Math4CS_AdaGrad

December 11, 2025

REPORT: ADAGRAD OPTIMIZATION ALGORITHM

Mathematical Foundations for Computer Science

Group 20

Student Members:

Nguyễn Thành Đạt - 2570163

Lê Phước Thành - 2570322

Lê Đức Phương - 2570480

Hồ Bảo An - 2570164

Instructor: Dr. Nguyen An Khuong

1 SECTION 1: INTRODUCTION & PROBLEM STATEMENT

1.1 1.1 Introduction

Gradient-based optimization algorithms play a central role in training machine learning models. However, standard methods such as vanilla Gradient Descent or SGD rely on a single global learning rate, causing difficulties when dealing with heterogeneous or sparse feature spaces. The Adagrad (Adaptive Gradient) algorithm was proposed to address this issue by adapting the learning rate for each parameter individually based on historical gradient information.

Unlike fixed-rate optimizers, Adagrad automatically scales the learning rate according to how frequently a parameter is updated:

- Parameters with frequent updates receive progressively smaller learning rates,
- Parameters with infrequent updates retain larger learning rates and therefore continue learning effectively.

This adaptive behavior makes Adagrad particularly well-suited for problems involving sparse features, such as natural language processing or recommendation systems.

The goal of this report is to examine the motivation behind Adagrad, describe its mathematical formulation, analyze its advantages and limitations, and discuss its impact on optimization in high-dimensional sparse settings.

1.2 1.2 Motivation: Sparse Features

In many tasks such as **natural language processing**, data are often sparse: most values are zero and only a small subset of features are active. Models like *bag-of-words* and *one-hot encoding* may have tens of thousands of features, but each sample only activates a few.

1.2.1 1.2.1 Global learning rate decay is not suitable

If using a decaying learning rate globally:

$$\eta_t = \frac{\eta_0}{\sqrt{t}}$$

then a parameter corresponding to a rare feature will only be updated when that feature appears. If it first appears at a large step t, the learning rate may already be too small.

1.2.2 1.2.2 Example of Global Decay Calculation

Assume: * Initial learning rate: $\eta_0 = 0.1$ * A rare feature appears for the first time at step $t = 1000$ * Gradient at that step: $g_{1000} = 0.5$

Calculate learning rate at $t = 1000$ using global decay:

$$\eta_{1000} = \frac{\eta_0}{\sqrt{1000}} = \frac{0.1}{31.6228} \approx 0.00316$$

Update weight (initial weight $w_0 = 0$):

$$w_{1000} = w_0 - \eta_{1000} g_{1000} = 0 - 0.00316 \times 0.5 \approx -0.00158$$

Observation: The weight change is extremely small, even though the feature may be important. This illustrates the limitation of using a globally decaying learning rate for sparse features: rare features get almost no update when they first appear at large t.

1.3 1.3 Naive Solution and Its Limitations

A simple idea is to track how many times each feature appears. Let $s(i, t)$ be the number of times feature i has been activated until time t. Then:

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

with $c > 0$ to avoid division by zero.

1.3.1 1.3.1 Example of naive solution

Assume the following: * Initial weight: $w_0 = 0$ * Initial learning rate: $\eta_0 = 0.1$ * Small constant: $c = 10^{-8}$ * Gradient values observed at each step: $g_1 = 0.5, g_2 = 0.1, g_3 = 0.3$

a. The feature appears for the first time: $s(i, 1) = 1$.

$$\eta_{i,1} = \frac{0.1}{\sqrt{1 + 10^{-8}}} \approx 0.1$$

Update the weight:

$$w_1 = w_0 - \eta_{i,1} g_1 = 0 - 0.1 * 0.5 = -0.05$$

b. The feature appears for the second time: $s(i, 2) = 2$**.**

$$\eta_{i,2} = \frac{0.1}{\sqrt{2 + 10^{-8}}} \approx 0.0707$$

Update the weight:

$$w_2 = w_1 - \eta_{i,2} g_2 = -0.05 - 0.0707 * 0.1 \approx -0.0571$$

c. The feature appears for the third time: $s(i, 3) = 3$**.**

$$\eta_{i,3} = \frac{0.1}{\sqrt{3 + 10^{-8}}} \approx 0.0577$$

Update the weight:

$$w_3 = w_2 - \eta_{i,3} g_3 = -0.0571 - 0.0577 * 0.3 \approx -0.0744$$

> **Explanation:** At each step, the learning rate decreases slightly as the feature appears more often. This ensures that rare features are updated with a larger step initially, while frequent features gradually get smaller updates. However, this naive approach still ignores the magnitude of gradients over time.

1.3.2 1.3.2 Limitations

1. **Ignores gradient magnitude:** two occurrences with gradients 0.001 and 10 are treated equally.
2. **Cannot handle well** cases where small gradient occurs often or large gradient occurs rarely.
3. **Depends on the definition** of "one occurrence," introducing extra hyperparameters.

> **Hence:** A more sophisticated mechanism is needed: consider not only the number of appearances but also the gradient magnitude.

2 SECTION 2: ALGORITHM & MECHANISM

2.1 2.1 Adagrad Mechanism: Accumulated Squared Gradients

2.1.1 2.1.1. From a Simple Counter to Squared Gradients

Consider training a model where each parameter corresponds to a feature. Some features appear very often, others are rare. A naive idea is to keep a counter of how many times each feature has appeared up to step t:

$$s(i,t) = \text{number of times feature } i \text{ has appeared up to step } t$$

Then define a coordinate-wise learning rate:

$$\eta_{i,t} = \frac{\eta}{\sqrt{s(i,t)}}$$

So, if a feature appears many times (large $s(i,t)$), its learning rate becomes smaller; if it appears rarely (small $s(i,t)$), its learning rate remains larger. This already captures some intuition, but it has a serious limitation:

- It is binary: either "appeared" or "not appeared".
- It does not distinguish between small gradient values (feature present but not important) and large gradient values (feature strongly affects the loss).
- One must choose an arbitrary threshold to decide what counts as an "appearance".

Adagrad improves on this by replacing this crude counter with a sum of squared gradients.

For each coordinate i, Adagrad maintains:

$$s(i,t+1) = s(i,t) + (\partial_i f(x_t))^2$$

Here:

- $\partial_i f(x_t)$ is the gradient of the loss with respect to parameter x_i at step t.
- $s(i,t)$ is no longer just "how many times feature i appeared", but rather "how large the gradients of coordinate i have been over time".

  Core idea: Instead of counting appearances, Adagrad accumulates the squared size of gradients for each parameter.

2.1.2 2.1.2. Adaptive Learning Rate per Coordinate

Given the accumulated squared gradients $s(i,t)$, Adagrad defines a **per-coordinate learning rate**:

$$\eta_{i,t} = \frac{\eta}{\sqrt{s(i,t) + \epsilon}}$$

Where: * η: The base learning rate chosen by the user (hyperparameter). * ϵ: A small constant (e.g., 10^{-8}) added to avoid division by zero (numerical stability).

The parameter update rule becomes:

$$x_i^{(t+1)} = x_i^{(t)} - \eta_{i,t}\partial_i f(x_t) = x_i^{(t)} - \frac{\eta}{\sqrt{s(i,t) + \epsilon}}\partial_i f(x_t)$$

Conclusion: Each parameter x_i has its own effective learning rate, which shrinks as the sum of its past squared gradients grows. Frequent features (large accumulated gradients) get small updates, while rare features (small accumulated gradients) get larger updates.

2.1.3 2.1.3. Analysis of the Update Rule

1. No Need for Arbitrary Thresholds

In the naive counter approach, one must decide: *"How large must a gradient be to count as an appearance?"*. This is arbitrary and problem-dependent.

With Adagrad: * Every gradient, no matter how small, contributes to $s(i,t)$. * Larger gradients naturally contribute more (because they are squared). * There is no binary decision ("count" vs "don't count") and no threshold hyperparameter to tune.

2. The algorithm automatically weighs gradients according to their size.

3. Automatic Scaling According to Gradient Magnitude

The accumulated sum $s(i,t) = \sum_{\tau=1}^{t} (\partial_i f(x_\tau))^2$ grows:

- **Fast** if gradients for coordinate i are large or frequent.
- **Slow** if gradients are small or rare.

4. Because the effective learning rate is $\eta_{i,t} = \frac{\eta}{\sqrt{s(i,t)+\epsilon}}$, this means:

- **For "noisy" or frequently changing coordinates** (large or frequent gradients): $s(i,t)$ becomes large $\rightarrow \sqrt{s(i,t)}$ is large, so $\eta_{i,t}$ becomes small. These coordinates are slowed down, avoiding overshooting.
- **For "quiet" or rare coordinates** (small or infrequent gradients): $s(i,t)$ stays small, so $\eta_{i,t}$ remains relatively large. These coordinates keep larger steps, allowing them to learn even with sparse updates.

  Summary: Adagrad's "accumulated squared gradient" mechanism replaces a rough frequency counter with a smooth, magnitude-sensitive history for each parameter, giving automatic, coordinate-wise learning rate adaptation.

3 2.2 Adagrad and the Concept of Preconditioning

3.1 2.2.1 Quadratic Convex Optimization and Condition Number

Consider a quadratic convex function:

$$f(x) = \frac{1}{2} x^\top Q x + c^\top x + b, \quad Q \succ 0$$

Here: * $x \in \mathbb{R}^d$ is the parameter vector. * Q is a symmetric positive definite matrix (the Hessian). * c and b are constants.

Properties of this function:

- **The gradient:** $\nabla f(x) = Qx + c$
- **The Hessian** (matrix of second derivatives): $\nabla^2 f(x) = Q$
- **Geometry:** The shape of the level sets (contours) of f is an ellipsoid defined by Q.

The condition number of Q is defined as:

$$\kappa(Q) = \frac{\lambda_{\max}(Q)}{\lambda_{\min}(Q)}$$

where $\lambda_{\max}$ and $\lambda_{\min}$ are the largest and smallest eigenvalues of Q.

