Task 1 : Ascending the Gradient Descent

Use the below dataset for Task 1:

```
np.random.seed(45)
num_samples = 40

# Generate data
x1 = np.random.uniform(-1, 1, num_samples)
f_x = 3*x1 + 4
eps = np.random.randn(num_samples)
y = f_x + eps
```

1. Use `torch.autograd` to find the true gradient on the above dataset using linear regression (in the form $\theta_1 x + \theta_0$) for any given values of (θ_0, θ_1).
2. Using the same (θ_0, θ_1) as above, calculate the stochastic gradient for all points in the dataset. Then, find the average of all those gradients and show that the stochastic gradient is a good estimate of the true gradient.
3. Implement full-batch, mini-batch, and stochastic gradient descent. Calculate the average number of iterations required for each method to get sufficiently close to the optimal solution, where "sufficiently close" means within a distance of ϵ (or ϵ-neighborhood) from the minimum value of the loss function. Visualize the convergence process for 15 epochs. Choose $\epsilon = 0.001$ for convergence criteria.
   Which optimization process takes a larger number of epochs to converge, and why?
   Show the contour plots for different epochs (or show an animation/GIF) for visualization of the optimization process. Also, make a plot for Loss vs. Epochs for all the methods.
4. Explore the article here on gradient descent with momentum. Implement gradient descent with momentum for the dataset. Visualize the convergence process for 15 steps. Compare the average number of steps taken with gradient descent (for variants full-batch and stochastic) with momentum to that of vanilla gradient descent to converge to an ϵ-neighborhood for both datasets. Choose $\epsilon = 0.001$.
   Write down your observations. Show the contour plots for different epochs for momentum implementation. Specifically, show all the vectors: gradient, current value of θ, momentum, etc.

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

1. Using autograd to find the true gradient on the above dataset using linear regression for any θ.

```
# Data
np.random.seed(45)
x_vals = np.random.uniform(-1, 1, 40)
y_vals = 3 * x_vals + 4 + np.random.randn(40)

# Convert to tensors
x = torch.tensor(x_vals, dtype=torch.float64).view(-1, 1)
y = torch.tensor(y_vals, dtype=torch.float64).view(-1, 1)

# Design matrix with bias
ones_col = torch.ones_like(x)
X = torch.cat((ones_col, x), dim=1)  # shape (40, 2)

# Initialize θ = [1.0, 2.0] for autograd
theta = torch.tensor([1.0, 2.0], dtype=torch.float64, requires_grad=True)

# Predictions and Loss
y_hat = X @ theta.view(-1, 1)
loss = torch.mean((y_hat - y) ** 2)

# Gradient via autograd
loss.backward()
print("Initial θ:", theta.detach().numpy())
print("Gradient at θ = [1, 2]:", theta.grad.numpy())
print("Loss at θ = [1, 2]:", loss.item())
```

```
Initial θ: [1. 2.]
Gradient at θ = [1, 2]: [-5.78581563  0.06868769]
Loss at θ = [1, 2]: 9.10843716496166
```

```
# Compute optimal θ using normal equation
X_detached = X.detach()
y_detached = y.detach()
theta_opt = torch.inverse(X_detached.T @ X_detached) @ X_detached.T @ y_detached

# Calculate loss at optimal θ
y_pred_opt = X_detached @ theta_opt
```

```
optimal_loss = torch.mean((y_pred_opt - y_detached) ** 2).item()

print("Optimal θ (closed-form):", theta_opt.view(-1).numpy())
print("Minimum possible Loss (MSE):", optimal_loss)
```

```
Optimal θ (closed-form): [3.9507064  2.68246893]
Minimum possible Loss (MSE): 0.5957541565733318
```

2. Using the same (θ_0, θ_1) as above, and calculate the stochastic gradient for all points in the dataset. Then, find the average of all those gradients and show that the stochastic gradient is a good estimate of the true gradient.

```
# Get the true gradient
def get_true_grad(X, y, theta):
    if theta.grad is not None:
        theta.grad.zero_()

    preds = X @ theta.view(-1, 1)
    loss = torch.mean((preds - y) ** 2)
    loss.backward()
    return theta.grad.clone(), loss.item()

true_grad, true_loss = get_true_grad(X, y, theta)

# Function to compute gradient on a single point (stochastic)
def grad_on_one_point(x_single, y_single):
    if theta.grad is not None:
        theta.grad.zero_()

    pred = x_single @ theta.view(-1, 1)
    loss = (pred - y_single) ** 2  # no mean, since it's one point
    loss.backward()
    return theta.grad.clone()

# Loop over each point and collect gradients
sgd_list = []

for i in range(len(y)):
    xi = X[i].view(1, -1).detach()
    yi = y[i].view(1, -1).detach()
    grad_i = grad_on_one_point(xi, yi)
    sgd_list.append(grad_i)

# Average all gradients
avg_sgd = torch.stack(sgd_list).mean(dim=0)

# Compare with true gradient
residual = true_grad - avg_sgd
```

```
# Final outputs
print("True Gradient:", true_grad.numpy())
print("Avg. Stochastic Gradient:", avg_sgd.numpy())
print("Difference (Residual):", residual.numpy())
```

```
True Gradient: [-5.78581563  0.06868769]
Avg. Stochastic Gradient: [-5.78581563  0.06868769]
Difference (Residual): [ 8.88178420e-16 -1.38777878e-16]
```

