# Homework 1

## Exercise 1

Here is the step-by-step explanation of the implementation, linking the Python code to the mathematical theory of Gradient Descent.

### 1. Defining the Optimization Problem

First, we define the mathematical landscape we want to navigate. In Machine Learning, this corresponds to the Loss Function $\mathcal{L}(\Theta)$, which measures the error of our model.

```
def loss_fn_1(theta):
    return (theta - 3)**2 + 1

def gradient_fn_1(theta):
    return 2 * (theta - 3)
```

- **Function Parameters & Types:**
- Both functions accept `theta`. In the context of the loop, `theta` is a **scalar (float)** representing the current parameter value. However, thanks to NumPy's polymorphism, `loss_fn_1` can also accept a **NumPy array**, which allows for element-wise calculation later.
- **Code & Theory Connection:**
- `loss_fn_1` implements the objective function $\mathcal{L}(\theta) = (\theta - 3)^2 + 1$. This is a convex function with a global minimum at $\theta = 3$.
- `gradient_fn_1` implements the gradient $\nabla\mathcal{L}(\theta)$. Analytically, the derivative is $2(\theta - 3)$. The gradient points in the direction of the steepest ascent. To find the minimum, the algorithm must move in the opposite direction of this value.

---

### 2. The Gradient Descent Algorithm

This is the core engine. The function `GD` encapsulates the iterative logic required to minimize the loss function.

```
def GD(loss_function, gradient_function, start_theta, learning_rate=0.01, n_iterations=100)
    theta = start_theta
    theta_history = [theta]
```

- **Parameters & Types:**
- `loss_function`, `gradient_function`: **Callables** (Python functions). This makes the code modular; you can pass any differentiable function here.

- `start_theta`: **Float**. The initial guess $\Theta^{(0)}$.
- `learning_rate`: **Float**. Represents $\eta$ (eta), the step size.
- `n_iterations`: **Int**. The stopping criterion, defined as the maximum number of steps (`maxit`).
- **Initialization:** We initialize `theta` at the starting point and create a list `theta_history` to store the trajectory.

---

## 3. The Iterative Update Loop

We now enter the loop that performs the actual optimization steps.

```
for k in range(n_iterations):
    grad = gradient_function(theta)

    theta = theta - learning_rate * grad
    theta_history.append(theta)
```

**Code & Theory Connection:**

- **Gradient Calculation:** `grad = gradient_function(theta)` computes $\nabla\mathcal{L}(\Theta^{(k)})$.
- **The Update Rule:** The line `theta = theta - learning_rate * grad` is the direct translation of the Gradient Descent update formula found in the theory:

$$\Theta^{(k+1)} = \Theta^{(k)} - \eta\nabla\mathcal{L}(\Theta^{(k)})$$

- By subtracting the gradient (scaled by $\eta$), we move $\theta$ towards the minimum. If the learning rate is too small, convergence is slow; if too large, it may oscillate or diverge.
- By subtracting the gradient (scaled by ), we move towards the minimum. If the learning rate is too small, convergence is slow; if too large, it may oscillate or diverge.

---

## 4. Vectorized Loss Calculation

After the loop finishes, we perform a crucial optimization for efficiency and analysis.

```
theta_history = np.array(theta_history)
loss_history = loss_function(theta_history)
return theta_history, loss_history
```

- **Code Explanation:**

- Instead of calculating the loss value inside the loop (which would require calling the function $N$ times scalar-wise), we convert the list of positions into a NumPy array.
- We then pass this entire array to loss_function. Thanks to NumPy's broadcasting, the mathematical operation $(\theta - 3)^2 + 1$ is applied to every element in the vector simultaneously.
- **Return Types:** The function returns two **NumPy arrays** containing the sequence of parameters and their corresponding loss values.

---

## 5. Execution and Hyperparameter Selection

Finally, we run the algorithm using different learning rates to observe how step size affects convergence.

```
start_theta = 0.0
n_iter = 50
etas = [0.05, 0.2, 1.0]

results = {}

for eta in etas:
    thetas, losses = GD(loss_fn_1, gradient_fn_1, start_theta, eta, n_iter)
    results[eta] = {
        'thetas': thetas,
        'losses': losses
    }
```

- **Code & Theory Connection:**
- We define a starting point $\Theta^{(0)} = 0.0$. Since the function is convex, the choice of $\Theta^{(0)}$ is less critical as it will ideally converge to the global minimum regardless.
- We iterate through different values of $\eta$ (`etas`). This allows us to empirically verify the theoretical behavior of step sizes:
- Small $\eta$ (0.05) leads to slow, steady convergence.
- Moderate $\eta$ (0.2) leads to efficient convergence.
- Large $\eta$ (1.0) might cause the parameter to bounce back and forth around the minimum.
- The results are stored in a dictionary to facilitate the comparison of trajectories and loss reduction over time.

# Analysis of Gradient Descent Behavior Across Different Learning Rates

In this section, we analyze the impact of the hyperparameter $\eta$ (learning rate) on the algorithm's ability to minimize the quadratic cost function $L(\theta) = (\theta - 3)^2$.

---

## 2. Discussion of Results

### A. Conservative Learning Rate ($\eta = 0.05$)

- **Final Result:** $\theta \approx 2.9845$
- **Analysis:** The algorithm shows a monotonic and very stable descent, free of oscillations. However, the step size is too small relative to the curvature of the function. After 50 iterations, the theoretical minimum ($\theta = 3$) has not yet been reached.
- **Conclusion:** This approach is safe but computationally inefficient (underestimated learning rate).

### B. Balanced Learning Rate ($\eta = 0.2$)

- **Final Result:** $\theta = 3.0000$
- **Analysis:** This represents ideal behavior. The initial steps are large due to the high gradient, progressively reducing as the algorithm approaches the minimum. Convergence to 3.0 is achieved quickly, well before the iteration limit.
- **Conclusion:** $\eta = 0.2$ is the optimal value for this specific error surface.

### C. Excessive Learning Rate ($\eta = 1.0$)

- **Final Result:** $\theta = 0.0000$ (Stagnation)
- **Analysis:** The Loss graph is a flat line, and the trajectory shows two fixed points on opposite sides of the parabola ($\theta = 0$ and $\theta = 6$).
  - The algorithm calculates an update step so large that it entirely overshoots the minimum and lands on the opposite side of the curve, at exactly the same height (same Loss).
  - On the next step, the opposite gradient sends it back to the starting point.
- **Conclusion:** The algorithm has entered an infinite oscillation cycle without ever converging. This demonstrates the risks of a learning rate that is too high.