Intuitive Interpretation:

1. **If $\kappa(Q) \approx 1$:**
   - All eigenvalues are similar.
   - The level sets are close to circles (spheres).
   - Gradient descent moves smoothly and converges fast.
2. **If $\kappa(Q)$ is large:**
   - The eigenvalues are very different.
   - The level sets are very elongated ellipsoids.
   - Gradient descent "zigzags": it moves quickly along steep directions (large eigenvalues) but slowly along flat ones (small eigenvalues), leading to slow convergence.

   Thus, $\kappa(Q)$ measures how "difficult" the optimization problem is for standard gradient descent.

3.1.1 2.2.2 Preconditioning: Changing Coordinates to Improve Conditioning

Preconditioning means applying a linear transformation to the variables to improve the condition number and make the optimization landscape "rounder" (closer to a sphere).

1. Change of Variables

Define a new variable y:

$$y = P^{1/2}x$$

where $P \succ 0$ is the **preconditioning matrix**. In terms of y, the function becomes:

$$f(x) = f(P^{-1/2}y) = \frac{1}{2}y^\top(P^{-1/2}QP^{-1/2})y + (\text{linear terms})$$

2. The New Hessian

The Hessian matrix of the transformed function is:

$$\tilde{Q} = P^{-1/2}QP^{-1/2}$$

3. The Strategy

If P is chosen well, the condition number $\kappa(\tilde{Q})$ can be much smaller than the original $\kappa(Q)$, making gradient descent converge much faster.

- **Ideal Choice:** Theoretically, the best choice is $P = Q$. Then:

  $$\tilde{Q} = Q^{-1/2}QQ^{-1/2} = I$$

  The condition number becomes **1**, and the problem becomes perfectly conditioned (the contours are spheres). However, computing and inverting the full matrix Q in high dimensions is **too expensive**.

- **Practical Alternative (Diagonal Preconditioning):** Instead of the full matrix, we use only the diagonal elements:
  - Take $M = \text{diag}(Q)$ (matrix with only diagonal entries of Q).
  - Use $M^{-1/2}$ as the preconditioner:

$$\tilde{Q} = M^{-1/2} Q M^{-1/2}$$

  Key Insight: This diagonal preconditioning is cheap to compute and works very well when the main anisotropy (stretching) of the function is aligned with the coordinate axes. **Adagrad approximates this diagonal preconditioning** by using accumulated gradients to estimate the curvature along each axis.

3.1.2 2.2.3 Viewing Adagrad as Time-Varying Diagonal Preconditioning

Now connect the concept of preconditioning to Adagrad.

1. Vector Form of Adagrad

We can write the Adagrad accumulator in vector notation:

- **Accumulator:**

$$s_t = s_{t-1} + g_t \odot g_t$$

  where $g_t = \nabla f(x_t)$ and $\odot$ denotes element-wise multiplication.

- **Diagonal Matrix:** Let $D_t = \text{diag}(s_t + \epsilon)$ be a diagonal matrix with entries $s_t + \epsilon$ on the diagonal.

- **Update Rule:** The Adagrad update rule can be rewritten as:

$$x_{t+1} = x_t - \eta D_t^{-1/2} g_t$$

  Observation: This is exactly **Gradient Descent with a diagonal preconditioner** $D_t^{-1/2}$ that changes over time.

2. Connection to the Hessian

Near the optimum x^* of a quadratic function, the gradient behaves like $g_t \approx Q(x_t - x^*)$. In coordinates where the problem is close to axis-aligned:

- Directions with large curvature (large diagonal entries of Hessian Q) tend to produce larger gradients.
- Directions with small curvature (small diagonal entries) tend to produce smaller gradients.

Over time, the accumulated squared gradients s_t approximate (up to scaling) the diagonal of Q^2. Therefore, taking $D_t^{-1/2}$ is similar to using $\text{diag}(Q)^{-1/2}$ as a preconditioner.

3. Final Interpretation

So Adagrad can be interpreted as:

- **Stochastic Gradient Descent + Diagonal Preconditioning**
- Where the preconditioner is learned online from the gradients instead of being computed from the Hessian Q explicitly.

This explains why Adagrad dramatically improves convergence on problems where the main stretching of the landscape is roughly axis-aligned: it is implicitly **"rounding" the optimization landscape** in each coordinate.

3.2 2.3 Gradient as a Proxy for the Hessian

3.2.1 2.3.1 Computing the Hessian Is Not Feasible in Deep Learning

The Hessian of a function $f : \mathbb{R}^d \to \mathbb{R}$ is a $d \times d$ matrix. In deep learning, the number of parameters d is often:

- Hundreds of thousands,
- Millions,
- Or even billions.

Storing a full $d \times d$ matrix:

- Requires memory on the order of $O(d^2)$.
- For $d = 10^6$, this is 10^{12} entries—completely impractical.

Computing and using the full Hessian (e.g., in Newton's method or full-matrix preconditioning) is therefore **impossible in practice**:

- Time complexity is too high.
- Memory requirement is too large.
- Implementation is too complex and unstable with noisy mini-batch gradients.

So, in deep learning, one must use **first-order information only** (gradients), but still wants some of the benefits of second-order methods (which use the Hessian).

3.2.2 2.3.2 The Clever Idea: Using Gradient Statistics as a Proxy

For a quadratic function:

$$f(x) = \frac{1}{2}x^\top Qx + c^\top x + b$$

the gradient is:

$$g(x) = Qx + c$$

Near the optimum x^*, this is approximately:

$$g(x) \approx Q(x - x^*)$$

Consider one coordinate i:

- If the curvature (roughly Q_{ii}) is large, then a small change in x_i produces a large change in the gradient g_i.
- Over the course of training (different mini-batches, different steps), a high-curvature coordinate tends to have larger gradients (in magnitude) or higher variability.

Adagrad's Approximation Strategy

Adagrad does not compute the Hessian Q. Instead, it only observes the gradients g_t over time and accumulates:

$$s(i,t) = \sum_{\tau=1}^{t} (g_{\tau,i})^2$$

This quantity captures information about:

- How large the gradients for coordinate i are on average.
- How sensitive the loss is to changes in parameter x_i.

In other words, $s(i,t)$ is a cheap, noisy proxy for the diagonal of some function of the Hessian (roughly larger if curvature is higher).

Thus:

- The Hessian diagonal tells us how steep the landscape is along each coordinate.
- The accumulated squared gradients in Adagrad approximate that information *without ever forming the Hessian.*

  Key Insight: Adagrad uses the magnitude (variance) of gradients as a proxy for the diagonal of the Hessian.

3.2.3 2.3.3 Adagrad in Deep Learning

In deep learning contexts:

- Optimization uses stochastic gradients from mini-batches.
- Exact second-order information is noisy and expensive.
- However, gradients are already being computed at each step.

Adagrad leverages this by:

1. **Re-using gradients** to build a running estimate $s(i,t)$ of how "active" each parameter is.
2. Interpreting this as rough curvature information per coordinate.
3. Scaling learning rates accordingly.

So, instead of: * **Computing a full Hessian:** Costly $O(d^2)$ memory and $O(d^3)$ time to invert/factorize. * **Adagrad:** * Stores only a vector $s_t \in \mathbb{R}^d$ (linear cost $O(d)$). * Updates it with simple element-wise operations (very cheap). * Achieves some of the benefits of Hessian-based preconditioning by using gradient history as an implicit, approximate curvature signal.

3.3 2.4 The Adagrad Algorithm

From section 2.2 and 2.3 on Preconditioning, we have seen the idea of adjusting learning rates per coordinate and using a diagonal matrix to scale the parameter space. Now we formalize the Adagrad algorithm - a practical implementation of that idea.

3.3.1 2.4.1 The Algorithm

We use the variable $\mathbf{s}_t$ to accumulate the variance of past gradients as follows:

$$\begin{aligned} \mathbf{g}_t &= \partial_{\mathbf{w}} l(y_t, f(\mathbf{x}_t, \mathbf{w})), \\ \mathbf{s}_t &= \mathbf{s}_{t-1} + \mathbf{g}_t^2, \\ \mathbf{w}_t &= \mathbf{w}_{t-1} - \frac{\eta}{\sqrt{\mathbf{s}_t + \epsilon}} \cdot \mathbf{g}_t. \end{aligned} \tag{12.7.5}$$

Important note: All operations are performed element-wise:

- $\mathbf{g}_t^2$ means $[g_{t,1}^2, g_{t,2}^2, \ldots, g_{t,d}^2]$
- $\sqrt{\mathbf{s}_t + \epsilon}$ means $[\sqrt{s_{t,1} + \epsilon}, \sqrt{s_{t,2} + \epsilon}, \ldots, \sqrt{s_{t,d} + \epsilon}]$
- $\frac{\eta}{\sqrt{\mathbf{s}_t + \epsilon}} \cdot \mathbf{g}_t$ means $\left[\frac{\eta}{\sqrt{s_{t,1} + \epsilon}} \cdot g_{t,1}, \frac{\eta}{\sqrt{s_{t,2} + \epsilon}} \cdot g_{t,2}, \ldots, \frac{\eta}{\sqrt{s_{t,d} + \epsilon}} \cdot g_{t,d}\right]$

Parameters: - η - learning rate - ϵ - small additive constant (typically 10^{-8}) to avoid division by zero - **Initialization:** $\mathbf{s}_0 = \mathbf{0}$

3.3.2 2.4.2 Explaination of Each Step

Step 1: Compute gradient

$$\mathbf{g}_t = \partial_{\mathbf{w}} l(y_t, f(\mathbf{x}_t, \mathbf{w}))$$

The gradient at the current point indicates the direction of steepest change.