We calculated the gradient using each individual data point (stochastic) and then averaged them. The resulting average matched the gradient computed using the entire dataset. The residual difference was extremely small (~1e-16), which confirms that the stochastic gradient is a reliable approximation of the full gradient.

3,4. Implementing full-batch, mini-batch, and stochastic gradient descent to minimize a loss function, computing average steps to reach an ϵ-neighborhood (ϵ = 0.001), and visualizing convergence over 15 epochs using loss plots. Extending to gradient descent with momentum (full-batch and stochastic), plotting contour plots across epochs, comparing with vanilla GD, and noting observations.

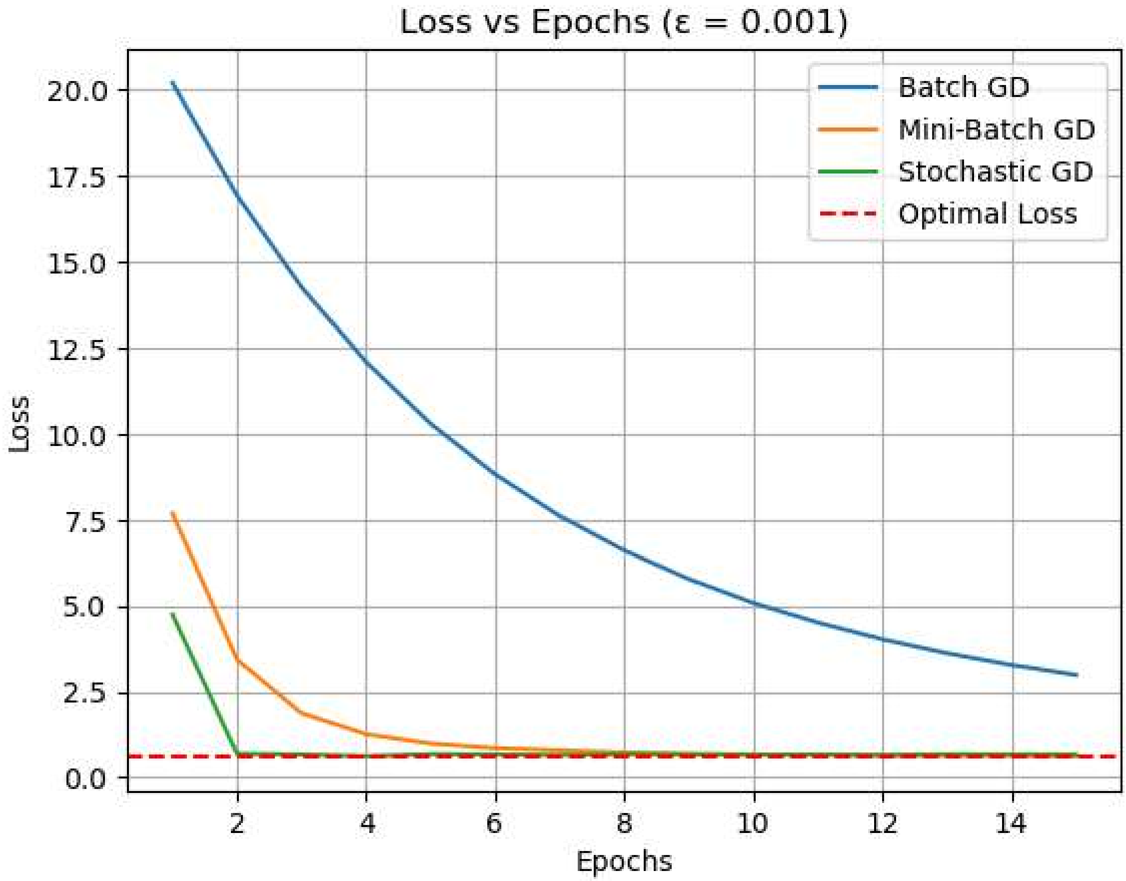
```
# Imports and Data Setup

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

np.random.seed(45)
num_samples = 40
x_vals = np.random.uniform(-1, 1, num_samples)
y_vals = 3 * x_vals + 4 + np.random.randn(num_samples)

x = torch.tensor(x_vals, dtype=torch.float64).view(-1, 1)
y = torch.tensor(y_vals, dtype=torch.float64).view(-1, 1)
X = torch.cat([torch.ones_like(x), x], dim=1)

# Compute Optimal Theta

theta_opt = torch.linalg.inv(X.T @ X) @ X.T @ y
loss_opt = torch.mean((X @ theta_opt - y) ** 2).item()



# GRadient Descent Function
```

```
def gradient_descent(method='batch', lr=0.05, eps=1e-3, max_epochs=2000, batch_size=8, momer
    theta = torch.randn(2, dtype=torch.float64, requires_grad=True)
    prev_update = torch.zeros_like(theta)

    losses = []
    thetas = []
    grads = []
    momentums = []

    iter_count = 0  # Track number of iterations (parameter updates)

    for epoch in range(max_epochs):
        if method == 'batch':
            batches = [(X, y)]
        elif method == 'mini':
            idx = torch.randperm(num_samples)
            X_shuff = X[idx].detach()
            y_shuff = y[idx].detach()
            batches = [(X_shuff[i:i+batch_size], y_shuff[i:i+batch_size]) for i in range(0,
        elif method == 'stochastic':
            idx = torch.randperm(num_samples)
            X_shuff = X[idx].detach()
            y_shuff = y[idx].detach()
            batches = [(X_shuff[i:i+1], y_shuff[i:i+1]) for i in range(num_samples)]
        else:
            raise ValueError("Invalid method")

        total_loss = 0
        for xb, yb in batches:
            iter_count += 1  #  Count iteration per batch
            theta = theta.detach().requires_grad_(True)
            pred = xb @ theta.view(-1, 1)
            loss = torch.mean((pred - yb)**2)
            loss.backward()

            with torch.no_grad():
                grad = theta.grad
                if momentum:
                    update = lr * grad + momentum * prev_update
                    prev_update = update
                else:
                    update = lr * grad
                theta -= update

            grads.append(grad.clone().detach())
            momentums.append(prev_update.clone().detach())
            theta. grad.zero_()
            total_loss += loss.item()

        avg_loss = total_loss / len(batches)
        losses.append(avg_loss)
```