---

## 3. Summary Table

| Eta ($\eta$) | Final $\theta$ | Behavior |
|---|---|---|
| **0.05** | 2.9845 | Slow; does not reach target in 50 steps. |
| **0.20** | **3.0000** | **Optimal Convergence.** |
| **1.00** | 0.0000 | Persistent oscillation (Bounce). |

## Exercise 2

### Discussion of Results

Based on the implemented **Gradient Descent with Backtracking** and the resulting plots, here is the analysis of the algorithm's behavior on the non-convex function $\mathcal{L}(\theta) = \theta^4 - 3\theta^2 + 2$.

**1. Why different initializations converge to different minima** Gradient Descent is a **local optimization algorithm**. It determines the direction of the update based solely on the gradient at the current point, without knowledge of the global landscape. The function used here is **non-convex**, possessing two distinct local minima (at $\theta \approx \pm 1.22$) and a local maximum at $\theta = 0$.

- **Basins of Attraction:** The landscape is divided into "basins of attraction" separated by the local maximum (the "hill" at $\theta = 0$).
- **Initialization at $\theta_0 = -2.0$:** The point starts on the left side of the hill. The negative gradient pulls it further to the right, but it stays within the left basin, eventually converging to the **left local minimum**.
- **Initialization at $\theta_0 = 2.0$:** Similarly, this point starts in the right basin and converges to the **right local minimum**.
- **Initialization at $\theta_0 = 0.5$:** This point is crucial. Although it is numerically close to 0, it lies on the positive slope of the local maximum ($\theta > 0$). The gradient points strictly towards the right valley. Consequently, the algorithm treats it as belonging to the right basin of attraction and converges to the **right local minimum**, same as $\theta_0 = 2.0$.

**2. How backtracking automatically chooses a suitable step size** The polynomial $\theta^4$ creates a landscape with varying curvature: it is very flat near the minima but becomes extremely steep as $|\theta|$ increases.

- **Mechanism:** At each iteration, the backtracking algorithm proposes a large step (initial $\eta = 1.0$). It checks the **Armijo condition**: does this step decrease the loss sufficiently relative to the gradient's magnitude?
- **Observation:** In the "Loss vs. Iterations" plots, we see a monotonic decrease in loss without oscillations. This indicates that when the algorithm was at steep areas (e.g., $\theta = -2$), the Armijo condition failed for $\eta = 1.0$, forcing the inner loop to multiply $\eta$ by $\beta$ (shrinking it) until the step was safe.
- **Adaptivity:** As the parameters approached the flat valley of the minimum, the gradient magnitude decreased. The backtracking line search

likely accepted larger step sizes (closer to the initial $\eta$) because the risk of overshooting was lower, allowing for precise convergence.

**3. Situations where constant step size would fail** A constant step size approach is highly sensitive to the scale of the gradient, which poses a major problem for this specific function.

- **The Steepness Problem:** The gradient is $\nabla\mathcal{L} = 4\theta^3 - 6\theta$. At $\theta = -2$, the gradient is $-20$. At $\theta = -3$, it is $-90$. The magnitude grows cubically.
- **Divergence:** If we chose a constant step size suitable for the detailed convergence near the minimum (e.g., $\eta = 0.1$), applied at $\theta = -3$, the update would be roughly $\Delta\theta \approx 0.1 \times 90 = 9.0$. This massive jump would send the parameter to the opposite side of the graph, likely even further out where the gradient is steeper, leading to numerical overflow (divergence).
- **Oscillation:** If we chose a very small constant step size to prevent divergence at the edges (e.g., $\eta = 0.001$), the convergence near the flat minima would become excruciatingly slow, requiring thousands of iterations instead of the $\approx 5$ iterations achieved here with backtracking.
-

## Exercise 3

### Code Explanation

The Python script implements the standard Gradient Descent (GD) algorithm with a constant step size to minimize a simple quadratic function $\mathcal{L}(\Theta) = \frac{1}{2}\Theta^T A\Theta$.

- **Loss Function ($\mathcal{L}$):** Defined by the diagonal matrix $A = \text{diag}(1, 25)$. This structure means the loss is $\mathcal{L}(\theta_1, \theta_2) = 0.5 \cdot (1 \cdot \theta_1^2 + 25 \cdot \theta_2^2)$.
- **Gradient Function ($\nabla\mathcal{L}$):** Calculated as $A\Theta$, specifically $[\theta_1, 25\theta_2]^T$.
- **GD Function:** Performs the iterative update $\Theta^{(k+1)} = \Theta^{(k)} - \eta\nabla\mathcal{L}(\Theta^{(k)})$.
- **Experiment Setup:** The code tests three different constant learning rates ($\eta = 0.02, 0.05, 0.1$) starting from the same point, $\Theta_0 = [5.0, 1.0]$.
- **Visualization:** The resulting plots show the **level sets** (contours) of the ill-conditioned loss function overlaid with the path (trajectory) taken by the GD algorithm for each learning rate.

---

### Analysis of Results: Ill-Conditioning and Geometry

The plots perfectly illustrate the challenges Gradient Descent faces when optimizing an **ill-conditioned** objective function.

| Observation | $\eta = 0.02$ | $\eta = 0.05$ | $\eta = 0.1$ |
|---|---|---|---|
| **Path Behavior** | Slow but stable convergence. | Clear zig-zag and oscillation. | Rapid divergence/explosion. |

**1. The Elongated Ellipses Produced by Ill-Conditioning** The contour lines (level sets) are not circular but are highly **elongated** along the $\theta_1$ axis and compressed along the $\theta_2$ axis. This geometry is a direct result of the large **condition number** (ratio of the largest to smallest eigenvalues: $25/1 = 25$). The function is 25 times steeper in the $\theta_2$ direction than in the $\theta_1$ direction. This means movement along $\theta_1$ yields little change in loss, while movement along $\theta_2$ yields massive changes.

**2. The Gradient Direction Compared to the Level Set Lines** At any point on the trajectory (except near the origin), the gradient vector (which dictates the step direction) is always **perpendicular** to the contour line passing through that point. Because the contours are stretched, the perpendicular direction points severely toward the steepest edge ($\theta_2$), rather than pointing directly toward the target minimizer $(0, 0)$. This forces the algorithm to spend its initial steps correcting the steep dimension ($\theta_2$), rather than making progress toward $\theta_1$.

**3. Zig-Zag Behaviour for Large Condition Numbers** This behavior is clearly visible in the middle plot ($\eta = 0.05$). * When the learning rate is too large relative to the curvature in the steep dimension ($\theta_2$), the step **overshoots** the minimum along that axis, landing on the opposite side of the valley. * The subsequent step corrects the overshoot, but again overshoots the minimum. * This results in the characteristic back-and-forth **zig-zag pattern**, wasting steps correcting the same error repeatedly, preventing smooth convergence down the valley floor. * For $\eta = 0.1$, the overshooting is so massive that the trajectory explodes outside the visible range (or oscillates between extremely large values).