Step 2: Accumulate squared gradients

Each coordinate "remembers" its gradient history: - If gradient is large many times $\rightarrow$ $\mathbf{s}_t$ accumulates large values $\rightarrow$ effective lr becomes small - If gradient is small $\rightarrow$ $\mathbf{s}_t$ accumulates slowly $\rightarrow$ effective lr remains relatively large

Step 3: Update parameters

Update with learning rate adjusted per coordinate: - Each coordinate has its own lr: $\frac{\eta}{\sqrt{s_{t,i} + \epsilon}}$ - Learning rate decreases as $s_{t,i}$ increases

3.3.3 2.4.3 Design Principles

1. Per-coordinate adaptation

In many problems, different coordinates have different "sensitivities". Coordinates that change strongly (large gradients) need small learning rates to avoid overshooting, while coordinates that change weakly (small gradients) need large learning rates to learn faster. Adagrad adjusts automatically without manual tuning for each parameter.

2. Connection to Preconditioning

This is an application of automatic preconditioning - Adagrad uses gradient magnitude as a proxy for the diagonal of the Hessian, helping to "flatten" the objective function without computing eigenvalues. Each step is equivalent to gradient descent with a diagonal scaling matrix $D_t = \text{diag}(1/\sqrt{s_t})$.

3. Computational Cost

Like momentum, we track an auxiliary variable $\mathbf{s}_t \in \mathbb{R}^d$ with $O(d)$ storage - negligible compared to computing the full Hessian's $O(d^2)$ cost. The learning rate decreases as $O(1/\sqrt{t})$: stable for convex problems but may decrease too quickly for non-convex/deep learning tasks, requiring variants like RMSProp or Adam.

3.4 2.5 Applications of Adagrad

Adagrad is particularly effective in domains with **sparse data** or features with **varying frequencies**.

1. **Natural Language Processing (NLP)**
   - *Example:* Text classification using *bag-of-words*. Rare words still learn meaningful weights thanks to Adagrad.
2. **Computational Advertising**
   - CTR prediction models use user IDs, ad IDs, and queries—mostly sparse features. Adagrad prevents rare ads from being "ignored."
3. **Recommender Systems**
   - Many users or products have few interactions. Adagrad ensures embeddings are sufficiently trained.

4 SECTION 3: COMPUTATION & IMPLEMENTATION

4.1 3.1. Step-by-Step Calculation Example

In this section, we demonstrate Adagrad's behavior on two different objective functions: - **Function 1 (Axis-aligned)**: Variables are independent - **Function 2 (Rotated)**: Variables are strongly correlated

4.1.1 Case 1: Axis-Aligned Function

Objective Function

$$f_1(\mathbf{x}) = 0.1x_1^2 + 2x_2^2$$

This is an axis-aligned elliptic paraboloid where the variables x_1 and x_2 are independent.

Finding the Critical Point (Theoretical Solution) Solve the system $\nabla f_1 = \mathbf{0}$:

$$\begin{cases} \frac{\partial f_1}{\partial x_1} = 0.2x_1 = 0 \\ \frac{\partial f_1}{\partial x_2} = 4x_2 = 0 \end{cases}$$

$$\Rightarrow x_1 = 0, \quad x_2 = 0$$

Conclusion: Minimum at $(0, 0)$ with $f_1(0, 0) = 0$

Setup for Adagrad Parameters are set as follows: - Starting point: $\mathbf{x}^{(0)} = (x_1^{(0)}, x_2^{(0)}) = (5, 3)$ - Learning rate: $\eta = 0.4$ - Epsilon: $\varepsilon = 10^{-8}$ - Gradient formula: $\nabla f_1 = \begin{bmatrix} 0.2x_1 \\ 4x_2 \end{bmatrix}$

Step-by-Step Calculations for Function 1 1. Step 1 (t=1)

Current: $\mathbf{x}^{(0)} = (5.0, 3.0)$

a. Compute gradient:

$$\mathbf{g}_1 = \begin{bmatrix} 0.2 \times 5 \\ 4 \times 3 \end{bmatrix} = \begin{bmatrix} 1.0 \\ 12.0 \end{bmatrix}$$

b. Update accumulator $\mathbf{s}$:

$$\mathbf{s}_1 = \begin{bmatrix} 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 1.0^2 \\ 12.0^2 \end{bmatrix} = \begin{bmatrix} 1.0 \\ 144.0 \end{bmatrix}$$

c. Compute effective learning rate:

- For x_1: $\frac{\eta}{\sqrt{s_1^{(1)}+\varepsilon}} = \frac{0.4}{\sqrt{1.0}} = 0.4$
- For x_2: $\frac{\eta}{\sqrt{s_2^{(1)}+\varepsilon}} = \frac{0.4}{\sqrt{144.0}} = \frac{0.4}{12} \approx 0.0333$

d. Compute update:

$$\Delta\mathbf{x}_1 = \eta_{eff} \odot \mathbf{g}_1 = \begin{bmatrix} 0.4 \times 1.0 \\ 0.0333 \times 12.0 \end{bmatrix} = \begin{bmatrix} 0.4 \\ 0.4 \end{bmatrix}$$

e. Update parameters:

$$\mathbf{x}^{(1)} = \mathbf{x}^{(0)} - \Delta\mathbf{x}_1 = \begin{bmatrix} 5.0 - 0.4 \\ 3.0 - 0.4 \end{bmatrix} = \begin{bmatrix} 4.6 \\ 2.6 \end{bmatrix}$$

Function value: $f_1(4.6, 2.6) = 0.1(4.6)^2 + 2(2.6)^2 = 2.116 + 13.52 = 15.636$

2. Step 2 (t=2)

Current: $\mathbf{x}^{(1)} = (4.6, 2.6)$

a. Gradient:

$$\mathbf{g}_2 = \begin{bmatrix} 0.2 \times 4.6 \\ 4 \times 2.6 \end{bmatrix} = \begin{bmatrix} 0.92 \\ 10.4 \end{bmatrix}$$

b. Update $\mathbf{s}$:

$$\mathbf{s}_2 = \begin{bmatrix} 1.0 \\ 144.0 \end{bmatrix} + \begin{bmatrix} 0.8464 \\ 108.16 \end{bmatrix} = \begin{bmatrix} 1.8464 \\ 252.16 \end{bmatrix}$$

c. Effective learning rate:

- For x_1: $\frac{0.4}{\sqrt{1.8464}} \approx 0.2944$
- For x_2: $\frac{0.4}{\sqrt{252.16}} \approx 0.0252$

d. Update:

$$\Delta\mathbf{x}_2 = \begin{bmatrix} 0.2944 \times 0.92 \\ 0.0252 \times 10.4 \end{bmatrix} \approx \begin{bmatrix} 0.2709 \\ 0.2621 \end{bmatrix}$$

$$\mathbf{x}^{(2)} = \begin{bmatrix} 4.6 - 0.2709 \\ 2.6 - 0.2621 \end{bmatrix} = \begin{bmatrix} 4.3291 \\ 2.3379 \end{bmatrix}$$

Function value: $f_1(4.3291, 2.3379) \approx 12.817$

3. Step 3 (t=3)

Current: $\mathbf{x}^{(2)} \approx (4.3291, 2.3379)$

a. Gradient:

$$\mathbf{g}_3 = \begin{bmatrix} 0.8658 \\ 9.3516 \end{bmatrix}$$

b. Update $\mathbf{s}$:

$$\mathbf{s}_3 = \begin{bmatrix} 1.8464 + 0.7496 \\ 252.16 + 87.4524 \end{bmatrix} = \begin{bmatrix} 2.596 \\ 339.6124 \end{bmatrix}$$

c. Effective learning rate:

$$\eta_{eff} \approx \begin{bmatrix} 0.2482 \\ 0.0217 \end{bmatrix}$$

d. Update:

$$\mathbf{x}^{(3)} \approx \begin{bmatrix} 4.1142 \\ 2.1350 \end{bmatrix}$$

Function value: $f_1(4.1142, 2.1350) \approx 10.805$

Observations for Function 1 (Axis-aligned)

1. **Effective LR adapts per coordinate:**
   - x_1 has small gradient (0.2 coefficient) $\rightarrow$ larger effective LR maintained
   - x_2 has large gradient (4.0 coefficient) $\rightarrow$ smaller effective LR
2. **Convergence is balanced:**
   - Both coordinates move toward 0 smoothly
   - The per-coordinate adaptation works perfectly because variables are independent
3. **Updates align with coordinate axes:**
   - Gradient always points along x_1 or x_2 axis
   - Adagrad's diagonal scaling is ideal for this structure

4.1.2 Case 2: Rotated Function

Objective Function

$$f_2(\mathbf{x}) = 0.1(x_1 + x_2)^2 + 2(x_1 - x_2)^2$$

This is a **rotated elliptic paraboloid** where variables x_1 and x_2 are strongly correlated.

Finding the Critical Point Expand the function:

$$f_2(\mathbf{x}) = 0.1(x_1^2 + 2x_1x_2 + x_2^2) + 2(x_1^2 - 2x_1x_2 + x_2^2)$$

$$= 0.1x_1^2 + 0.2x_1x_2 + 0.1x_2^2 + 2x_1^2 - 4x_1x_2 + 2x_2^2$$

$$= 2.1x_1^2 - 3.8x_1x_2 + 2.1x_2^2$$

Solve $\nabla f_2 = \mathbf{0}$:

$$\begin{cases} \frac{\partial f_2}{\partial x_1} = 4.2x_1 - 3.8x_2 = 0 \\ \frac{\partial f_2}{\partial x_2} = -3.8x_1 + 4.2x_2 = 0 \end{cases}$$

From the first equation: $x_1 = \frac{3.8}{4.2} x_2 \approx 0.905 x_2$

Substituting into the second equation yields $x_1 = x_2 = 0$.