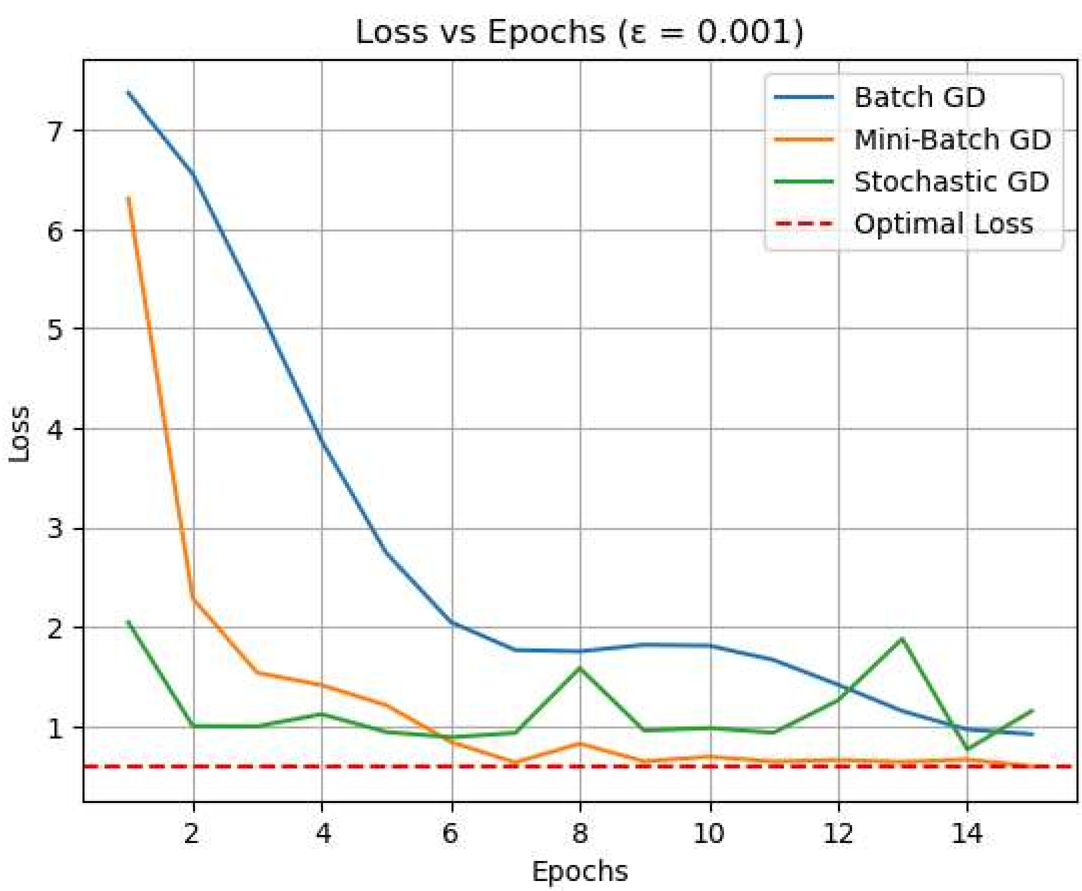
```
        thetas.append(theta.detach().clone())

        if torch.norm(theta - theta_opt.squeeze()) <= eps:
            break

    return losses, thetas, grads, momentums, epoch + 1, iter_count


# Plot Loss vs Epochs

def plot_losses(momentum=None, epoch_cap=None):
    methods = {'batch': 'Batch GD', 'mini': 'Mini-Batch GD', 'stochastic': 'Stochastic GD'}
    results = {}

    for method in methods:
        losses, _, _, _, epochs_run, iterations = gradient_descent(method=method, momentum=m
        results[method] = (epochs_run, iterations)
        if epoch_cap is None:
            x_vals = range(1, len(losses)+1)
            y_vals = losses
        else:
            x_vals = range(1, min(epoch_cap, len(losses)) + 1)
            y_vals = losses[:min(epoch_cap, len(losses))]

        plt.plot(x_vals, y_vals, label=methods[method])

    plt.axhline(y=loss_opt, linestyle='--', color='red', label='Optimal Loss')
    plt.xlabel("Epochs")
    plt.ylabel("Loss")
    plt.title("Loss vs Epochs (ε = 0.001)")
    plt.grid(True)
    plt.legend()
    plt.show()

    print("Epochs & Iterations until convergence (ε = 0.001):")
    for m in methods:
        e, it = results[m]
        print(f"{methods[m]}: {e} epochs, {it} iterations")
```

```
# Average Epochs Over Trials

def average_epochs_and_iterations(method, trials=10, momentum=None):
    epoch_counts = []
    iter_counts = []
    for seed in range(trials):
        torch.manual_seed(seed)
        _, _, _, _, epochs, iterations = gradient_descent(method=method, momentum=momentum)
        epoch_counts.append(epochs)