**4. The Relation to Slow Convergence in Narrow Valleys** The plot for $\eta = 0.02$ (left) demonstrates the trade-off. * The algorithm quickly reduces the error in the steep $\theta_2$ dimension (in the first few steps), successfully navigating the narrow valley width. * However, because $\eta$ must be kept small (to avoid zigzagging in $\theta_2$), the movement along the shallow $\theta_1$ direction becomes extremely slow. The trajectory spends the majority of its iterations inching horizontally along the $\theta_1$ axis, demonstrating the inefficiency of GD in directions with low curvature. The convergence speed is dominated by the weakest (shallowest) direction. *

## Exercise 4

The plots exhibit the expected theoretical behavior for a quadratic problem: 1. **Left Plot:** The **Exact Line Search (Blue)** shows the characteristic “orthogonal” path where every turning point is tangent to a level set (the new gradient is orthogonal to the previous search direction). The **Backtracking (Red)** follows a similar zig-zag path but with slightly different step lengths determined by the Armijo condition. 2. **Right Plot:** Both methods show **linear convergence** (which appears as a straight line on a semi-log scale), which is the standard behavior for Gradient Descent on strongly convex functions.

### Brief Code Description

The code solves the optimization problem for the quadratic loss $\mathcal{L}(\theta) = \frac{1}{2}\theta^T A\theta$ where $A = \text{diag}(5, 2)$. * **Exact Line Search:** Calculates the mathematically optimal step size $\eta_k$ at each iteration using the formula $\eta_k = \frac{g_k^T g_k}{g_k^T A g_k}$. This ensures the maximum possible drop in loss along the current gradient direction. * **Backtracking:** Uses a heuristic loop. It starts with a large $\eta$ (e.g., 1.0) and iteratively shrinks it by a factor $\beta$ until the **Armijo condition** is met (sufficient decrease in loss). * **Visualization:** It compares the two methods by plotting their spatial trajectories over the contour map and their loss decay over time.

---

### Analysis of Results

- **Speed of Convergence:**
  - **Early Stages:** The **Exact Line Search (Blue)** is generally faster in the first few iterations (as seen in the trajectory plot, it gets to the center “valley” very quickly). This is because it computes the greedy optimal step.
  - **Long Term:** Interestingly, in your log-plot, **Backtracking (Red)** actually achieves a lower loss in later iterations (steeper slope). This is a known phenomenon: while Exact Line Search is locally optimal (greedy), it forces the algorithm into a rigid “canonical zig-zag” pattern determined entirely by the condition number of matrix $A$. Backtracking, by being slightly “sub-optimal” and noisy, can sometimes accidentally step into a more favorable position that breaks the worst-case zig-zag cycle, allowing for slightly faster asymptotic convergence in specific setups.
- **Smoothness of Step-sizes:**
  - **Exact Line Search:** The step sizes are **mathematically determined** and “smooth” in the sense that they follow a strict geometric rule (steps are always orthogonal to the next gradient). The trajectory looks like a clean, rigid geometric pattern.
  - **Backtracking:** The step sizes are **discrete and adaptive**. You might notice the red path takes steps that are sometimes larger or

smaller in a less predictable pattern than the blue path. This is because the step size $\eta$ depends on how many times the `while` loop runs to satisfy the Armijo condition. If the condition is met immediately with $\eta = 1$, a large step is taken; if not, the step is drastically cut.

## Exercise 5

### Code Description (New Parts)

Compared to previous exercises involving quadratic forms, this code introduces three key changes:

1. **Non-Convex Objective Function:** The code implements the **Rosenbrock function** (`rosenbrock_loss`) and its manually derived gradient (`rosenbrock_grad`). Unlike the previous matrix-based quadratic functions ($x^T Ax$), this function includes non-linear terms (squares of squares), creating a complex, non-convex landscape with a curved "banana-shaped" valley.
2. **Logarithmic Visualization:** Because the Rosenbrock function creates extremely high loss values away from the valley but very low values near the minimum, the code uses `np.logspace` for contour levels and sets the loss-vs-iteration y-axis to a logarithmic scale (`ax2.set_yscale('log')`) to visualize convergence meaningfuly.
3. **Multiple Constants Comparison:** The execution loop now tests three specific small constant step sizes ($\eta \in \{10^{-3}, 10^{-4}, 10^{-5}\}$) alongside Backtracking for every starting point to highlight the sensitivity of hyperparameters.

---

### Analysis of Results

Based on the generated plots, here is the discussion regarding the behavior of Gradient Descent on the Rosenbrock landscape:

**1. Whether the method enters the valley Yes, all methods successfully enter the valley.** In all four starting scenarios, even the smallest step sizes eventually descend from the high steep sides into the parabolic valley ($\theta_2 \approx \theta_1^2$). This is because the gradient magnitude on the "walls" of the valley is massive, forcing the algorithm quickly toward the floor of the valley regardless of the specific step size (provided it doesn't diverge).

**2. How long it takes to "turn" correctly along the valley direction It takes a significant amount of iterations, and constant step sizes struggle immensely here.** Once inside the valley, the gradient points mostly toward the steep walls rather than along the gentle slope toward the global minimum $(1, 1)$. * **Constant Step Sizes:** Methods with small $\eta$ (e.g., $10^{-4}, 10^{-5}$)

get "stuck" bouncing slightly or moving infinitesimally slowly along the valley floor. They struggle to align the descent direction with the curvature of the banana shape. * **Backtracking:** It adapts much faster. For example, in the `Start: [-1.5, 2.0]` plot, the red line enters the valley and makes the sharp right turn toward $(1, 1)$ much earlier than the constant step approaches.

**3. Whether the method zig-zags Yes, zig-zagging is prevalent, especially for constant step sizes.** The Rosenbrock valley acts like a narrow canyon. * **Constant $\eta$:** If the step size is slightly too large (e.g., $\eta = 10^{-3}$ in some regions), the trajectory bounces back and forth between the valley walls (oscillating) rather than moving down the center. This wastes computational effort. * **Backtracking:** While it also exhibits some zig-zagging when entering the valley (where gradients change rapidly), the adaptive line search helps dampen these oscillations by reducing the step size exactly when the curvature forces a turn, allowing for a smoother path compared to the aggressive constant steps.

**4. Whether step sizes become too small or too large**

- **Too Small (Constant $\eta$):** The values $\eta = 10^{-4}$ and $\eta = 10^{-5}$ (Orange and Blue lines) are far too small for this problem. While safe, their progress is agonizingly slow. In the log-plots, they appear nearly flat compared to backtracking, indicating they would require millions of iterations to reach the minimum.
- **Too Large / Divergence:** While explicit divergence isn't visually dominated in these specific plots (due to the small constants chosen), $\eta = 10^{-3}$ (Cyan) often shows instability or slower convergence than backtracking because it is too coarse for the intricate valley floor.
- **Backtracking Efficiency:** Backtracking strikes the balance. It utilizes large steps when far from the valley (descending quickly) and automatically shrinks the steps to navigate the narrow, curved valley floor without diverging, achieving the lowest loss in all test cases.