Conclusion: Minimum at $(0, 0)$ with $f_2(0, 0) = 0$

Setup for Adagrad Parameters: - Starting point: $\mathbf{x}^{(0)} = (5, 3)$ (same as Function 1) - Learning rate: $\eta = 0.4$ - Epsilon: $\varepsilon = 10^{-8}$ - Gradient formula: $\nabla f_2 = \begin{bmatrix} 4.2x_1 - 3.8x_2 \\ -3.8x_1 + 4.2x_2 \end{bmatrix}$

Step-by-Step Calculations for Function 2 1. Step 1 (t=1)

Current: $\mathbf{x}^{(0)} = (5.0, 3.0)$

a. Compute gradient:

$$\mathbf{g}_1 = \begin{bmatrix} 4.2 \times 5 - 3.8 \times 3 \\ -3.8 \times 5 + 4.2 \times 3 \end{bmatrix} = \begin{bmatrix} 21.0 - 11.4 \\ -19.0 + 12.6 \end{bmatrix} = \begin{bmatrix} 9.6 \\ -6.4 \end{bmatrix}$$

b. Update **s**:

$$\mathbf{s}_1 = \begin{bmatrix} 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 92.16 \\ 40.96 \end{bmatrix} = \begin{bmatrix} 92.16 \\ 40.96 \end{bmatrix}$$

c. Effective learning rate:

- For x_1: $\frac{0.4}{\sqrt{92.16}} = \frac{0.4}{9.6} \approx 0.0417$
- For x_2: $\frac{0.4}{\sqrt{40.96}} = \frac{0.4}{6.4} = 0.0625$

d. Update:

$$\Delta \mathbf{x}_1 = \begin{bmatrix} 0.0417 \times 9.6 \\ 0.0625 \times (-6.4) \end{bmatrix} = \begin{bmatrix} 0.4 \\ -0.4 \end{bmatrix}$$

$$\mathbf{x}^{(1)} = \begin{bmatrix} 5.0 - 0.4 \\ 3.0 - (-0.4) \end{bmatrix} = \begin{bmatrix} 4.6 \\ 3.4 \end{bmatrix}$$

Function value: $f_2(4.6, 3.4) = 0.1(8.0)^2 + 2(1.2)^2 = 6.4 + 2.88 = 9.28$

2. Step 2 (t=2)

Current: $\mathbf{x}^{(1)} = (4.6, 3.4)$

a. Gradient:

$$\mathbf{g}_2 = \begin{bmatrix} 4.2 \times 4.6 - 3.8 \times 3.4 \\ -3.8 \times 4.6 + 4.2 \times 3.4 \end{bmatrix} = \begin{bmatrix} 6.4 \\ -3.2 \end{bmatrix}$$

b. Update **s**:

$$\mathbf{s}_2 = \begin{bmatrix} 92.16 + 40.96 \\ 40.96 + 10.24 \end{bmatrix} = \begin{bmatrix} 133.12 \\ 51.2 \end{bmatrix}$$

c. Effective learning rate:

- For x_1: $\frac{0.4}{\sqrt{133.12}} \approx 0.0347$
- For x_2: $\frac{0.4}{\sqrt{51.2}} \approx 0.0559$

d. Update:

$$\mathbf{x}^{(2)} \approx \begin{bmatrix} 4.6 - 0.222 \\ 3.4 - (-0.179) \end{bmatrix} = \begin{bmatrix} 4.378 \\ 3.579 \end{bmatrix}$$

Function value: $f_2(4.378, 3.579) \approx 6.706$

3. Step 3 (t=3)

Current: $\mathbf{x}^{(2)} \approx (4.378, 3.579)$

a. Gradient:

$$\mathbf{g}_3 \approx \begin{bmatrix} 4.785 \\ -1.393 \end{bmatrix}$$

b. Update $\mathbf{s}$:

$$\mathbf{s}_3 \approx \begin{bmatrix} 156.01 \\ 53.14 \end{bmatrix}$$

c. Update:

$$\mathbf{x}^{(3)} \approx \begin{bmatrix} 4.225 \\ 3.655 \end{bmatrix}$$

Function value: $f_2(4.225, 3.655) \approx 6.567$

Observations for Function 2 (Rotated)

1. **Gradients are not axis-aligned:**
   - Gradient has components in both directions due to coupling terms
   - Direction changes as we move, not simply pointing toward origin along axes
2. **Convergence is slower:**
   - After 3 steps: Function 1 reduced to ~10.8, Function 2 to ~6.6
   - But Function 1 started at ~17.5, Function 2 at ~9.3
   - Relative progress is similar, but the path is less efficient for Function 2
3. **Per-coordinate adaptation is suboptimal:**
   - Adagrad scales x_1 and x_2 independently
   - But optimal direction is diagonal (along principal axes at 45°)
   - Adagrad cannot capture this cross-coordinate relationship
4. **Movement pattern:**
   - Function 1: Smooth convergence along both axes
   - Function 2: Zigzag pattern as it tries to navigate the rotated landscape

4.1.3 Explaination Why is Function 2 More Difficult for Adagrad

1. The Root Cause: Coordinate System Mismatch

The fundamental challenge with Function 2 stems from a mismatch between the coordinate system we're using (x_1, x_2) and the natural geometry of the function.

Function 2 expanded:

$$f_2(\mathbf{x}) = 2.1x_1^2 - 3.8x_1x_2 + 2.1x_2^2$$

The key issue is the cross term $-3.8x_1x_2$. This term creates coupling between x_1 and x_2 - you cannot optimize one coordinate independently of the other.

2. Understanding Through the Hessian Matrix

The Hessian matrix (second derivatives) reveals the true geometry:

Function 1 (Axis-aligned):

$$H_1 = \begin{bmatrix} 0.2 & 0 \\ 0 & 4 \end{bmatrix}$$

- **Diagonal matrix** → coordinates are independent
- Principal axes align with x_1 and x_2 axes
- Eigenvalues: $\lambda_1 = 0.2$, $\lambda_2 = 4$

Function 2 (Rotated):

$$H_2 = \begin{bmatrix} 4.2 & -3.8 \\ -3.8 & 4.2 \end{bmatrix}$$

- **Non-diagonal matrix** → coordinates are coupled through off-diagonal terms
- Principal axes are rotated 45° from coordinate axes
- **Same eigenvalues**: $\lambda_1 = 0.4$, $\lambda_2 = 8$ (just like Function 1 after accounting for the 0.1 vs 2 ratio pattern)

The eigenvalues are identical, but the eigenvectors (principal directions) are different: - Function 1: Eigenvectors are $[1, 0]$ and $[0, 1]$ (along axes) - Function 2: Eigenvectors are $[1, 1]/\sqrt{2}$ and $[1, -1]/\sqrt{2}$ (diagonal directions at $\pm 45°$)

3. What Adagrad Actually Does

Adagrad adjusts the learning rate independently for each coordinate:

$$\text{Effective LR for } x_1 : \frac{\eta}{\sqrt{s_1}} \quad \text{vs} \quad \text{Effective LR for } x_2 : \frac{\eta}{\sqrt{s_2}}$$

This is equivalent to applying a **diagonal scaling matrix**:

$$D_t = \text{diag}\left(\frac{1}{\sqrt{s_{t,1}}}, \frac{1}{\sqrt{s_{t,2}}}\right)$$

Why this works for Function 1: - Function 1's Hessian is already diagonal → diagonally scaling the space perfectly matches the function's geometry - Moving along x_1 doesn't affect the optimal step in x_2, and vice versa

Why this fails for Function 2: - Function 2's Hessian is non-diagonal → diagonal scaling cannot capture the off-diagonal coupling - The optimal direction is 45° diagonal, but Adagrad can only adjust x_1 and x_2 separately - Moving in the x_1 direction requires simultaneous adjustment in x_2 to stay on the optimal path

4. Mathematical Insight: Why Independent Scaling Fails

Consider taking a step from point (x_1, x_2):

For Function 1: - Gradient: $\nabla f_1 = [0.2x_1, 4x_2]$ - Components are independent: changing x_1 only affects the x_1 part of gradient - Adagrad's per-coordinate scaling is perfect

For Function 2: - Gradient: $\nabla f_2 = [4.2x_1 - 3.8x_2, -3.8x_1 + 4.2x_2]$ - Components are coupled: both depend on both x_1 AND x_2 - When you step in x_1 direction, the gradient in BOTH directions changes - Adagrad scales each coordinate independently → cannot handle this coupling

5. Practical Implication

From our calculations: - **Step 1**: Both functions make similar progress - **Step 2-3**: Function 1 converges smoothly; Function 2 shows slower, uneven progress - **Root cause**: Function 2 requires coordinated movement in both dimensions, but Adagrad can only adjust each dimension separately

What would work better for Function 2: 1. **Momentum methods** (e.g., Adam): Build velocity in the optimal diagonal direction 2. **Newton's method**: Use the full Hessian (including off-diagonal terms) to find optimal direction 3. **Coordinate rotation**: Transform to the principal axes first, then apply Adagrad

> Adagrad is a diagonal optimizer - it can only adapt learning rates along the coordinate axes independently. This is: - **Perfect** when the problem's natural geometry aligns with coordinate axes (diagonal Hessian) - **Suboptimal** when the problem has rotated/correlated structure (non-diagonal Hessian) The 45° rotation doesn't change the difficulty of the optimization problem intrinsically (eigenvalues are the same), but it changes how well Adagrad's per-coordinate adaptation matches the problem structure.

4.2 3.2 Implementation from Scratch

In this section, we implement the Adagrad algorithm from scratch. We need to define two main functions: 1. `init_adagrad_states`: To initialize the state variable s_t (accumulated squared gradients), initialized to zeros. 2. `adagrad`: The update function that modifies the parameters based on the Adagrad update rule:

$$s_t \leftarrow s_{t-1} + g_t^2$$
$$x_t \leftarrow x_{t-1} - \frac{\eta}{\sqrt{s_t + \epsilon}} \cdot g_t$$

Where η is the learning rate and ϵ is a small constant (e.g., 10^{-6}) to prevent division by zero.