        iter_counts.append(iterations)
```

```
        return np.mean(epoch_counts), np.mean(iter_counts)



    # Compare Momentum vs No Momentum

    def compare_momentum():
        print("\nAverage over 10 trials WITHOUT momentum:")
        for method in ['batch', 'mini', 'stochastic']:
            avg_ep, avg_it = average_epochs_and_iterations(method, momentum=None)
            print(f"{method.title():<10}: {avg_ep:.2f} epochs, {avg_it:.2f} iterations")

        print("\nAverage over 10 trials WITH momentum:")
        for method in ['batch', 'mini', 'stochastic']:
            avg_ep, avg_it = average_epochs_and_iterations(method, momentum=0.9)
            print(f"{method.title()} + Momentum: {avg_ep:.2f} epochs, {avg_it:.2f} iterations")


    plot_losses(momentum=None, epoch_cap=15)
    plot_losses(momentum=0.9, epoch_cap=15)
```




```
Epochs & Iterations until convergence (ε = 0.001):
Batch GD: 264 epochs, 264 iterations
Mini-Batch GD: 83 epochs, 415 iterations
Stochastic GD: 2000 epochs, 80000 iterations
```



```
Epochs & Iterations until convergence (ε = 0.001):
Batch GD: 102 epochs, 102 iterations
Mini-Batch GD: 1624 epochs, 8120 iterations
Stochastic GD: 2000 epochs, 80000 iterations
```

```
compare_momentum()
```

```
Average over 10 trials WITHOUT momentum:
Batch     : 258.50 epochs, 258.50 iterations
Mini      : 94.00 epochs, 470.00 iterations
Stochastic: 1798.10 epochs, 71924.00 iterations

Average over 10 trials WITH momentum:
Batch + Momentum: 102.00 epochs, 102.00 iterations
Mini + Momentum: 1725.40 epochs, 8627.00 iterations
Stochastic + Momentum: 1843.00 epochs, 73720.00 iterations
```

