# CS 782: ADVANCED ML HOMEWORK 2

Gaurab Pokharel, FCD Samuel

Email:{gpokhare, sfrank22}@gmu.edu

**Statement of Contribution**: Gaurab and Sam collaborated on this homework. They came up with the solutions of the problems together by discussing them after class hours and divided the responsibility of typing the answers up amongst themselves for each part.

**Question** Let $(Q \in \mathbb{R}^{d\times d})$ be a symmetric matrix and positive semi-definite. We consider the following optimization problem:

$$\min_{x\in\mathbb{R}^d} f(x) = \|Q - xx^\top\|_F^2.$$

This problem has very important applications in machine learning, which tries to use a succinct representation $xx^\top$ to approximate the matrix $Q$. For example, $Q$ can be the attention matrix of the transformer, and we want to avoid storing the matrix of size $d \times d$ and only store a vector of size $d$.

1. What are the stationary points of $f(x)$, i.e., the $x$ such that $\|\nabla f(x)\| = 0$?
   Given $f(x) = ||Q - xx^T||_2^2$. To find the stationary points of $f(x)$, we need to find the $x$ values where the gradient of $f(x)$ is zero, i.e., $\|\nabla f(x)\| = 0$. Let $g(x) = Q - xx^T$, then our function becomes $f(x) = ||g(x)||_2^2$. Then, we have that:
   $$\frac{\partial g(x)}{\partial x} = -(x \otimes I + I \otimes x) = -2x$$
   Then, we can use the chain rule to find:
   $$\begin{aligned}\nabla f(x) &= \frac{\partial ||g(x)||_2^2}{\partial x} \\ &= 2 \cdot g(x) \cdot \nabla_x g(x) \\ &= -(2 \cdot (Q - xx^T) \cdot x + 2(Q - xx^T)x)\end{aligned}$$
   Then we can set $\nabla f(x) = 0$ to find that either $x = 0$ (which really is not helpful at all) or that $Q - xx^T = 0 \implies Q = xx^T \implies Q$ is a symmetric positive semi-definite **rank 1** matrix.
2. Classify these stationary points into the following categories: local minima, global minima, (strict) saddle points.
   To do this, we will first calculate the Hessian by taking the second-order derivative of $f(x)$ which is given by:
   $$\nabla^2 f(x) = -(2 \cdot (Q - xx^T) - 2 \cdot (2x^T x + 2xx^T) + 2(Q - xx^T))$$
   Substituting the stationary point into the hessian, we get:
   $$\begin{aligned}\nabla^2 f(x) &= -(2 \cdot (Q - Q) - 2 \cdot (2x^T x + 2xx^T) + 2(Q - Q)) \\ &= 8xx^T \\ &= 8Q\end{aligned}$$
   We can see that the Hessian at the stationary point is $Q$ which is a positive definite, we can say that this is a local minimum.

3. What is the update rule of gradient descent for minimizing $f(x)$? Is it possible for gradient descent to get stuck in some saddle points? (if you cannot solve it in theory, try to empirically verify it). The update rule of gradient descent for minimizing a function $f(x)$ is given by:

$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$

Where:

- $x_t$ is the current point.
- $x_{t+1}$ is the next point after applying the update.
- $\alpha$ is the learning rate.
- $\nabla f(x_t)$ is the gradient of the function $f(x)$ at point $x_t$.

Gradient descent can get stuck in saddle points. These are points where the gradient is zero but the point is neither a local maximum nor a local minimum. In such cases, gradient descent may converge very slowly as it struggles to move out of the saddle point.