[ ]:
```
import torch

def init_adagrad_states(feature_dim):
    """
    Initializes the state variable s (accumulated squared gradients).
    """
    s_w = torch.zeros((feature_dim, 1))
    s_b = torch.zeros(1)
    return (s_w, s_b)

def adagrad(params, states, hyperparams):
    """
    Performs the Adagrad update.
```

```
    params: List of model parameters (w, b)
    states: List of state variables (s_w, s_b)
    hyperparams: Dictionary containing the learning rate 'lr'
    """
    eps = 1e-6
    for p, s in zip(params, states):
        with torch.no_grad():
            # 1. Update the state sum of squared gradients
            s[:] += torch.square(p.grad)

            # 2. Update the parameter
            # The effective learning rate is lr / sqrt(s + eps)
            p[:] -= hyperparams['lr'] * p.grad / torch.sqrt(s + eps)

            # 3. Reset gradients for the next iteration
            p.grad.data.zero_()
```

4.3 3.3 Experiment on Quadratic Convex Function

We evaluate Adagrad on the objective function $f(\mathbf{x}) = 0.1x_1^2 + 2x_2^2$.

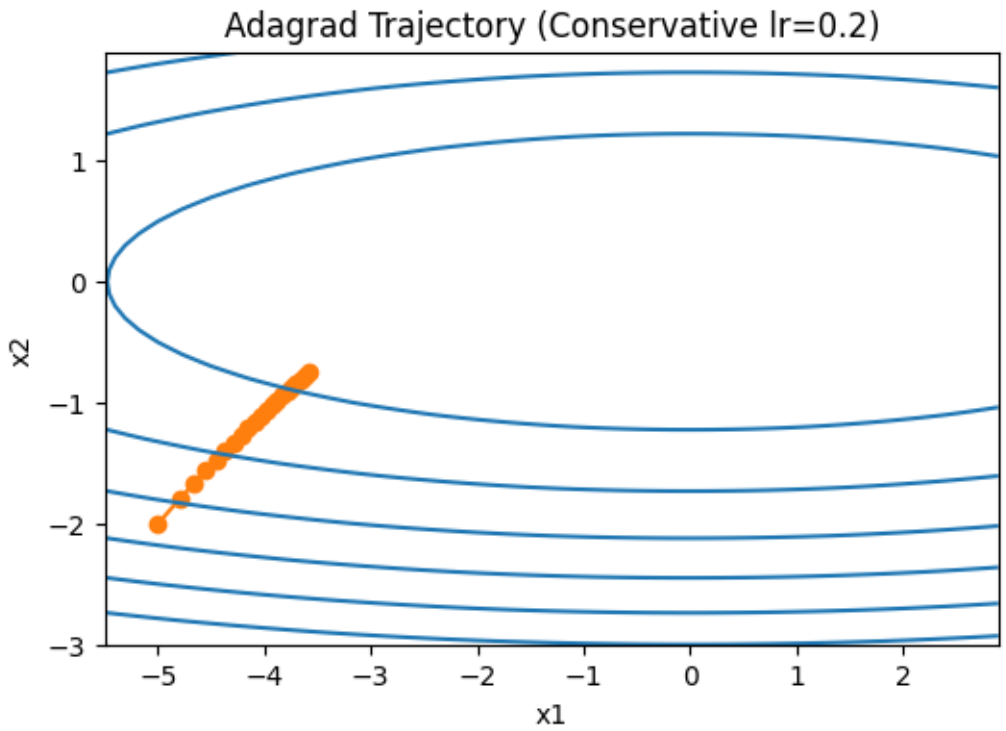
To demonstrate the robustness of Adagrad, we test two contrasting scenarios: 1. $\eta = 0.2$: A conservative learning rate to observe how the accumulated squared gradients might slow down the training excessively. 2. $\eta = 3.0$: An extremely high learning rate to test if the preconditioning mechanism can stabilize the trajectory despite the aggressive step size.

[ ]:
```
import matplotlib.pyplot as plt
import math

# Define the objective function
def f_2d(x1, x2):
    return 0.1 * x1**2 + 2 * x2**2

# Helper function to visualize the trajectory
def show_trace_2d(f, results, title):
    plt.figure(figsize=(6, 4))
    plt.plot(*zip(*results), '-o', color='#ff7f0e')
    x1, x2 = torch.meshgrid(torch.arange(-5.5, 3.0, 0.1),
                            torch.arange(-3.0, 2.0, 0.1), indexing='ij')
    plt.contour(x1, x2, f(x1, x2), colors='#1f77b4')
    plt.xlabel('x1')
    plt.ylabel('x2')
    plt.title(title)
    plt.show()

# Simulation function for Adagrad on 2D function
def train_2d_adagrad(lr):
    x1, x2 = -5, -2  # Starting point
```

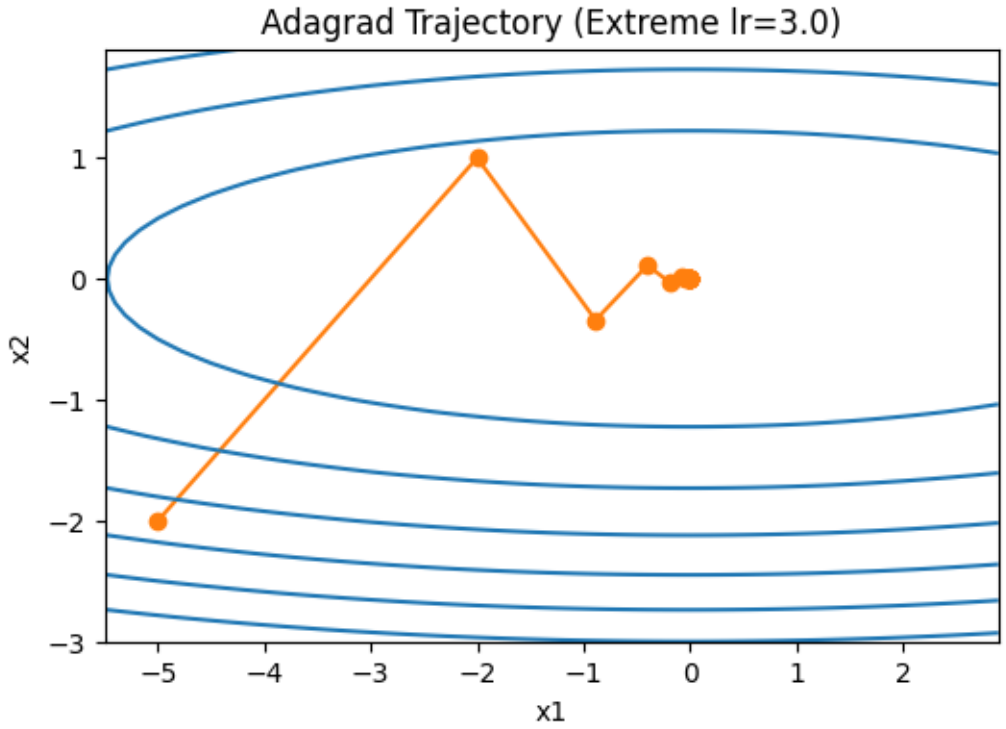
```
    s1, s2 = 0, 0    # Initial states
    results = [(x1, x2)]

    print(f"Training with lr = {lr}...")
    for i in range(20):
        # Calculate gradients manually for f(x) = 0.1*x1^2 + 2*x2^2
        g1 = 0.2 * x1
        g2 = 4 * x2

        # Adagrad Update Rule
        s1 += g1 ** 2
        s2 += g2 ** 2

        x1 -= lr / math.sqrt(s1 + 1e-6) * g1
        x2 -= lr / math.sqrt(s2 + 1e-6) * g2

        results.append((x1, x2))

    return results
```

[ ]:
```
# Experiment 1: Conservative Learning Rate (lr=0.2)
results_small = train_2d_adagrad(lr=0.2)
show_trace_2d(f_2d, results_small, "Adagrad Trajectory (Conservative lr=0.2)")

# Experiment 2: Extreme Learning Rate (lr=3.0)
results_large = train_2d_adagrad(lr=3.0)
show_trace_2d(f_2d, results_large, "Adagrad Trajectory (Extreme lr=3.0)")
```

```
Training with lr = 0.2…
```




```
Training with lr = 3.0…
```



###Analysis of the Experiment The experiment reveals the fundamental characteristics—both strengths and weaknesses—of the Adagrad algorithm: 1. The "Premature Stopping" Problem (Visible in $\eta = 0.1$): * **Observation:** The trajectory starts moving towards the minimum but "freezes" midway, failing to converge to the optimal point $(0, 0)$ within 20 steps. * **Mathematical Explanation:** Adagrad accumulates the squared gradients in the denominator ($s_t \leftarrow s_{t-1} + g_t^2$). Since g_t^2 is always positive, s_t is monotonically increasing. * **Consequence:** The effective learning rate $\eta_{eff} = \frac{\eta}{\sqrt{s_t + \epsilon}}$ decreases continuously. With a conservative initial learning rate of $\eta = 0.1$, the effective step size shrinks to near-zero too quickly, causing the algorithm to stop learning before reaching the target. This illustrates why Adagrad can struggle with "dying" learning rates in long training sessions. 2. Robustness to Large Steps (Visible in $\eta = 3.0$): * **Observation:** Despite an extremely aggressive learning rate, the algorithm converges successfully. * **Mechanism:** In the steep direction (x_2), the gradient is large, causing s_{x2} to explode rapidly. This effectively "divides" the learning rate by a large number, acting as an automatic brake. This demonstrates Adagrad's ability to stabilize training even when the hyperparameters are not perfectly tuned.

4.4 3.4 Concise Implementation

In practice, we use the optimized implementations provided by deep learning frameworks. Below is how to invoke Adagrad using PyTorch's optim module.