Observations

1. **Stochastic Gradient Descent (SGD)** took the **most epochs and iterations** to converge — around **1798 epochs (without momentum)** and **1843 epochs (with momentum)** on average. This is because it updates parameters using only one data point at a time, which causes a lot of fluctuation (noise) in the gradient direction, making it slower to settle near the minimum.

2. **Batch Gradient Descent** was the **most stable and efficient** method. It used the full dataset for each update, resulting in smooth convergence.
   When momentum was added, the performance improved a lot, average epochs reduced from **258.5 (vanilla)** to just **102 (with momentum)**.

3. **Mini-Batch Gradient Descent** performed well **without momentum**, requiring only **94 epochs** on average.
   But with momentum, its performance **got worse**, taking **1725.4 epochs** to converge.
   This is likely because the randomness in mini-batches, combined with momentum, may have led to overshooting or instability.

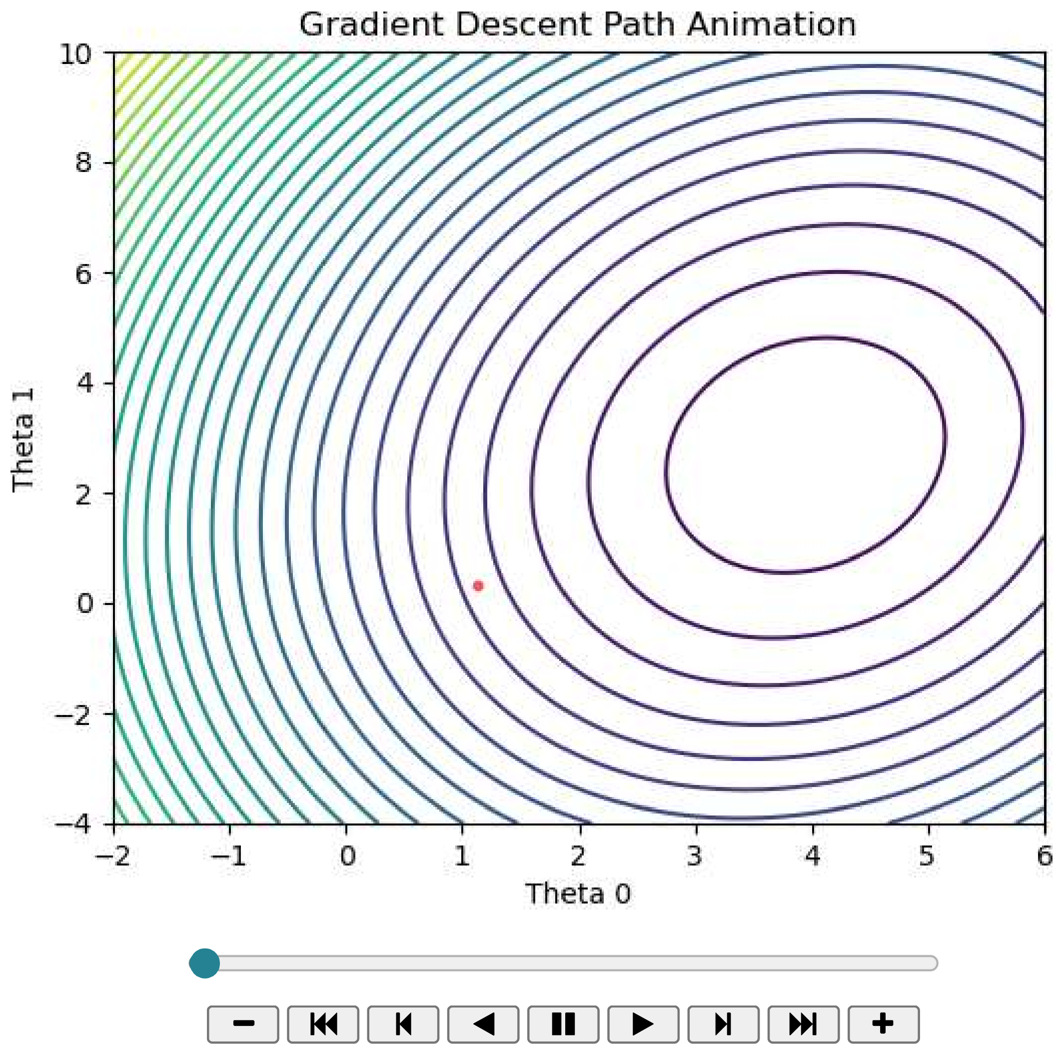
4. In general, **momentum is useful when updates are consistent** (like in batch GD), but **less helpful or even harmful** when updates are noisy (like in mini/stochastic GD).
   Momentum works best when the directionof updatess doesn't vary too much; otherwise, it can add to the instability instead of speeding things up.

```
from matplotlib import animation
from IPython.display import HTML

def generate_contour_animation(thetas, gif_name='gd_animation.gif'):
    w0 = np.linspace(-2, 6, 100)
```

```
    w1 = np.linspace(-4, 10, 100)
    W0, W1 = np.meshgrid(w0, w1)
    Z = np.zeros_like(W0)

    for i in range(W0.shape[0]):
        for j in range(W0.shape[1]):
            w = torch.tensor([W0[i, j], W1[i, j]], dtype=torch.float64)
            y_pred = X @ w.view(-1, 1)
            Z[i, j] = torch.mean((y - y_pred)**2).item()

    fig, ax = plt.subplots(figsize=(6, 5))
    ax.contour(W0, W1, Z, levels=30, cmap='viridis')
    ax.set_xlabel("Theta 0")
    ax.set_ylabel("Theta 1")
    ax.set_title("Gradient Descent Path Animation")

    # Thin red line with small transparent markers
    path, = ax.plot([], [], color='red', linewidth=1, marker='o', markersize=3, alpha=0.6)

    theta_np = np.array([[t[0].item(), t[1].item()] for t in thetas])

    def init():
        path.set_data([], [])
        return path,

    def animate(i):
        path.set_data(theta_np[:i+1, 0], theta_np[:i+1, 1])
        return path,

    anim = animation.FuncAnimation(
        fig, animate, init_func=init, frames=len(thetas),
        interval=300, blit=True
    )

    anim.save(gif_name, writer='pillow')
    plt.close()
    print(f"Saved animation to {gif_name}")
    return HTML(anim.to_jshtml())
```

```
_, thetas, _, _, _,_ = gradient_descent(method='batch', momentum=0.9)
generate_contour_animation(thetas, gif_name="batch_mmntm_gd.gif")
```

