called the **Yogi updates**

$$\mathbf{s}_t = \mathbf{s}_{t-1} + (1-\beta_2)\mathbf{g}_t^2 \odot \operatorname{sgn}(\mathbf{g}_t^2 - \mathbf{s}_{t-1})$$