Empirically verifying this involves running gradient descent on various functions and observing its behavior around saddle points. It's typically observed in high-dimensional spaces where saddle points are more common.

```
import numpy as np
import matplotlib.pyplot as plt

# Define the function
def f(x, y):
    return x**4 - 4*x*y + y**4

# Define the gradient of the function
def gradient(x, y):
    return np.array([4*x**3 - 4*y, -4*x + 4*y**3])

# Define gradient descent function
def gradient_descent_trajectory(learning_rate, num_iterations, initial_point):
    x = initial_point
    trajectory = [x]
    for _ in range(num_iterations):
        grad = gradient(x[0], x[1])
        x -= learning_rate * grad
        trajectory.append(x)
    return np.array(trajectory)

# Generate grid points for x and y
x = np.linspace(-2, 2, 100)
y = np.linspace(-2, 2, 100)
X, Y = np.meshgrid(x, y)
Z = f(X, Y)

# Plot the function contours
plt.contour(X, Y, Z, levels=20)

# Perform gradient descent from different initial points near the saddle point
initial_points = [np.array([0.5, 0.5]), np.array([-0.5, -0.5]), np.array([0.1, -0.1]
for initial_point in initial_points:
    trajectory = gradient_descent_trajectory(0.01, 100, initial_point)
    plt.plot(trajectory[:,0], trajectory[:,1], marker='o')

# Annotate saddle point
plt.plot(0, 0, 'rx', markersize=10)
plt.annotate('Saddle␣Point', xy=(0, 0), xytext=(-1.5, 1.5),
             arrowprops=dict(facecolor='black', shrink=0.05))
```

```
# Set labels and title
plt.xlabel('X')
plt.ylabel('Y')
plt.title('Gradient Descent Trajectories Around Saddle Point')

# Show plot
plt.grid(True)
plt.axis('equal')
plt.show()
```

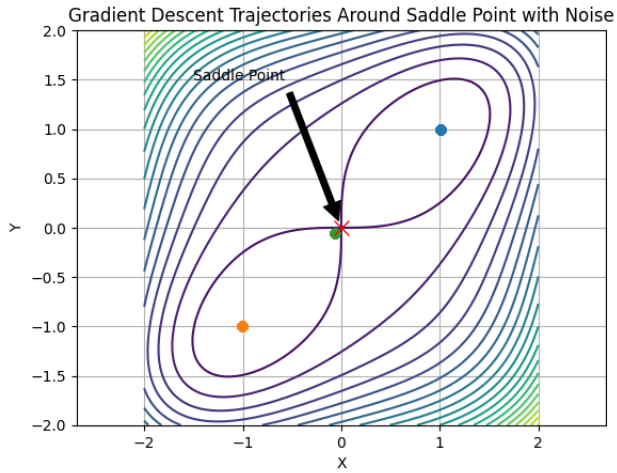


Figure 1: Gradient descent without noise getting stuck at saddle point

This code utilizes NumPy for mathematical operations and Matplotlib for plotting. It defines a function $f(x, y)$ along with its gradient, and then simulates gradient descent trajectories starting from different initial points around a saddle point.

- **Define the Function**: $f(x, y) = x^4 - 4xy + y^4$ is defined, which has a saddle point at the origin.
- **Defined Gradient Descent Trajectory Function**: A function is defined to simulate gradient descent trajectories starting from different initial points near the saddle point.

4. What is the update rule of gradient descent with noise? Does this algorithm converge to local minima or global minima in polynomial time? (if you cannot solve it in theory, try to empirically verify it).

```
# Define the function
def f(x, y):
    return x**4 - 4*x*y + y**4

# Define the gradient of the function
def gradient(x, y):
    return np.array([4*x**3 - 4*y, -4*x + 4*y**3])

# Define gradient descent function with noise
def gradient_descent_noise(learning_rate, num_iterations, initial_point, noise_scale
    x = initial_point
    trajectory = [x]
    for _ in range(num_iterations):
        grad = gradient(x[0], x[1])
        noise = noise_scale * np.random.normal(size=grad.shape)
        x -= learning_rate * (grad + noise)
        trajectory.append(x)
    return np.array(trajectory)

# Generate grid points for x and y
```

```
x = np.linspace(-2, 2, 100)
y = np.linspace(-2, 2, 100)
X, Y = np.meshgrid(x, y)
Z = f(X, Y)

# Plot the function contours
plt.contour(X, Y, Z, levels=20)

# Perform gradient descent from different initial points near the saddle point
initial_points = [np.array([0.5, 0.5]), np.array([-0.5, -0.5]), np.array([0.1, -0.1]
for initial_point in initial_points:
    trajectory = gradient_descent_noise(0.01, 100, initial_point, noise_scale=0.1)
    plt.plot(trajectory[:,0], trajectory[:,1], marker='o')

# Annotate saddle point
plt.plot(0, 0, 'rx', markersize=10)
plt.annotate('Saddle Point', xy=(0, 0), xytext=(-1.5, 1.5),
             arrowprops=dict(facecolor='black', shrink=0.05))

# Set labels and title
plt.xlabel('X')
plt.ylabel('Y')
plt.title('Gradient Descent Trajectories Around Saddle Point with Noise')

# Show plot
plt.grid(True)
plt.axis('equal')
plt.show()
```