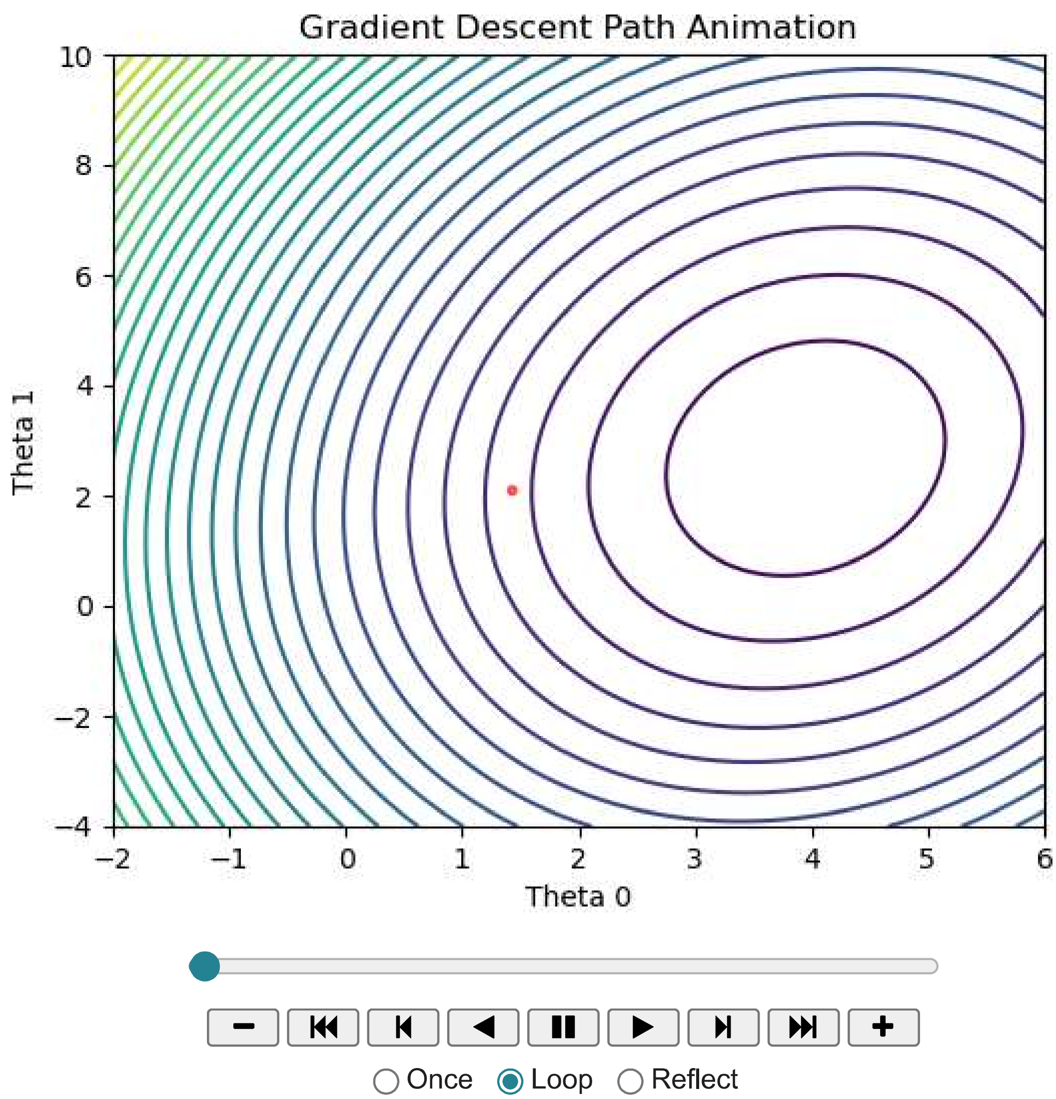
```
Saved animation to batch_mmntm_gd.gif
Animation size has reached 21107392 bytes, exceeding the limit of 20971520.0. If you're
```