# Homework 2

## Exercise 2.1

### Analysis of Gradient Descent Variants

#### 1. Why Full Gradient Descent (GD) is smooth but slow for large $N$

- **Smoothness (Deterministic Trajectory):** Full Batch Gradient Descent computes the gradient using the **entire dataset** ($N$) for every single update step. Because it calculates the exact average gradient over all data points, the resulting vector points directly toward the steepest descent of the global cost function. This results in a smooth, deterministic trajectory without fluctuations.

- **Slowness (Computational Redundancy):** The downside is computational cost. If $N$ is 1,000,000, the algorithm must calculate the error and gradient for one million points just to move the parameters ($\theta$) a single tiny step. This makes it extremely slow per iteration and computationally expensive, especially if the data does not fit into memory.

**2. Why Stochastic Gradient Descent (SGD) is noisy but progresses faster**

- **Noise (High Variance):** SGD uses a **single random data point** (batch size = 1) to estimate the gradient. Since one data point is a very rough approximation of the whole dataset, the gradient estimate is "noisy." The algorithm might move in a direction that is optimal for that specific point but suboptimal for the dataset as a whole, causing the trajectory to "zigzag" or oscillate wildly.
- **Speed (Frequent Updates):** despite the zigzagging, SGD is often "faster" in terms of wall-clock time to reach a decent error rate. It updates the parameters $N$ times in a single epoch, whereas Full GD updates only once per epoch. This allows the model to learn immediately and continuously, often getting close to the minimum long before Full GD has finished its first few epochs.

**3. How batch size affects noise level and convergence stability** The batch size controls the trade-off between the accuracy of the gradient estimate and the speed of computation: * **Small Batch (High Noise):** With a small batch size, the variance of the gradient estimate is high. This noise prevents the model from settling perfectly into the minimum (it tends to wander around it), requiring a decaying learning rate to force convergence. However, this noise can be beneficial as it adds a form of regularization and helps the model escape shallow local minima or saddle points. * **Large Batch (High Stability):** As batch size increases, the gradient estimate approaches the true gradient (Full GD). The path becomes smoother and convergence is more stable, allowing for larger learning rates. However, if the batch is too large, you lose the computational benefits of stochastic updates and the "exploration" benefits of noise. * **The "Sweet Spot":** Mini-batch GD (e.g., sizes 32, 64, 128) is usually preferred because it utilizes matrix vectorization (SIMD instructions) efficiently to speed up calculation while maintaining enough stochastic noise to generalize well.

## Exercise 2.2

### Discussion: Variance of the Stochastic Gradient

**1. Why the variance decreases with larger batches**

- **Averaging out the Noise:** The data inherently contains noise (random fluctuations not representative of the true underlying pattern). When the batch size ($N_{batch}$) is small (e.g., 1), the gradient calculation is dominated by the noise of that single data point, leading to a high variance.

- **Law of Large Numbers:** As you increase the batch size, you are averaging the gradient over more data points. According to the Law of Large Numbers, the average of a sample converges to the expected value (the true full-batch gradient) as the sample size increases. The random "errors" (noise) in individual points tend to cancel each other out, driving the variance toward zero.
- **Zero Variance at Full Batch:** As seen in the plot at $N_{batch} = 200$ (the full dataset size $N$), the variance drops to effectively zero. This is because there is no longer any "random sampling"; every "sample" is identical to the full dataset, yielding the exact same deterministic gradient every time.

**2. Why SGD becomes more stable as $N_{batch}$ increases**

- **Consistent Direction:** Stability in this context refers to how consistently the gradient vector points toward the true minimum of the loss function. With a higher $N_{batch}$, the calculated gradient $g_k$ is a more accurate approximation of the true gradient $\nabla_\Theta \mathcal{L}$.
- **Smoother Updates:** Because the variance is lower, the difference between consecutive gradient updates is smaller. This reduces the "jitter" or "zigzagging" effect seen in pure SGD, preventing the optimization algorithm from making erratic jumps in the wrong direction due to a single noisy outlier.

**3. The trade-off between stability and computational cost**

- **Computational Cost:** Computing the gradient for a large batch requires performing matrix operations on large tensors. This increases the CPU/GPU memory requirement and the time required to compute a single update step.
- **Convergence Efficiency:** While small batches (low stability) are noisy, they are computationally cheap and allow for many updates per second. Large batches (high stability) provide accurate updates but are expensive.
- **The Sweet Spot (Mini-Batch):** The ideal trade-off is usually found in "Mini-Batch GD" (e.g., sizes 32 to 128). This range provides enough variance reduction to ensure stable convergence (as seen in the steep drop in your plot between sizes 1 and 20) while maintaining the computational speed of stochastic updates.

## Exercise 2.3

**1. How noise helps escape shallow minima or bad regions**

- **Breaking Symmetry and Saddle Points:** In the specific landscape of the function provided (which resembles a Rosenbrock function), there is a "ridge" or saddle point region near Θ_1 = 0. As seen in the trajectory with **No Noise (Blue line)**, standard Gradient Descent follows the exact gradient path. If initialized perfectly on the axis of symmetry (e.g., at $\Theta$_1 = 0), the gradient with respect to $\Theta$_1 is zero. The algorithm gets "stuck" on this ridge or moves extremely slowly because the deterministic gradient does not provide any lateral force to push it toward the basins of attraction at $\Theta$_1 = $\pm$1.

- **Exploration via Fluctuation:** The **Moderate Noise (Orange line)** and **High Noise (Green line)** introduce random perturbations ($\epsilon$_$k$) to the update step. Even if the true gradient is zero (or very small), the random noise "kicks" the parameters sideways. Once the parameters are knocked off the unstable ridge, the gradient becomes non-zero and strong, quickly guiding the optimization into the deeper valleys towards the global minima (marked with red Xs). Essentially, noise turns the optimization into an exploration process that prevents the model from stalling in flat regions or saddle points.

**2. How too much noise prevents convergence**

- **Inability to Settle:** While noise is beneficial for exploration, it becomes detrimental during the exploitation phase (fine-tuning). As seen in the plot with **High Noise (Green line)**, particularly with the larger learning rate, the trajectory behaves erratically.
- **The "Bounce" Effect:** Near the minimum, the true gradient $\nabla\mathcal{L}$ approaches zero. However, the noise term $\epsilon$_$k$ remains constant (with variance $\sigma^2 = 0.5$). Consequently, the update step is dominated by noise rather than the gradient signal. Instead of settling at the precise optimal point $[-1, 1]$, the parameters oscillate violently around it. This prevents the loss from stabilizing at the lowest possible value, as the algorithm constantly "overshoots" the target due to the high variance of the stochastic updates.