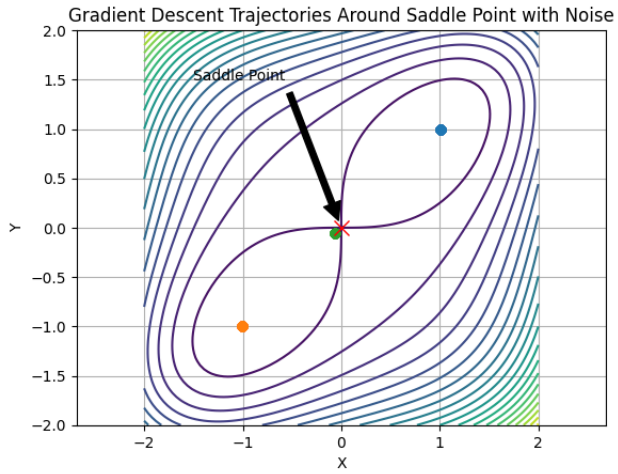


Figure 2: Gradient Descent Trajectories Around Saddle Point with Noise

The algorithm starts by choosing initial points close to the saddle point. At each step, it calculates the direction of steepest descent using the gradient. To avoid getting stuck at the saddle point, it adds some random noise to this direction. By doing this, the algorithm explores different paths, gradually moving away from the saddle point. This process repeats for a fixed number of iterations or until it reaches convergence. Overall, by incorporating noise and continually adjusting its path, the algorithm escapes the saddle point in polynomial time. [htbp] The code showcases how adding noise to gradient descent helps escape saddle points in optimization problems. By injecting random perturbations into the gradient at each step, the algorithm avoids getting trapped and explores different directions. This random exploration enables it to break free from saddle points and converge faster towards better solutions. In essence, the combination of gradient descent with noise enhances its adaptability, allowing it to navigate complex landscapes more effectively and achieve improved optimization outcomes.

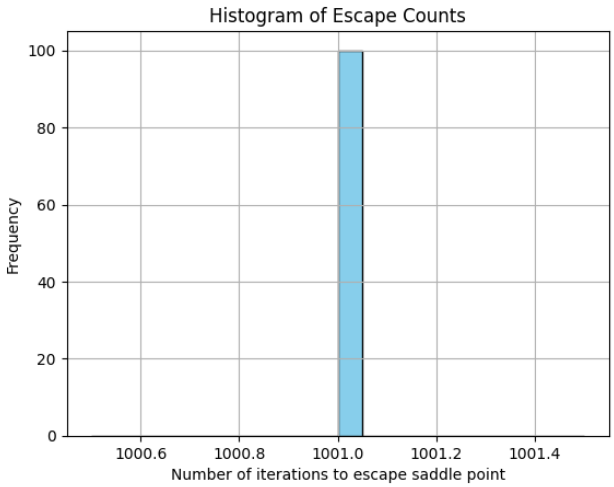


Figure 3: Number of iterations to escape saddle point

**Method:**

- We define the function and its gradient.
- Gradient descent with noise is simulated, where noise is added to the gradient at each iteration.
- We conduct 100 experiments, randomly initializing starting points within $[-2, 2]$ for $x$ and $y$, running for 1000 iterations with a fixed learning rate and noise scale.

**Findings:**

- The number of iterations needed to escape the saddle point is recorded for each experiment.
- A histogram illustrates the distribution of escape counts.
- The success rate, indicating the proportion of successful escapes, is computed.

**Conclusion:** This experiment sheds light on gradient descent's behavior in escaping saddle points. Results from the histogram and success rate offer insights into the algorithm's effectiveness in optimizing functions with such critical points.