Once Loop Reflect

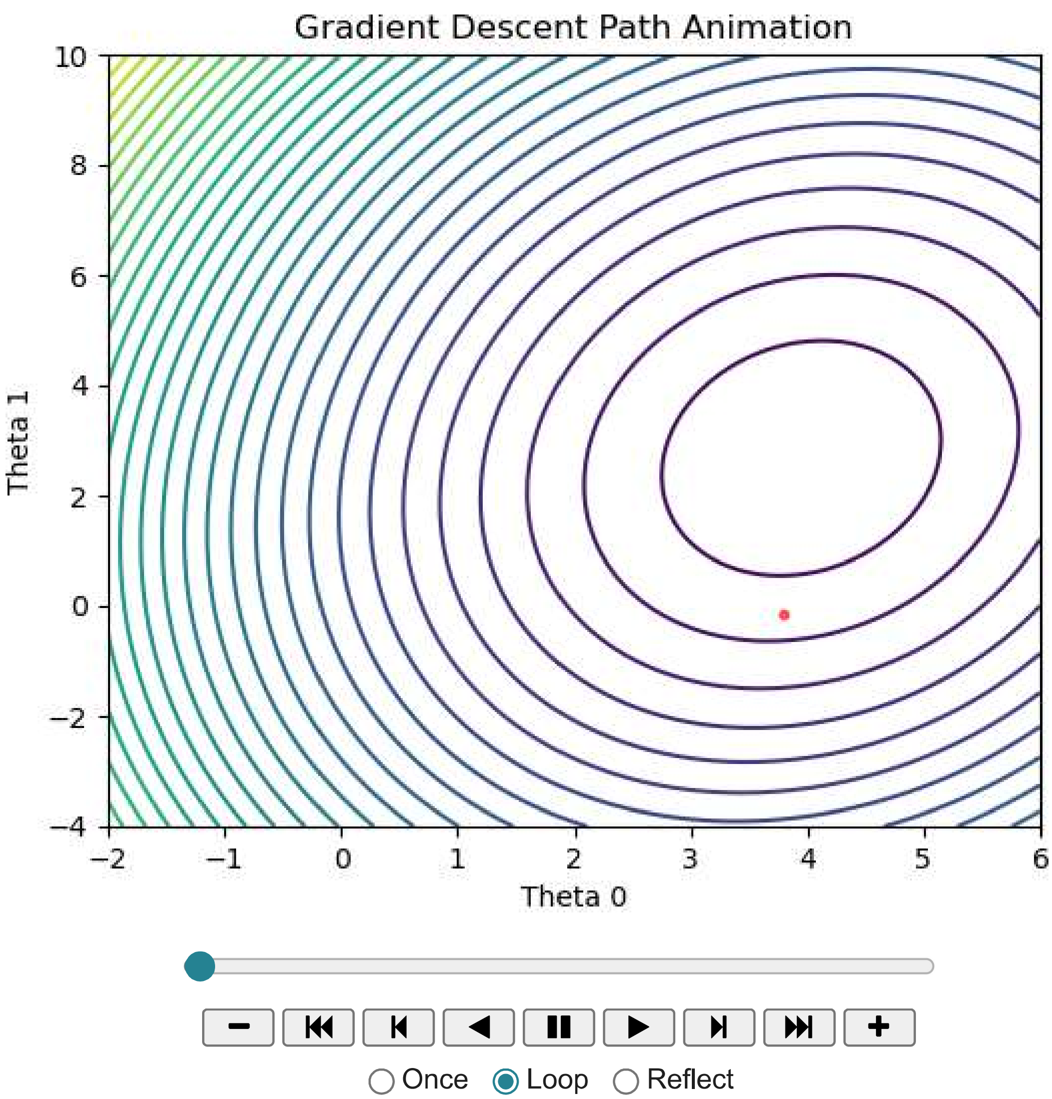
```
_, thetas, _, _, _,_ = gradient_descent(method='batch', momentum=None)
generate_contour_animation(thetas, gif_name="batch_gd.gif")
```

```
Saved animation to batch_gd.gif
Animation size has reached 21060344 bytes, exceeding the limit of 20971520.0. If you're
```