## Exercise 2.4

### 1. Why GD gives a smooth curve and SGD oscillates

The smoothness of the loss curve is determined by the consistency of the gradient calculation. In **Full Gradient Descent (Black line)**, the algorithm computes the gradient using the entire dataset ($N = 1338$) for every single update. Because the data does not change between iterations, the calculated gradient vector points precisely in the direction of the steepest descent for the global loss surface. This results in a deterministic, monotonic decrease in error, creating the smooth curve visible in the plot.

In contrast, **SGD with Batch Size = 1 (Red line)** approximates the gradient using a single, randomly selected example. This individual sample may have a label ($y$) that is an outlier or simply noisy compared to the average. Consequently, the gradient calculated from it might point in a direction that increases the global error, even if it decreases the error for that specific point. This high variance in the gradient estimate causes the “zigzagging” or oscillation seen in the loss plot and the erratic spikes in the gradient norm.

### 2. Why larger batches reduce noise but cost more per iteration

There is a direct trade-off between statistical stability and computational effort. As observed in the results, **Batch Size = 50 (Blue line)** is significantly smoother and converges faster than **Batch Size = 1 (Red line)**. This happens because averaging gradients over 50 samples reduces the variance of the estimate (by a factor of roughly $1/\sqrt{50}$), effectively cancelling out the noise from individual outliers.

However, this stability comes at a cost. Computing the gradient for a batch of 50 requires 50 times more matrix multiplication operations (or vector dot products) than a batch of 1. While modern hardware (SIMD/GPUs) handles vectorization efficiently, making batches of 32-64 very fast, increasing the batch size indefinitely (up to Full GD) eventually leads to memory bottlenecks and diminishing returns in convergence speed per second of computation.

### 3. Why all methods roughly converge to the same region

The loss function for Linear Regression (Mean Squared Error) is **convex**, meaning it is shaped like a bowl with a single global minimum. Regardless of the path taken—whether it is the direct line of Full GD or the “drunk walk” of SGD—gravity (the gradient) eventually pulls the parameters toward the bottom of this bowl.

Comparing your final parameters confirms this:

- **Full GD:** `[0.001, 0.273, 0.168, 0.057]`
- **SGD (Batch=50):** `[-0.000, 0.279, 0.166, 0.056]`

These values are nearly identical because both methods reached the basin of the minimum. The slight deviation seen in **SGD Batch=1** (`[-0.107, ...]`) suggests that while it reached the correct *region*, the fixed learning rate was too high to allow it to settle perfectly into the minimum, causing it to bounce around the optimal point (noise floor).

### 4. Why SGD is more suitable for large datasets, even when noisy

While Full GD (Black line) is smooth, notice that it descended much slower than the SGD methods; even at epoch 200, it is just reaching the loss levels that SGD (Batch=50) reached around epoch 10.

For very large datasets (e.g., $N = 1,000,000$), Full GD requires processing one million records just to take **one** step. In contrast, SGD updates the parameters after seeing only a few examples. If the dataset contains redundant information (common in large data), SGD can make thousands of useful updates and reach a "good enough" solution before Full GD has even finished its first epoch. The noise of SGD is a small price to pay for the massive speed advantage in terms of *updates per second*.

# Homework 3

## Exercise 3.1

**. Discussion: Why is the decision boundary linear?**

- **Mathematical Justification:** Logistic Regression makes a prediction based on the probability $P(y = 1|x) = \sigma(z)$, where $\sigma$ is the sigmoid function and $z = \Theta^T x$. The **decision boundary** is the threshold where the model is maximally uncertain, meaning the probability of being class 1 is exactly 0.5.

  Since $\sigma(0) = 0.5$, this boundary occurs precisely when the linear input $z$ is zero:

$$\Theta^T x = \theta\_0 + \theta\_1 x\_1 + \theta\_2 x\_2 = 0$$

- **Geometric Interpretation:** The equation above is a linear equation in the input variables $x\_1$ and $x\_2$. In 2D space, this defines a straight line. If we were in higher dimensions (D>2), this equation would define a flat hyperplane. Therefore, despite using a non-linear activation function (sigmoid) to output probabilities, the boundary separating the classes remains linear in the feature space.

## Exercise 3.2

### 1. Theoretical Discussion

**Why do gradients become noisier for small batches?**

The gradient computed on a mini-batch is an **statistical estimate** of the true gradient (the gradient calculated on the entire dataset).

- **Mathematical Intuition:** If the dataset has variance $\sigma^2$, the variance of the mean of a batch of size $m$ is approximately $\frac{\sigma^2}{m}$.
- **Batch Size = 1 (SGD):** You are estimating the direction of steepest descent based on a single data point. If that point is an outlier or has a label that contradicts its neighbors (noise), the computed gradient will point in a wildly different direction than the true gradient. This high variance is what we call "noise."

**Why do larger batches give smoother curves?**

- **Averaging Effect:** As you increase the batch size (e.g., to 10 or to $N$), you are averaging the gradient over more samples. The random fluctuations ("noise") from individual data points tend to cancel each other out.
- **Convergence to Truth:** As the batch size approaches the total dataset size $N$ (Full GD), the estimate approaches the true gradient. Consequently, the update steps become consistent and deterministic, leading to a smooth, monotonic descent towards the minimum.

### 2. Comments on the Obtained Results

Your plots highlight a critical distinction between "convergence per epoch" and "stability."

- **Convergence Speed (Loss vs Epoch):**
  - **SGD (Batch=1) [Blue Line]:** It converges the fastest in terms of epochs. By the end of Epoch 0, it has already reached a near-optimal loss. This is because in one single epoch, SGD performs $N = 400$ parameter updates (one for each point). Even though individual steps are noisy, the cumulative effect of 400 steps moves the model much further than the single step performed by Full GD.
  - **Full GD [Green Line]:** It is the slowest in terms of epochs. It makes only **one** update per epoch. It takes about 10-15 epochs to reach the accuracy that SGD reached in just 1 epoch.
- **Smoothness & Noise (Why does SGD look smooth here?):**
  - You might wonder why the SGD curve in your plot looks smooth and not "noisy" or "zig-zagging" as theory suggests.
  - **Reason:** Your code records the loss (`loss_hist.append`) only **at the end of each epoch**, after the inner loop has finished.

```
for _ in range(epochs):
for i in range(0, N, batch_size):
    # ... update theta ...
loss_hist.append(loss_f(theta, X, y)) # <--- Logged once per epoch
```

  If you were to plot the loss at every *step* (inside the inner loop), the Blue line (Batch=1) would look extremely jittery and noisy. However, looking at the aggregate result at the end of the epoch masks this noise, showing only the rapid convergence benefits of the frequent updates.

- **Accuracy:**

  All methods successfully reach 100% accuracy. SGD hits this target almost instantly due to the high frequency of updates, while Full GD takes a steadier, slower path.

## Exercise 3.3

### Discussion of Results

The results show near-perfect performance, which is expected given that the synthetic dataset generated in Exercise 1 consists of two well-separated Gaussian clusters. However, even with this "easy" dataset, we can observe the fundamental trade-offs in classification thresholds.

**1. How lower thresholds increase recall and lower precision**

- **Observation:** At a **threshold of 0.3**, your model is "liberal" or "optimistic" in predicting the positive class (Class 1). It classifies any sample with a probability $> 30$ as positive.
- **Result:** This strategy ensured **Recall was perfect (1.0)**, meaning no positive cases were missed (FN=0). However, it allowed one "False Positive" to slip through (FP=1), which slightly lowered the **Precision to 0.995**.
- **Theory:** Lowering the threshold reduces the standard of evidence required to predict "Positive." This catches more true positives (increasing Recall) but inevitably captures more noise or negative instances (increasing False Positives and lowering Precision).

**2. How higher thresholds increase precision and reduce recall**

- **Observation:** When you raised the **threshold to 0.7**, the model became "conservative" or "strict." It only predicted "Positive" if it was extremely confident ($> 70$ probability).
- **Result:** This strictness filtered out the single False Positive encountered earlier (FP dropped to 0). Consequently, **Precision increased to a perfect 1.0**. While Recall usually drops with higher thresholds, in this specific dataset the positive examples were so distinct that they all remained correctly classified (Recall stayed at 1.0).
- **Theory:** Raising the threshold acts as a filter that removes uncertain predictions. This minimizes False Positives (maximizing Precision) but increases the risk of rejecting actual positive cases that are subtle or weak (increasing False Negatives and lowering Recall).

**3. Why classification metrics depend on the application**

As discussed in class, there is no single "best" threshold; it depends entirely on the cost of errors in the specific real-world domain:

- **High Precision Preference (High Threshold):** In **Spam Detection**, we want to avoid False Positives at all costs (e.g., sending an important work email to the spam folder). We prefer to let some spam through (lower Recall) rather than lose legitimate mail.
- **High Recall Preference (Low Threshold):** In **Medical Diagnosis** (e.g., screening for a tumor), a False Negative is dangerous (telling a sick patient they are healthy). We lower the threshold to catch every possible

case (High Recall), accepting that this will lead to more False Positives (healthy people sent for further testing).

## Exercose 3.4

### Discussion of the results

Here is the consolidated answer incorporating all the requested considerations regarding the experiment on the Pima Indians Diabetes dataset.

#### 1. Why Normalization is Required

In the data preprocessing step, standardizing the features using `(X - mean) / std` is critical for three main reasons:

- **Conditioning of the Loss Surface:** Real-world features have vastly different ranges (e.g., “Age” $\approx 20-80$ vs. “Insulin” $\approx 0-800$). Without standardization, the loss function contours form highly elongated ellipses (deep, narrow valleys). Gradient Descent struggles in this landscape, bouncing back and forth across the valley walls instead of moving down the valley floor. Standardization makes the contours more spherical (well-conditioned), allowing the gradient to point more directly toward the minimum.
- **Stable Optimization:** With unscaled data, weights associated with large inputs (like Insulin) would need to be very small, while weights for small inputs (like Age) would need to be large. A single global learning rate $\eta$ cannot satisfy both requirements simultaneously, leading to instability or divergence.
- **Meaningful Gradient Magnitudes:** Standardization ensures that all features contribute approximately equally to the initial gradient updates, preventing one feature from dominating the learning process simply because it has larger numerical values.

#### 2. SGD vs. Adam Comparison

Looking at the comparison plots and the final metrics, we observe distinct behaviors:

- **Convergence Speed:**

  **Adam (Orange line)** converges significantly faster than **SGD (Blue line)**. In the Loss plot, Adam drops vertically in the first 10-20 epochs, reaching a loss of $\approx 0.50$ almost immediately. SGD, conversely, follows a slow, linear descent and barely reaches a loss of $\approx 0.51$ after 200 epochs.
- **Oscillation:**

While SGD (Batch=32) appears relatively stable due to batch averaging, it is "slow" to navigate the curvature. Adam appears smoother in its descent because it effectively dampens oscillations in high-variance directions (via the momentum term $m\_t$) and accelerates in flat directions (via the variance term $v\_t$).

- **Adaptive Learning Rates:**

  The superior performance of Adam illustrates the power of adaptive learning rates.

  - **SGD** uses a fixed learning rate $\eta$ for all parameters. If $\eta$ is small enough to be stable for the most sensitive parameter, it is often too small for the others, leading to slow convergence.
  - **Adam** computes an individual effective learning rate for each parameter $\theta\_j$ by dividing by $\sqrt{\hat{v}\_t}$. This allows it to take large steps for parameters with small gradients (flat regions) and small, cautious steps for parameters with large gradients (steep regions), navigating the complex landscape of the Diabetes dataset much more efficiently.

### 3. Final Model Evaluation

The quantitative results confirm Adam's superiority on this task:

- **Accuracy:** Adam achieved **78.26%**, beating SGD's **76.43%**.
- **F1 Score:** Adam achieved **0.6514** vs. SGD's **0.6373**.
- **Precision:** Adam was notably more precise (**0.7393** vs. **0.6883**), meaning it produced fewer False Positives (55 vs. 72). This is often desirable in medical diagnostics to avoid unnecessary alarm, even though the Recall was slightly lower.

**Conclusion:** For real-world datasets with heterogeneous features, adaptive optimizers like Adam generally offer a better trade-off between speed and stability compared to vanilla SGD, minimizing the need for extensive hyperparameter tuning.

## 5th point answer

### 1. Which method converges faster? Adam converges significantly faster.

Looking at the "Loss: SGD vs Adam" plot, the difference is stark.

- **Adam (Orange Line):** The loss drops vertically, reaching a value below 0.55 within the first 20 epochs and stabilizing around 0.47 shortly after.

- **SGD (Blue Line):** The loss decreases linearly and very slowly. Even after 200 epochs, it has only reached $\approx 0.51$, a value that Adam surpassed in its first few iterations.

In terms of accuracy, Adam jumps to $\approx 77\%$ almost immediately, whereas SGD takes the full 200 epochs to slowly climb to $\approx 76\%$.

**2. Which oscillates more?**

- **SGD (in this specific case):** Appears very smooth, but this is deceptive. The smoothness here is due to the learning rate ($10^{-3}$) being too small for the flat regions of the loss surface. SGD is taking tiny, consistent steps down a long slope, never moving fast enough to "oscillate" across valley walls.
- **Adam:** Shows some jitter (oscillation) in the accuracy plot once it reaches the plateau. This is expected behavior for a fast optimizer: it quickly reaches the minimum and then "bounces" slightly around the optimal parameters because it is sensitive to the noise in the mini-batches. However, it does not suffer from the detrimental "zigzag" oscillation of simple SGD because momentum ($m\_t$) smoothes the trajectory.

**3. Relation to Adaptive Learning Rates (Class Theory)** The superior performance of Adam relates directly to the two adaptive components discussed in class:

- **Momentum ($\hat{m}\_t$):**

  Adam maintains a moving average of past gradients (momentum). This helps the model "gain speed" in directions where the gradient consistently points the same way, pushing it through flat plateaus where standard SGD would stall. This explains why Adam drops so fast in the early epochs compared to SGD.

- **RMSProp / Adaptive Scaling ($\frac{1}{\sqrt{\hat{v}\_t+\epsilon}}$):**

  This is the crucial "adaptive learning rate" component.

  - Standard SGD uses the same scalar learning rate $\eta$ for all parameters. If one feature has a steep gradient and another has a flat gradient, a single $\eta$ cannot satisfy both (it will be too slow for one or unstable for the other).
  - Adam divides the update by the square root of the accumulated squared gradients ($\sqrt{\hat{v}\_t}$).
    - For parameters with small gradients (flat regions): $\hat{v}\_t$ is small $\rightarrow$ the step size is **boosted** (division by a small number).
    - For parameters with large gradients (steep regions): $\hat{v}\_t$ is large $\rightarrow$ the step size is **dampened** (division by a large number).

**Conclusion:** Adam creates a custom learning rate for each parameter. It was able to identify that the "Diabetes" dataset loss landscape had varying curvatures and automatically boosted the step size for the slow parameters, allowing it to converge in a fraction of the time it took SGD.

# Homework 4

## Exercise 4.1

### 1. Code & Reconstruction Verification

Your implementation correctly performs the SVD decomposition and verification.

- **Reconstruction Error:** You obtained a Frobenius norm error of approximately `7.71e-15`.
- **Conclusion:** This value is effectively zero within the limits of machine precision (floating-point arithmetic). This numerically proves that $A$ can be perfectly reconstructed as the product $U\Sigma V^T$, satisfying the definition of SVD

### 2. Discussion of Singular Values

**Why do singular values appear in descending order?**

- **Definition:** By mathematical definition and convention in algorithms (like `np.linalg.svd`), singular values are always ordered such that $\sigma_1 \geq \sigma_2 \geq ... \geq 0$<source-footnote ng-version="0.0.0-PLACEHOLDER" _nghost-ng-c1831576998="">.<sources-carousel-inline ng-version="0.0.0-PLACEHOLDER" _nghost-ng-c2676262027=""><source-inline-chips _ngcontent-ng-c2676262027="" _nghost-ng-c3554374441="" class="ng-star-inserted"><source-inline-chip _ngcontent-ng-c3554374441="" _nghost-ng-c2970764686="" class="ng-star-inserted">
- **Maximizing Variance:** The first singular value $\sigma_1$ captures the direction of maximum variance (most information) in the data. The second, $\sigma_2$, captures the maximum remaining variance orthogonal to the first, and so on. This ordering is crucial for tasks like compression, as it allows us to truncate the series at $k$ and know we have kept the "best" possible $k$-dimensional approximation.

**Why do small singular values correspond to "less important" directions?**

- **Sum of Dyads:** The matrix $A$ can be written as a weighted sum of rank-1 matrices (dyads):

$$A = \sum_{i=1}^{r} \sigma_i u_i v_i^T$$

Here, $\sigma_i$ acts as the **weight** or importance score for the $i$-th component

- **Information Content:** If $\sigma_i$ is large, the term $u_i v_i^T$ contributes significantly to the structure of $A$. If $\sigma_i$ is very small, that term adds very little detail (often just noise). Removing terms with small singular values results in a compressed matrix $A_k$ that is still very close to the original $A$ in terms of distance (Frobenius norm) **Why does floating-point arithmetic make exact zeros rare?**

- **Numerical Noise:** In theory, if a matrix has rank $r < \min(m, n)$, the singular values $\sigma_{r+1}$ onwards should be exactly 0.
- **Reality:** In Python (and all computers), numbers are represented with finite precision (IEEE 754 standard). The tiny errors accumulated during the random generation of numbers and the iterative calculation of SVD mean that a theoretical "zero" often appears as a very small number (e.g., $10^{-16}$) rather than a hard 0. This is why we often look for a "gap" in singular values to determine the **numerical rank** rather than looking for exact zeros.

## Exercise 4.2

**Explain why SVD gives the *optimal* rank-k approximation.**

The SVD provides the optimal rank-$k$ approximation because of the **Eckart-Young-Mirsky Theorem**. Here is the intuition based on the provided text:

1. **Decomposition into Information Content:** The SVD allows us to write the matrix $A$ as a sum of rank-1 matrices (dyads):

$$A = \sum_{i=1}^{r} \sigma_i u_i v_i^T$$

where the scalar $\sigma_i$ represents the "weight" or **importance** of the $i$-th component <source-footnote ng-version="0.0.0-PLACEHOLDER" _nghost-ng-c1783869716="“>.<sources-carousel-inline ng-version="0.0.0-PLACEHOLDER" _nghost-ng-c1881558635=""><source-inline-chips _ngcontent-ng-c1881558635="" _nghost-ng-c3554374441="" class="ng-star-inserted"><source-inline-chip _ngcontent-ng-c3554374441="" _nghost-ng-c2970764686="" class="ng-star-inserted">

1. **Ordered Importance:** Since singular values are always sorted in descending order ($\sigma_1 \geq \sigma_2 \geq ... \geq 0$), the first term $\sigma_1 u_1 v_1^T$ captures the most "energy" (variance) of the matrix, the second captures the next most, and so on.
2. **Minimizing the Residual:** The error introduced by truncating the series at $k$ (i.e., approximating $A$ with $A_k$) is determined entirely by the singular values we *discarded*. Specifically, the Frobenius error is the square root of the sum of the squared discarded singular values:

$$||A - A_k||_F = \sqrt{\sum_{i=k+1}^{r} \sigma_i^2}$$

To make this error as small as possible, we must discard the *smallest* possible $\sigma$ values. SVD guarantees this by keeping the largest $\sigma$ values (indices 1 to $k$) and discarding the tail (indices $k+1$ to $r$). This ensures that $A_k$ is mathematically the closest rank-$k$ matrix to $A$.

## Exercise 4.3

### Discussion of Results

**1. How visual quality improves with k**

- **Low k (e.g., k=5):** The image is extremely blurry and blocky. Only the grossest features (light sky vs. dark coat) are visible. This represents the "mean" structure of the image without details.
- **Medium k (e.g., k=20 to 50):** Structural elements like the tripod legs and the camera shape become recognizable. The "ghosting" artifacts reduce significantly.
- **High k (e.g., k=100 to 200):** The image becomes nearly indistinguishable from the original to the human eye. Fine details, such as the texture of the grass, are restored.

**2. Why most of the "energy" is contained in the first singular values**

- **Information Concentration:** Singular values $\sigma_i$ represent the "importance" or information content of the corresponding rank-1 layer (dyad)
- **Redundancy in Data:** Real-world images (like the Cameraman) are highly structured; adjacent pixels are likely to be similar. This correlation means that a few dominant "patterns" (the first few singular vectors) can describe the majority of the image variance. The remaining thousands of singular values mostly capture high-frequency noise or very subtle details

**3. The trade-off between compression and fidelity**

- There is a direct inverse relationship: as **rank k** increases, the **fidelity** (visual quality) improves (error drops), but the **compression ratio** worsens (file size grows).
- **The Sweet Spot:** The goal is to find a $k$ where the error is low enough that the image looks good, but $k$ is still small enough to save significant space. In your results, **k=50** offers a strong balance: the image is clearly recognizable, yet you still save **80.45%** of the storage space.

**4. The connection between SVD and optimal low-rank approximation**

- The Eckart-Young-Mirsky theorem states that the matrix $X_k$ constructed by truncating the SVD to the top $k$ singular values is the **optimal** rank-$k$ matrix in terms of minimizing the error

- Mathematically, no other matrix of rank $k$ can produce a smaller Frobenius norm difference $||X - X_k||_F$ than the one produced by SVD. This guarantee makes SVD the “gold standard” for linear dimensionality reduction and compression tasks.

## Exercise 4.5

### 1. Analysis of Digits 3 vs. 4 (2D Projection)

- **Separation:** In the first scatter plot, we observe two distinct clusters. The digit **3** (yellow/lighter points) and the digit **4** (purple/darker points) are separated reasonably well along the first Principal Component (PC1).
- **Overlap:** There is a noticeable overlap region in the center where the clusters merge. This indicates that while the global structure of a “3” differs from a “4”, there are handwriting variations (e.g., sloppy writing) that make them pixel-wise similar in the lowest dimensions.
- **Generalization:** The test set points (marked with ‘x’ in) map consistently onto the regions defined by the training set. This confirms that the principal components learned from the training data successfully capture the true, invariant features of the digits, rather than just memorizing the training samples.

### 2. Comparison with Other Digit Pairs

The effectiveness of PCA depends heavily on the geometric distinctness of the classes:

- **Digits 1 vs. 7:**

  These digits separate relatively well, forming curved, elongated clusters. This is expected as “1” and “7” are visually distinct (straight line vs. angled corner), though slanted writing can cause some overlap.

- **Digits 5 vs. 8:**

  These clusters show **significant overlap** and are much harder to separate. “5” and “8” share many geometric features (loops and curves in similar locations), making them difficult to distinguish using only linear projections like PCA. This suggests that non-linear manifold learning (like t-SNE) or higher dimensions (>2) would be required for clear separation.

### 3. Analysis of 3D Projection ($k = 3$)

- **Observation:** The 3D scatter plot adds the third principal component (PC3) to the visualization.
- **Impact:** Adding dimensions helps resolve ambiguities. Points that appear to overlap in the 2D plane might be separated along the vertical Z-axis (PC3). This illustrates the trade-off in dimensionality reduction: reducing $d$ too much (e.g., to 2) sacrifices information that might be crucial for separating similar classes, while keeping more components ($k = 3$ or higher) retains more variance and separability.

## Exercise 4.6

### 1. Code & Results Evaluation

#### Step 3: Centroid Classifier

- **Vectorization:** You used `np.linalg.norm(..., axis=1)` correctly. This is much faster than looping through rows.
- **Broadcasting:** The line `Z_test - mu_3` effectively subtracts the centroid from every test sample simultaneously. This is the "Pythonic" way to handle this math.
- **Results:**
  - **Linear Accuracy:** 97.39%
  - **Centroid Accuracy:** 97.33%
  - **Observation:** The results are nearly identical. This implies that for digits 3 and 4, the clusters in PCA space are roughly **spherical** and have **similar spreads (variance)**. When two clusters are perfect spheres of equal size, the optimal decision boundary *is* the perpendicular bisector of the line connecting their centroids. Your Linear Classifier found essentially this same line.

#### Step 4: Comparison Visualization

- **The "Error" Plot:** This is the highlight of your work. By plotting the "Green Circles" (misclassified points) on top of the boundary and centroids, you instantly show *where* the model fails:
  - The failures are exclusively in the "twilight zone" between the two clusters.
  - There are no outliers deep inside the wrong cluster, which suggests the data is clean.

### 2. Addressing the "Repeat with other digits" section

The exercise asks you to repeat this for `{(1,7), (5,8), (2,3)}` and observe the changes. Since you haven't run those yet, here is a guide on **what to look for and how to interpret it** when you do.

The relationship between **Cluster Shape** and **Classifier Performance** will change depending on the digits:

#### Case A: Digits 1 vs 7 (Structural Similarity)

- **Expectation:** These might be harder to separate than 3 vs 4.
- **Cluster Shape:** The digit "1" often varies mostly in *angle* (slant). This creates a long, thin, elliptical cluster in PCA space (like a cigar).
- **Centroid vs. Linear:**

  - If the “1” cluster is very elongated, the **Centroid Classifier** might fail. It assumes clusters are round. It might misclassify a “1” that is far from the center (a very slanted “1”) as a “7”.
  - The **Linear Classifier** usually adapts better here because it cares about the *boundary*, not the center.

**Case B: Digits 5 vs 8 (Topological Similarity)**

- **Expectation:** This is often the hardest pair in MNIST because 5s and 8s share similar bottom loops and curvature.
- **Cluster Separation:** You will likely see the red and blue clouds **overlapping** significantly in the center.
- **Decision Boundary:** The boundary will cut through a dense region of points.
- **Metrics:** Accuracy will likely drop (perhaps to ~85-90% instead of 97%). You will see many more “Green Circles” (errors) mixed into the clusters.

**Case C: Digits 2 vs 3**

- **Expectation:** Moderately distinct.
- **Observation:** Look at the **Centroid Separation**. If the centroids are far apart relative to the spread of the data, accuracy will remain high.

### 3. Summary of Key Concepts

When you run those experiments, you are effectively testing **Geometry vs. Learning**:

| **Feature** | **Centroid Classifier** | **Linear Classifier (Logistic Regression)** |
|---|---|---|
| **Model Type** | **Generative / Geometric** | **Discriminative** |
| **Assumption** | Assumes classes are **spherical blobs** with equal variance. | Assumes classes are separated by a **flat plane** (line). |
| **Training** | Fast (just compute mean). No optimization loops. | Slower (requires Gradient Descent loops). |
| **Best for...** | Well-separated, round clusters. | Elongated clusters or touching clusters where the boundary isn’t in the middle. |