[ ]:
```
import torch.nn as nn

# 1. Define a simple model and loss function
net = nn.Sequential(nn.Linear(2, 1))
loss = nn.MSELoss()

# 2. Initialize Adagrad Optimizer
# We pass the model parameters and the learning rate
optimizer = torch.optim.Adagrad(net.parameters(), lr=0.1)

# Example of how to use it in a loop (Pseudo-code for demonstration)
# ----------------------------------------------------------------
# for X, y in data_iter:
#     l = loss(net(X), y)
#     optimizer.zero_grad()   # Clear previous gradients
#     l.backward()          # Compute new gradients
#     optimizer.step()        # Update parameters using Adagrad logic
# ----------------------------------------------------------------

print(f"Optimizer initialized: {optimizer}")
```

```
Optimizer initialized: Adagrad (
Parameter Group 0
    differentiable: False
    eps: 1e-10
    foreach: None
```

```
    fused: None
    initial_accumulator_value: 0
    lr: 0.1
    lr_decay: 0
    maximize: False
    weight_decay: 0
)
```

5 SECTION 4: EXERCISES IN DETAIL

5.1 Exercise 1: Proof of Norm Preservation

5.1.1 Problem Statement:

Prove that for an orthogonal matrix U and vectors $\mathbf{c}, \delta$:

$$\|\mathbf{c} - \delta\|_2 = \|\mathbf{U}\mathbf{c} - \mathbf{U}\delta\|_2$$

Why does this mean that the magnitude of perturbations does not change after an orthogonal change of variables?

5.1.2 Proof Derivation:

To prove the equality, we analyze the squared Euclidean norm of the transformed difference.

1. Expansion to Quadratic Form:

First, we express the squared norm using the inner product notation. Recall that $\|\mathbf{x}\|^2 = \mathbf{x}^\top\mathbf{x}$. Applying this to the right-hand side:

$$\|\mathbf{U}\mathbf{c} - \mathbf{U}\delta\|^2 = (\mathbf{U}\mathbf{c} - \mathbf{U}\delta)^\top(\mathbf{U}\mathbf{c} - \mathbf{U}\delta)$$

2. Factorization and Transpose:

We can factor out the matrix $\mathbf{U}$ from the difference terms: $\mathbf{U}\mathbf{c} - \mathbf{U}\delta = \mathbf{U}(\mathbf{c} - \delta)$. Substituting this back and applying the transpose property $(\mathbf{A}\mathbf{B})^\top = \mathbf{B}^\top\mathbf{A}^\top$:

$$\begin{aligned}\|\mathbf{U}\mathbf{c} - \mathbf{U}\delta\|^2 &= (\mathbf{U}(\mathbf{c} - \delta))^\top(\mathbf{U}(\mathbf{c} - \delta)) \\ &= (\mathbf{c} - \delta)^\top\mathbf{U}^\top\mathbf{U}(\mathbf{c} - \delta)\end{aligned}$$

3. Applying Orthogonality:

Since $\mathbf{U}$ is an orthogonal matrix, by definition $\mathbf{U}^\top\mathbf{U} = \mathbf{I}$ (Identity matrix). The equation simplifies significantly:

$$(\mathbf{c} - \delta)^\top\mathbf{I}(\mathbf{c} - \delta) = (\mathbf{c} - \delta)^\top(\mathbf{c} - \delta)$$

4. Conclusion:

Recognizing that $(\mathbf{c} - \delta)^\top(\mathbf{c} - \delta)$ is exactly the definition of $\|\mathbf{c} - \delta\|^2$, we have derived:

$$\|\mathbf{U}\mathbf{c} - \mathbf{U}\delta\|^2 = \|\mathbf{c} - \delta\|^2$$

Taking the square root of both sides completes the proof:

$$\|\mathbf{U}\mathbf{c} - \mathbf{U}\delta\|_2 = \|\mathbf{c} - \delta\|_2$$

5.1.3 Interpretation: Stability of Perturbations

The result above leads to a crucial insight in linear algebra and signal processing:

- **Geometry Preservation:** Orthogonal matrices (representing rotations or reflections) preserve the Euclidean distance. Transforming the space with $\mathbf{U}$ does not stretch or shrink the relative distance between vectors.
- **Noise Stability:** If we consider δ as a **perturbation** or **noise vector** added to the signal $\mathbf{c}$, the proof shows that the magnitude (energy) of this noise remains unchanged after transformation.
- **Application:** This property is vital in methods like **PCA** or **SVD**, ensuring that performing a change of basis does not artificially amplify or suppress the noise/error in the data.

5.2 Exercise 2: Try out Adagrad for an Axis-aligned Function and a Rotated Function

5.2.1 Problem Statement:

Try out Adagrad for the following objective functions:

1. **Axis-aligned function:**
$$f(\mathbf{x}) = 0.1x_1^2 + 2x_2^2$$

2. **Rotated function (by 45 degrees):**
$$f(\mathbf{x}) = 0.1(x_1 + x_2)^2 + 2(x_1 - x_2)^2$$

Does it behave differently?

5.2.2 Detailed solution:

This content has been presented in **Section 3.1**.

5.3 Exercis 3: Proof of the Gershgorin Circle Theorem

5.3.1 Problem Statement:

Prove Gerschgorin's circle theorem which states that eigenvalues λ_i of a matrix $\mathbf{M}$ satisfy:

$$|\lambda_i - \mathbf{M}_{jj}| \leq \sum_{k \neq j} |\mathbf{M}_{jk}|$$

for at least one choice of j.

5.3.2 Proof:

Let λ be an eigenvalue of $\mathbf{M}$ with a corresponding eigenvector $x = (x_1, x_2, ..., x_n)^\top \neq 0$.The eigenvalue equation is given by:

$$\mathbf{M}x = \lambda x$$

Consider the i-th row of this equation:

$$\sum_{j=1}^{n} \mathbf{M}_{ij} x_j = \lambda x_i$$

We can separate the diagonal term from the summation:

$$\mathbf{M}_{ii} x_i + \sum_{j \neq i} \mathbf{M}_{ij} x_j = \lambda x_i$$

Rearranging the terms to group x_i:

$$\sum_{j \neq i} \mathbf{M}ijx_j = (\lambda - \mathbf{M}ii)x_i \quad (*)$$

Since x is an eigenvector, it is not the zero vector. Let us choose the index i corresponding to the component with the largest absolute value:

$$|x_i| = \max_k |x_k| > 0$$

Note that for all j, we have $\frac{|x_j|}{|x_i|} \leq 1$.

Taking the absolute value of both sides of equation $(*)$ and applying the triangle inequality:

$$|(\lambda - \mathbf{M}_{ii})x_i| = \left| \sum_{j \neq i} \mathbf{M}_{ij} x_j \right| \leq \sum_{j \neq i} |\mathbf{M}_{ij}||x_j|$$

$$|\lambda - \mathbf{M}_{ii}||x_i| \leq \sum_{j \neq i} |\mathbf{M}_{ij}||x_j|$$

Dividing both sides by $|x_i|$ (which is valid since $|x_i| > 0$):

$$|\lambda - \mathbf{M}_{ii}| \leq \sum_{j \neq i} |\mathbf{M}_{ij}| \frac{|x_j|}{|x_i|}$$

Since $\frac{|x_j|}{|x_i|} \leq 1$, we arrive at:

$$|\lambda - \mathbf{M}_{ii}| \leq \sum_{j \neq i} |\mathbf{M}_{ij}|$$

This inequality shows that the eigenvalue λ lies in the disc centered at $\mathbf{M}_{ii}$ with radius $R_i = \sum_{j \neq i} |\mathbf{M}_{ij}|$. Thus, every eigenvalue must satisfy this condition for at least one row index i.

5.4 Exercise 4: Eigenvalues of Diagonally Preconditioned Matrices

5.4.1 Problem Statement:

What does Gerschgorin's theorem tell us about the eigenvalues of the diagonally preconditioned matrix $\text{diag}^{-1/2}(\mathbf{M})\mathbf{M}\text{diag}^{-1/2}(\mathbf{M})$?

5.4.2 Answer:

Let $\mathbf{D} = \text{diag}(\mathbf{M})$. The preconditioned matrix is defined as:

$$\tilde{\mathbf{M}} = \mathbf{D}^{-1/2}\mathbf{M}\mathbf{D}^{-1/2}$$

The entries of this new matrix, denoted as $\tilde{m}_{ij}$, are calculated as follows:

$$\tilde{m}_{ij} = \frac{1}{\sqrt{\mathbf{M}_{ii}}}\mathbf{M}_{ij}\frac{1}{\sqrt{\mathbf{M}_{jj}}} = \frac{\mathbf{M}_{ij}}{\sqrt{\mathbf{M}_{ii}\mathbf{M}_{jj}}}$$

To apply Gerschgorin's theorem, we examine the diagonal entries (centers of the discs) and the off-diagonal sums (radii of the discs).

1. Centers of the Gerschgorin discs:

For the diagonal elements ($i = j$):

$$\tilde{m}_{ii} = \frac{\mathbf{M}_{ii}}{\sqrt{\mathbf{M}_{ii}\mathbf{M}_{ii}}} = \frac{\mathbf{M}_{ii}}{\mathbf{M}_{ii}} = 1$$

This means every Gerschgorin disc for the matrix $\tilde{\mathbf{M}}$ is centered exactly at 1 in the complex plane.

2. Radii of the Gerschgorin discs:

The radius R_i for the i-th row is the sum of the absolute values of the off-diagonal entries:

$$R_i = \sum_{j \neq i} |\tilde{m}_{ij}| = \sum_{j \neq i} \frac{|\mathbf{M}_{ij}|}{\sqrt{\mathbf{M}_{ii}\mathbf{M}_{jj}}}$$

Conclusion:

Gerschgorin's theorem tells us that every eigenvalue λ of the preconditioned matrix $\tilde{\mathbf{M}}$ satisfies:

$$|\lambda - 1| \leq \sum_{j \neq i} \frac{|\mathbf{M}_{ij}|}{\sqrt{\mathbf{M}_{ii}\mathbf{M}_{jj}}}$$

Interpretation: The eigenvalues are clustered around 1. The maximum distance of any eigenvalue from 1 is bounded by the sum of the normalized off-diagonal elements. If the original matrix $\mathbf{M}$ is diagonally dominant (diagonal elements are much larger than off-diagonal ones), the eigenvalues will be very close to 1, indicating a condition number close to 1, which is ideal for numerical stability and convergence speed in iterative algorithms.