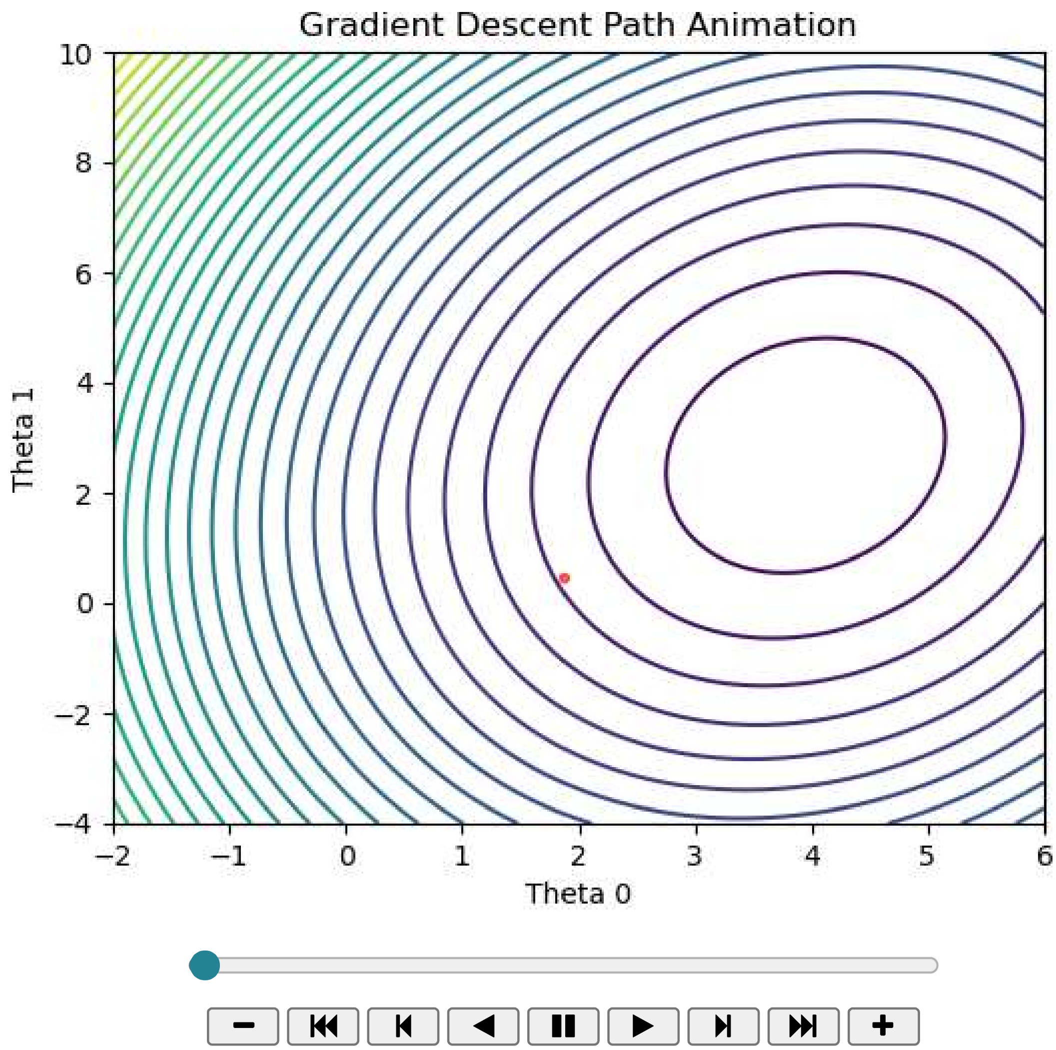

```
_, thetas, _, _, _,_ = gradient_descent(method='mini', momentum=0.9)
generate_contour_animation(thetas, gif_name="minibatch_mmntm_gd.gif")
```

```
Saved animation to minibatch_mmntm_gd.gif
Animation size has reached 21092148 bytes, exceeding the limit of 20971520.0. If you're
```




```
_, thetas, _, _, _, _ = gradient_descent(method='mini', momentum=None)
generate_contour_animation(thetas, gif_name="minibatch_gd.gif")
```

Saved animation to minibatch_gd.gif



Once Loop Reflect

```
_, thetas, _, _, _,_ = gradient_descent(method='stochastic', momentum=0.9)
generate_contour_animation(thetas, gif_name="stochastic_mmntm_gd.gif")
```

```
Saved animation to stochastic_mmntm_gd.gif
Animation size has reached 21002515 bytes, exceeding the limit of 20971520.0. If you're
```