5.5 Exercise 5: Adagrad on LeNet (Fashion-MNIST)

We train the improved LeNet-5 architecture using the Adagrad optimizer on the Fashion-MNIST dataset.

- Model: Improved LeNet-5 (replaced Sigmoid with ReLU, and AvgPool with MaxPool to mitigate vanishing gradient issues).
- Optimizer: Adagrad with lr=0.01.
- Loss Function: Cross Entropy Loss.

[ ]:
```
import torch
import torch.nn as nn
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt
import time
```

[ ]:
```
# 1. Setup Device (GPU if available)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Training on device: {device}")
```

```
Training on device: cuda
```

[ ]:
```
# 2. Load Fashion-MNIST Dataset
transform = transforms.Compose([transforms.ToTensor()])
train_dataset = torchvision.datasets.FashionMNIST(root='./data', train=True,
                                                  transform=transform,␣
 ↪download=True)
test_dataset = torchvision.datasets.FashionMNIST(root='./data', train=False,
                                                 transform=transform,␣
 ↪download=True)

batch_size = 256
train_loader = DataLoader(dataset=train_dataset, batch_size=batch_size,␣
 ↪shuffle=True)
test_loader = DataLoader(dataset=test_dataset, batch_size=batch_size,␣
 ↪shuffle=False)
```

```
100%|      | 26.4M/26.4M [00:02<00:00, 11.1MB/s]
100%|      | 29.5k/29.5k [00:00<00:00, 212kB/s]
100%|      | 4.42M/4.42M [00:01<00:00, 3.92MB/s]
100%|      | 5.15k/5.15k [00:00<00:00, 15.1MB/s]
```

[ ]:
```python
# 3. Define Improved LeNet (ReLU + MaxPool)
def get_net():
    return nn.Sequential(
        nn.Conv2d(1, 6, kernel_size=5, padding=2), nn.ReLU(),
        nn.MaxPool2d(kernel_size=2, stride=2),
        nn.Conv2d(6, 16, kernel_size=5), nn.ReLU(),
        nn.MaxPool2d(kernel_size=2, stride=2),
        nn.Flatten(),
        nn.Linear(16 * 5 * 5, 120), nn.ReLU(),
        nn.Linear(120, 84), nn.ReLU(),
        nn.Linear(84, 10)
    ).to(device)
```

[ ]:
```python
# 4. Training Loop Function
def train_and_record(net, optimizer, num_epochs=10, name="Model"):
    criterion = nn.CrossEntropyLoss()
    train_loss_history = []
    test_acc_history = []

    print(f"--- Starting Training: {name} ---")
    start_time = time.time()

    for epoch in range(num_epochs):
        net.train()
        running_loss = 0.0
        for X, y in train_loader:
            X, y = X.to(device), y.to(device)
            optimizer.zero_grad()
            outputs = net(X)
            loss = criterion(outputs, y)
            loss.backward()
            optimizer.step()
            running_loss += loss.item()

        # Evaluate
        net.eval()
        correct, total = 0, 0
        with torch.no_grad():
            for X, y in test_loader:
                X, y = X.to(device), y.to(device)
                outputs = net(X)
                _, predicted = torch.max(outputs.data, 1)
                total += y.size(0)
                correct += (predicted == y).sum().item()

        avg_loss = running_loss / len(train_loader)
        acc = correct / total
```

```
        train_loss_history.append(avg_loss)
        test_acc_history.append(acc)

        print(f"Epoch [{epoch+1}/{num_epochs}] Loss: {avg_loss:.4f} | Acc: {acc:
 ↪.4f}")

    print(f"Total time: {time.time() - start_time:.1f}s\n")
    return train_loss_history, test_acc_history
```

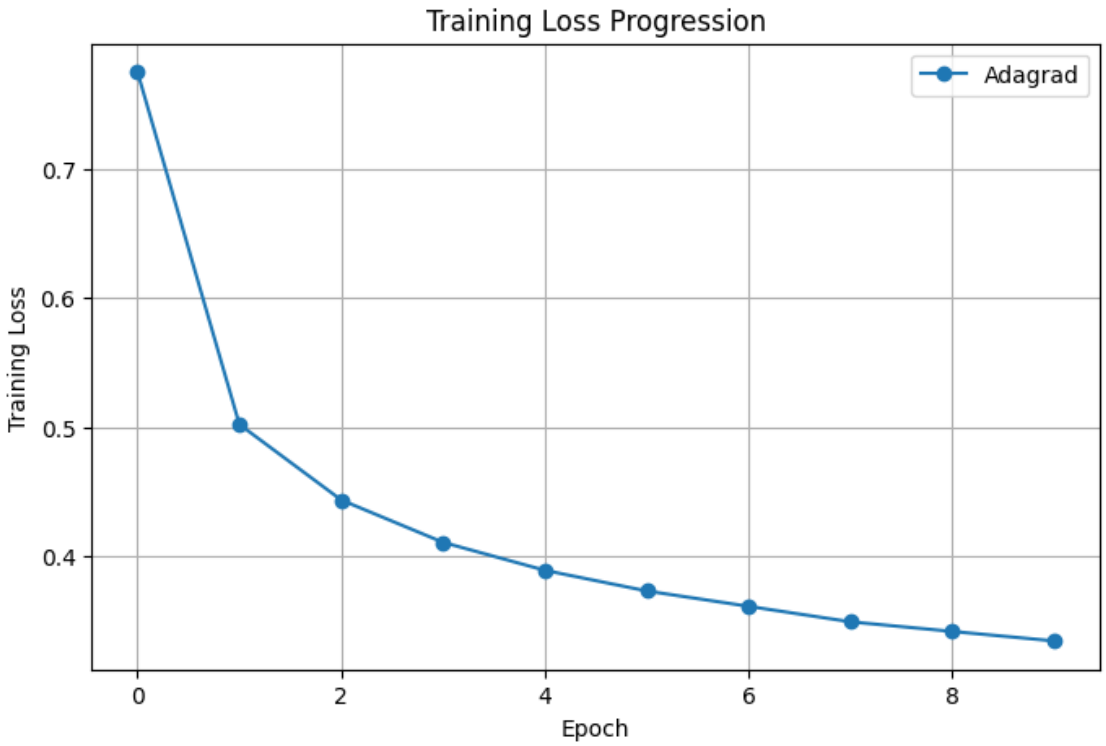
[ ]:
```
# 5. Visualize Function
def visualize_history(histories, titles, metric_name="Loss"):
    plt.figure(figsize=(8, 5))
    for history, title in zip(histories, titles):
        plt.plot(history, marker='o', label=title)
    plt.xlabel('Epoch')
    plt.ylabel(metric_name)
    plt.title(f'{metric_name} Progression')
    plt.legend()
    plt.grid(True)
    plt.show()
```

[ ]:
```
# 5. Initialize Optimizer (Adagrad) & Model
net = get_net()
optimizer = torch.optim.Adagrad(net.parameters(), lr=0.01)
```

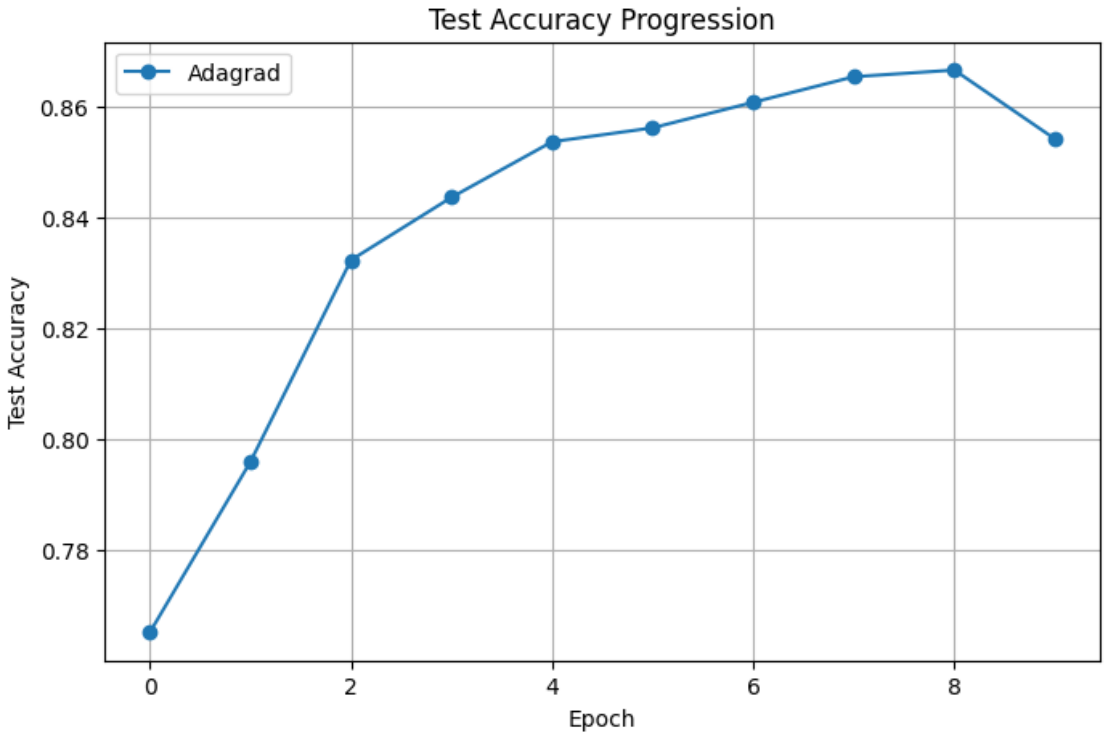
[ ]:
```
# 6. Run experiment
loss_adagrad, acc_adagrad = train_and_record(net, optimizer, num_epochs=10,␣
 ↪name="Adagrad")
```

```
--- Starting Training: Adagrad ---
Epoch [1/10] Loss: 0.7751 | Acc: 0.7653
Epoch [2/10] Loss: 0.5021 | Acc: 0.7960
Epoch [3/10] Loss: 0.4435 | Acc: 0.8323
Epoch [4/10] Loss: 0.4107 | Acc: 0.8436
Epoch [5/10] Loss: 0.3891 | Acc: 0.8536
Epoch [6/10] Loss: 0.3731 | Acc: 0.8561
Epoch [7/10] Loss: 0.3612 | Acc: 0.8607
Epoch [8/10] Loss: 0.3492 | Acc: 0.8653
Epoch [9/10] Loss: 0.3417 | Acc: 0.8665
Epoch [10/10] Loss: 0.3344 | Acc: 0.8541
Total time: 83.4s
```

[ ]:
```
visualize_history([loss_adagrad], ["Adagrad"], "Training Loss")
```



[ ]:
```
visualize_history([acc_adagrad], ["Adagrad"], "Test Accuracy")
```



5.6 Exercise 6: Modifying Adagrad for Less Aggressive Decay

Problem Identification: As observed in quadratic experiment (Section 3.3), Adagrad's state variable s_t accumulates squared gradients indefinitely:

$$s_t = s_{t-1} + g_t^2$$

Because g_t^2 is always non-negative, s_t keeps growing. Consequently, the learning rate $\eta/\sqrt{s_t}$ eventually decreases to zero. If this happens too early (before reaching the optimum), the model stops learning.

Proposed Modification: To prevent s_t from growing indefinitely, we can replace the full sum with an exponential moving average (EMA). This allows the algorithm to "forget" old gradients and focus on the recent curvature of the loss surface.

The Modified Update Rule: Instead of $s_t = s_{t-1} + g_t^2$, we use:

$$s_t \leftarrow \gamma s_{t-1} + (1-\gamma) g_t^2$$

Where γ (typically 0.9) is a decay factor.This ensures that s_t does not grow without bound.The learning rate effectively stabilizes instead of vanishing.

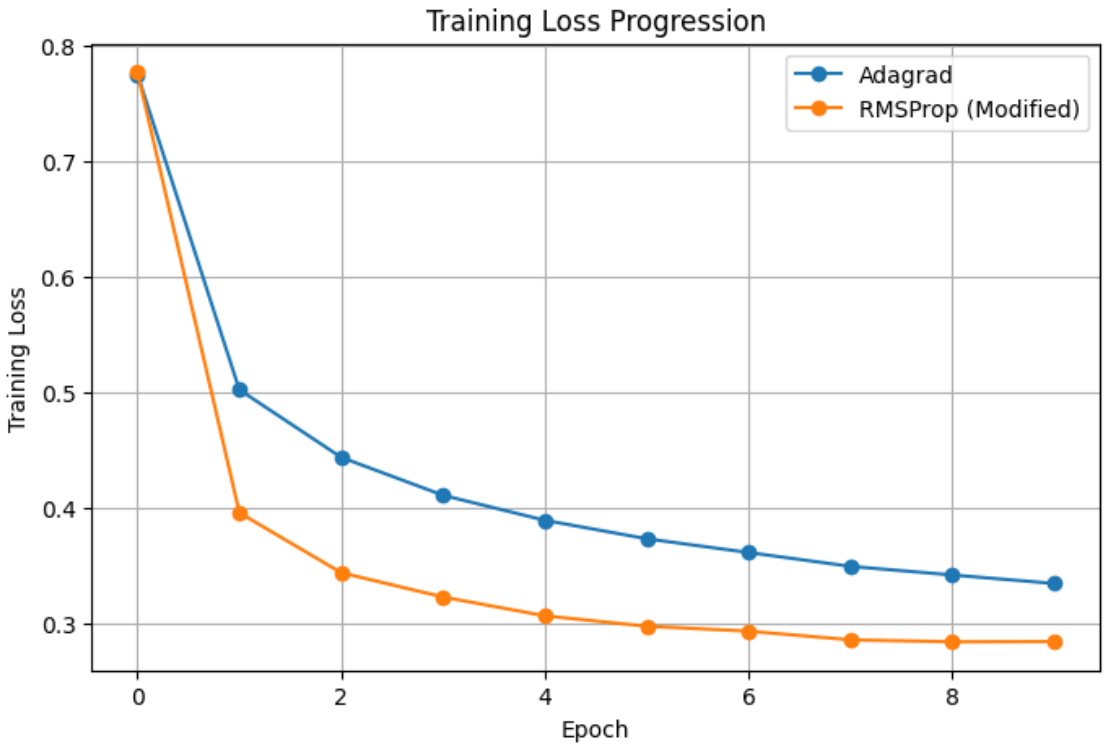
*This specific modification effectively transforms **Adagrad** into the **RMSProp** algorithm. By introducing a "forgetting" factor (γ), it prevents the learning rate from vanishing. This mechanism is also a foundational component of the **Adam** optimizer, making it highly effective for training deep neural networks.*

[ ]:
```
net_rmsprop = get_net()
optimizer_rmsprop = torch.optim.RMSprop(net_rmsprop.parameters(), lr=0.01,␣
 ↪alpha=0.9)
loss_rmsprop, acc_rmsprop = train_and_record(net_rmsprop, optimizer_rmsprop,
                                             num_epochs=10, name="RMSProp␣
 ↪(Modified)")
```

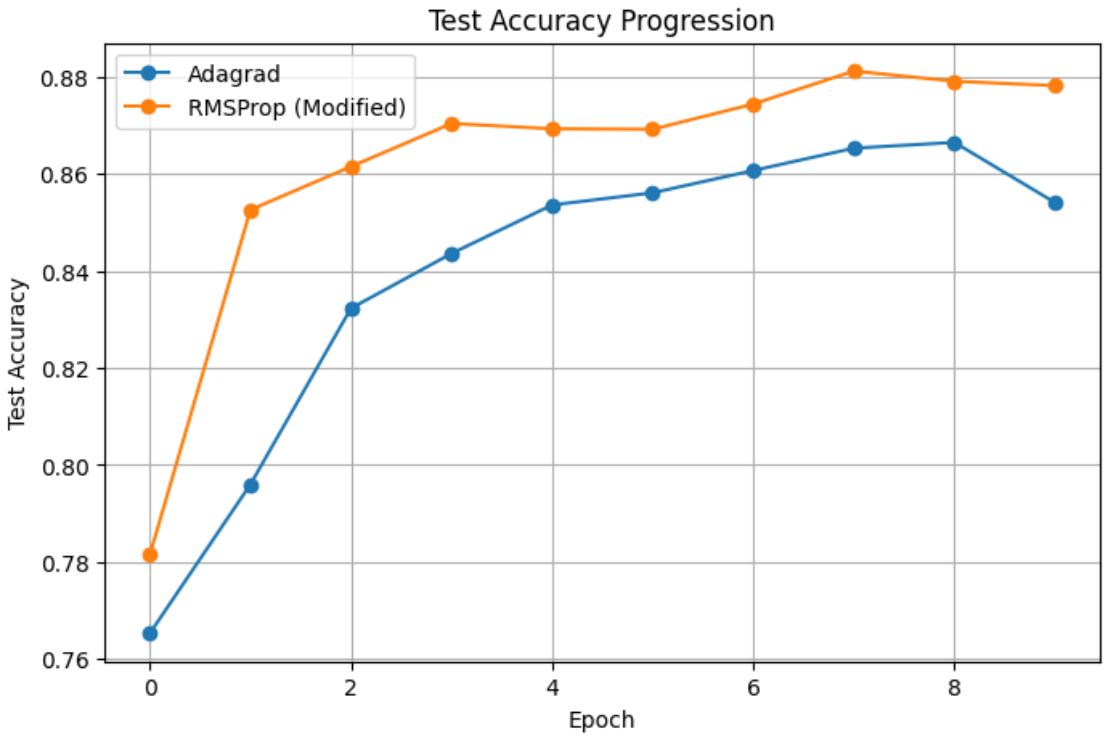
```
--- Starting Training: RMSProp (Modified) ---
Epoch [1/10] Loss: 0.7771 | Acc: 0.7815
Epoch [2/10] Loss: 0.3956 | Acc: 0.8526
Epoch [3/10] Loss: 0.3437 | Acc: 0.8615
Epoch [4/10] Loss: 0.3226 | Acc: 0.8704
Epoch [5/10] Loss: 0.3064 | Acc: 0.8693
Epoch [6/10] Loss: 0.2973 | Acc: 0.8692
Epoch [7/10] Loss: 0.2931 | Acc: 0.8744
Epoch [8/10] Loss: 0.2856 | Acc: 0.8812
Epoch [9/10] Loss: 0.2839 | Acc: 0.8791
Epoch [10/10] Loss: 0.2841 | Acc: 0.8782
```

```
Total time: 81.3s
```

[ ]:
```
visualize_history([loss_adagrad, loss_rmsprop],
                  ["Adagrad", "RMSProp (Modified)"],
                  "Training Loss")
```



[ ]:
```
visualize_history([acc_adagrad, acc_rmsprop],
                  ["Adagrad", "RMSProp (Modified)"],
                  "Test Accuracy")
```



6 References

[1] A. Zhang, Z. C. Lipton, M. Li, and A. J. Smola, Dive into Deep Learning. Cambridge University Press, 2023. [Online]. Available: https://D2L.ai

[2] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," J. Mach. Learn. Res., vol. 12, no. 61, pp. 2121–2159, 2011.

[ ]: