Домашнее задание №11

Импортируем библиотеки

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from typing import Tuple

sns.set_palette(palette="Reds")
```

Функция вычисления градиента

In [ ]:
```
def compute_gradients(
        X: np.ndarray,
        y: np.ndarray,
        weights: np.ndarray,
) -> Tuple[np.ndarray, np.ndarray]:
    """Функция для вычисления градиента

    Args:
        - X (np.ndarray): Матрица объектов-признаков
        - y (np.ndarray): Вектор таргетов
        - weights (np.ndarray): Значение для весов.

    Returns:
        - Tuple[np.ndarray, np.ndarray]: Возвращает значение градиента и вектор с ошибка
    """
    n = X.shape[0]
    y_pred = X.dot(weights)  # Ответы с текущими весами
    err = y_pred - y  # Ошибка
    grad = 2 * X.T.dot(err) / n # Градиент
    return grad, err
```

1. Momentum

In [ ]:
```
def momentum_optimizer(
        X: np.ndarray,
        y: np.ndarray,
        w = None,
        learning_rate=0.01,
        momentum=0.9,
        epochs=100
) -> Tuple[np.ndarray, np.ndarray, np.ndarray]:
    """Функция для оптимизации весов с помощью метода инерции (Momentum)

    Args:
        - X (np.ndarray): Матрица объектов-признаков
        - y (np.ndarray): Вектор таргетов
        - w (_type_, optional): Начальное значение для весов. Defaults to None.
        - learning_rate (float): Длина шага. Defaults to 0.01.
        - momentum (float):  Скорость изменения. Defaults to 0.9.
        - epochs (int): количество эпох. Defaults to 100.

    Returns:
        - Tuple[np.ndarray, np.ndarray, np.ndarray]: Возвращает полученные веса, вектор
    """
    _, m = X.shape
```

```
    if w is None:
        w = np.random.standard_normal(m)
    weights = w.copy()
    w_history = [weights]
    err_history = []

    velocity = np.zeros(m)


    for _ in range(epochs):
        gradients, errors = compute_gradients(X, y, weights)  # Замените на свою функцию

        velocity = momentum * velocity + learning_rate * gradients

        weights = weights - velocity

        w_history.append(weights.copy())
        err_history.append(errors.mean())

    return weights,  np.array(w_history), np.array(err_history)
```

2. RMSProp

In [ ]:

```
def rmsprop_optimizer(
        X: np.ndarray,
        y: np.ndarray,
        w=None,
        learning_rate=0.05,
        decay_rate=0.9,
        epochs=100,
        epsilon=1e-8
) -> Tuple[np.ndarray, np.ndarray, np.ndarray]:
    """Функция для оптимизации весов с помощью RMSprop

    Args:
        - X (np.ndarray): Матрица объектов-признаков
        - y (np.ndarray): Вектор таргетов
        - w (_type_, optional): Начальное значение для весов. Defaults to None.
        - learning_rate (float): Длина шага. Defaults to 0.05.
        - decay_rate (float):  Коэффициент сглаживания. Defaults to 0.9.
        - epochs (int): количество эпох. Defaults to 100.

    Returns:
        - Tuple[np.ndarray, np.ndarray, np.ndarray]: Возвращает полученные веса, вектор
    """
    _, m = X.shape

    if w is None:
        w = np.random.standard_normal(m)
    weights = w.copy()

    w_history = [weights]
    err_history = []

    cache = np.zeros(m)

    for _ in range(epochs):
        gradients, errors = compute_gradients(X, y, weights)  # Замените на свою функцию
        cache = decay_rate * cache + (1 - decay_rate) * gradients**2
        weights = weights - learning_rate * gradients / (np.sqrt(cache) + epsilon)
        w_history.append(weights.copy())
        err_history.append(errors.mean())

    return weights,  np.array(w_history), np.array(err_history)
```

3. ADAM

In [ ]:
```
def adam_optimizer(X, y, w=None, learning_rate=0.05, beta1=0.9, beta2=0.999, epsilon=1e-
    """Функция для оптимизации весов с помощью метода ADAM

    Args:
        - X (np.ndarray): Матрица объектов-признаков
        - y (np.ndarray): Вектор таргетов
        - w (_type_, optional): Начальное значение для весов. Defaults to None.
        - learning_rate (float): Длина шага. Defaults to 0.05.
        - beta1 (float):  Коэффициент сглаживания 1. Defaults to 0.9.
        - beta2 (float):  Коэффициент сглаживания 2. Defaults to 0.999.
        - epochs (int): количество эпох. Defaults to 100.

    Returns:
        - Tuple[np.ndarray, np.ndarray, np.ndarray]: Возвращает полученные веса, вектор
    """
    n, m = X.shape[0], X.shape[1]

    if w is None:
        w = np.random.standard_normal(m)
    weights = w.copy()
    w_history = [weights]
    err_history = []


    m_t = np.zeros(m)
    v_t = np.zeros(m)
    t = 0

    for epoch in range(epochs):
        t += 1
        gradients, errors = compute_gradients(X, y, weights)  # Замените на свою функцию
        m_t = beta1 * m_t + (1 - beta1) * gradients
        v_t = beta2 * v_t + (1 - beta2) * gradients**2
        m_hat = m_t / (1 - beta1**t)
        v_hat = v_t / (1 - beta2**t)
        weights = weights - learning_rate * m_hat / (np.sqrt(v_hat) + epsilon)
        w_history.append(weights.copy())
        err_history.append(errors.mean())

    return weights,  np.array(w_history), np.array(err_history)
```

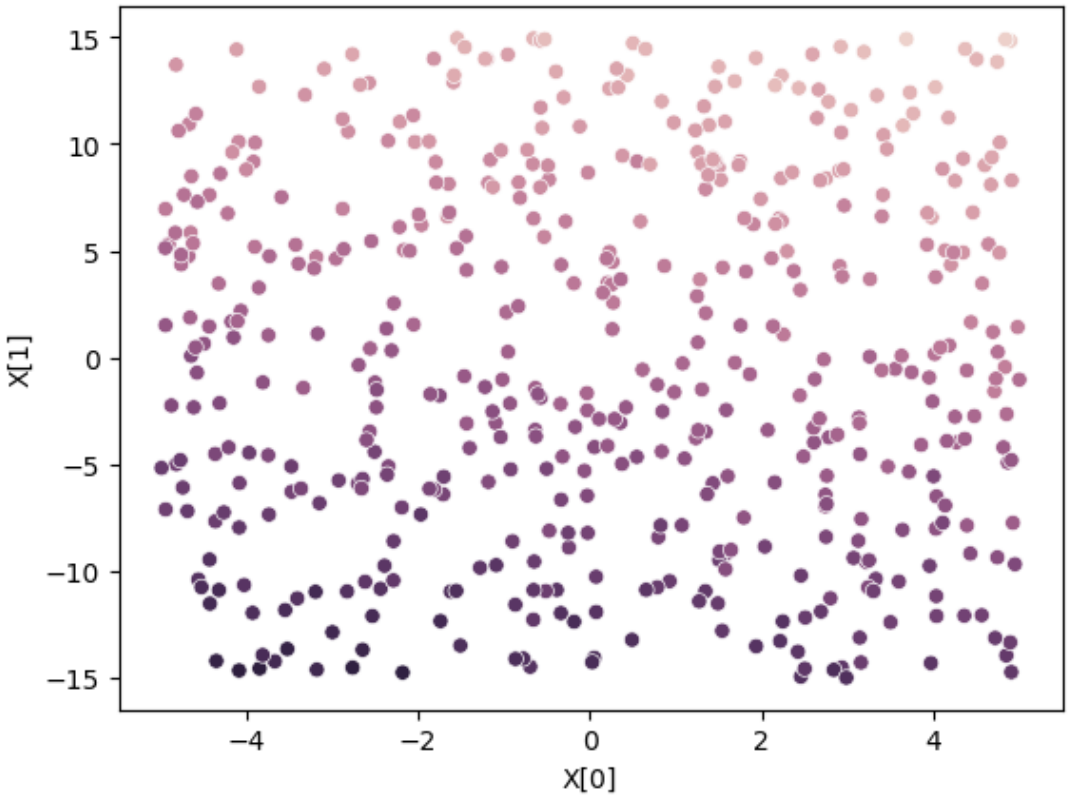
Сгенерируем данные

Для начала сгенериуем выборку из 500 объектов и проверим на ней работу методов

In [ ]:
```
n, m = 500, 2

w_true = np.random.standard_normal(m)

X = np.random.uniform(-5, 5, (n, m))
X *= (np.arange(m) * 2 + 1)[np.newaxis, :]

y = X.dot(w_true) + np.random.normal(0, 1, (n))
```

In [ ]:
```
g = sns.scatterplot(x=X[:, 0], y=X[:, 1], hue=y)
g.legend_.remove()
g.set(xlabel="X[0]", ylabel="X[1]")
```

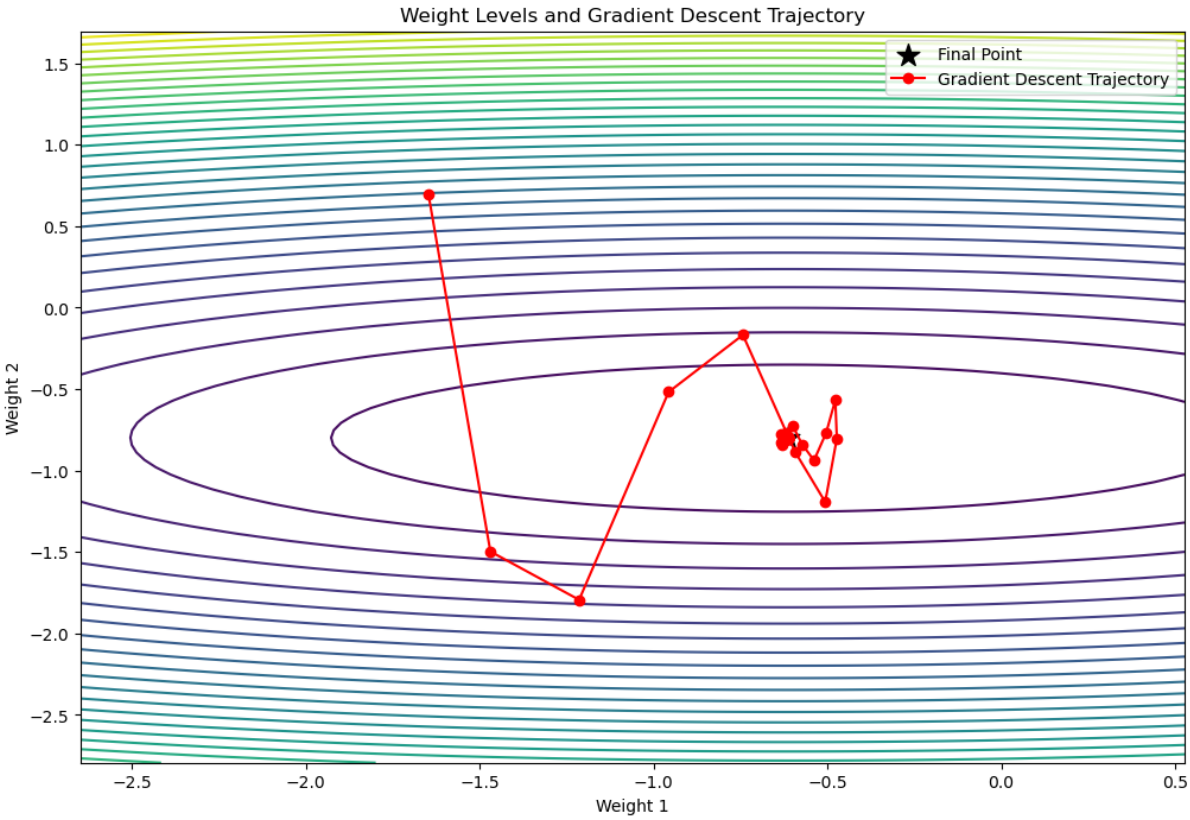
Out[ ]:
```
[Text(0.5, 0, 'X[0]'), Text(0, 0.5, 'X[1]')]
```



In [ ]:
```
def plot_weight_levels(X, y, w_history: np.ndarray):
    w1_vals = np.linspace(min(w_history[:, 0]) - 1, max(w_history[:, 0]) + 1, 100)
    w2_vals = np.linspace(min(w_history[:, 1]) - 1, max(w_history[:, 1]) + 1, 100)

    W1, W2 = np.meshgrid(w1_vals, w2_vals)
    J_vals = np.zeros_like(W1)

    for i in range(len(w1_vals)):
        for j in range(len(w2_vals)):
            w_tmp = np.array([W1[i, j], W2[i, j]])
            J_vals[i, j] = np.mean((X.dot(w_tmp) - y) ** 2) / 2

    plt.figure(figsize=(12, 8))
    plt.contour(W1, W2, J_vals, levels=30, cmap='viridis')

    # w_history = w_history[w_history[:, 0].argsort()[::-1]]
    print(w_history[-1])
    plt.scatter(w_history[-1][0], w_history[-1][1], marker='*', s=200, color='black', la

    plt.plot(w_history[:, 0], w_history[:, 1], marker='o', linestyle='-', color='red', l

    plt.title('Weight Levels and Gradient Descent Trajectory')
    plt.xlabel('Weight 1')
    plt.ylabel('Weight 2')
    plt.legend()
    plt.show()
```

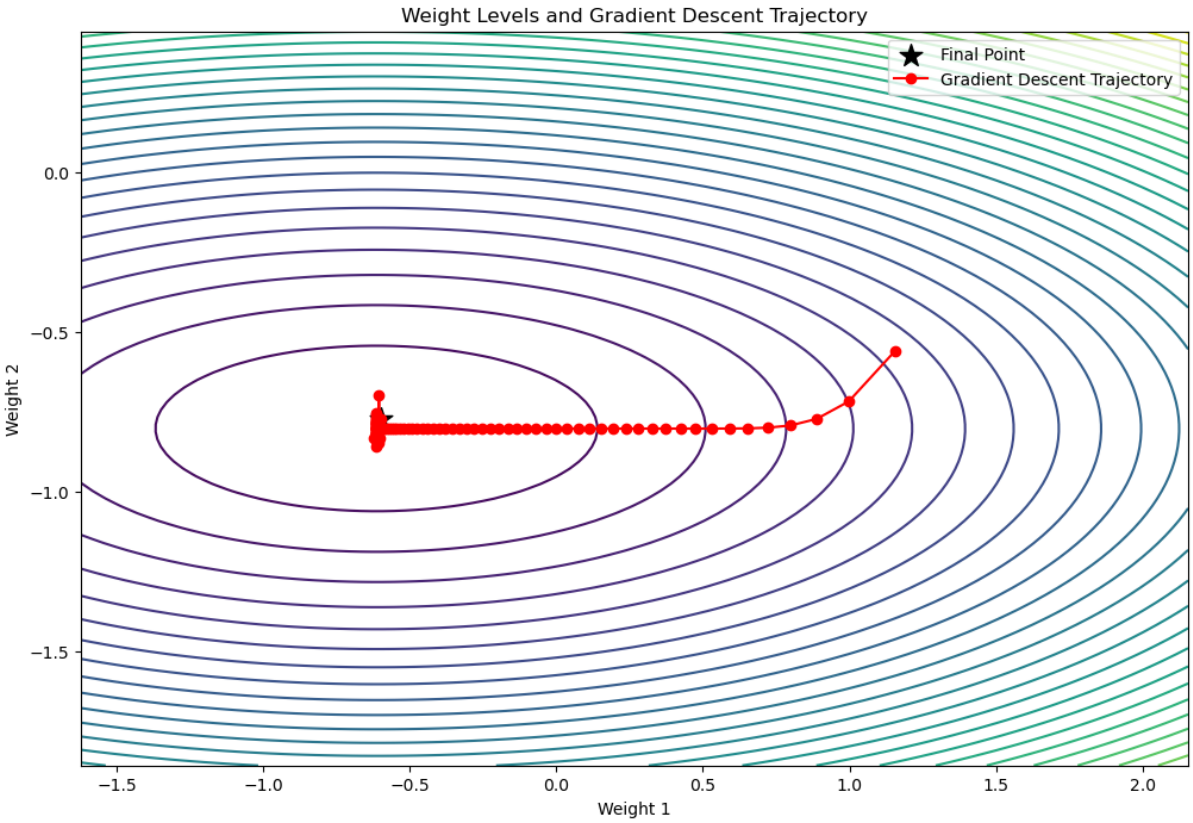
In [ ]:
```
w, w_history, momentum_errors = momentum_optimizer(X=X, y=y, momentum=0.6, epochs=100)
print(w_history[-1])
print()
plot_weight_levels(X, y, w_history)
```

```
[-0.61401856 -0.80114647]

[-0.61401856 -0.80114647]
```



In [ ]:
```
w, w_history, rmsprop_errors = rmsprop_optimizer(X=X, y=y, epochs=100)
print(w_history[-1])
print()
plot_weight_levels(X, y, w_history)
```

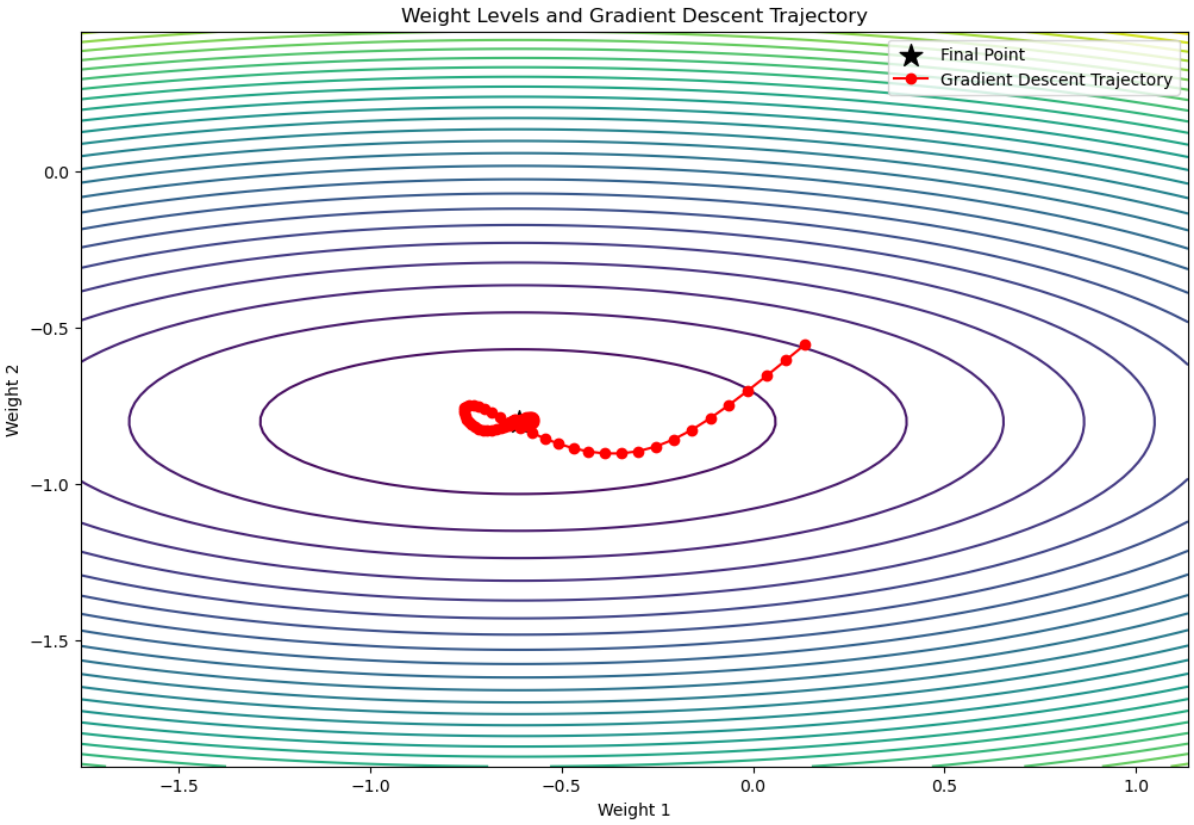
```
[-0.59814493 -0.77006269]

[-0.59814493 -0.77006269]
```



In [ ]:
```
w, w_history, adam_errors = adam_optimizer(X=X, y=y, epochs=100)
print(w_history[-1])
```

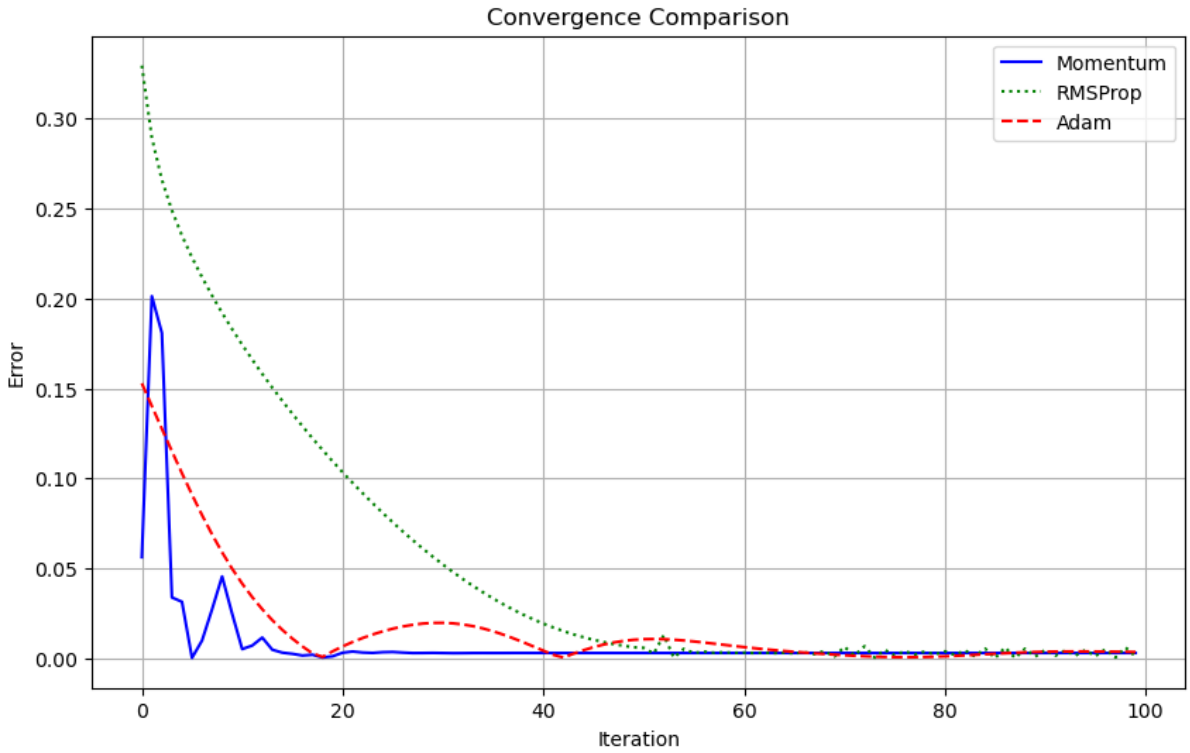
```
print()
plot_weight_levels(X, y, w_history)
```

```
[-0.61071458 -0.80223365]

[-0.61071458 -0.80223365]
```



In [ ]:
```
def plot_convergence(momentum_errors, rmsprop_errors, adam_errors):
    plt.figure(figsize=(10, 6))

    plt.plot(momentum_errors, label='Momentum', color='blue')

    plt.plot(rmsprop_errors, label='RMSProp', color='green', linestyle='dotted')


    plt.plot(adam_errors, label='Adam', color='red', linestyle='--')

    plt.title('Convergence Comparison')
    plt.xlabel('Iteration')
    plt.ylabel('Error')
    plt.legend()
    plt.grid(True)
    plt.show()
```

In [ ]:
```
plot_convergence(abs(momentum_errors), abs(rmsprop_errors), abs(adam_errors))
```



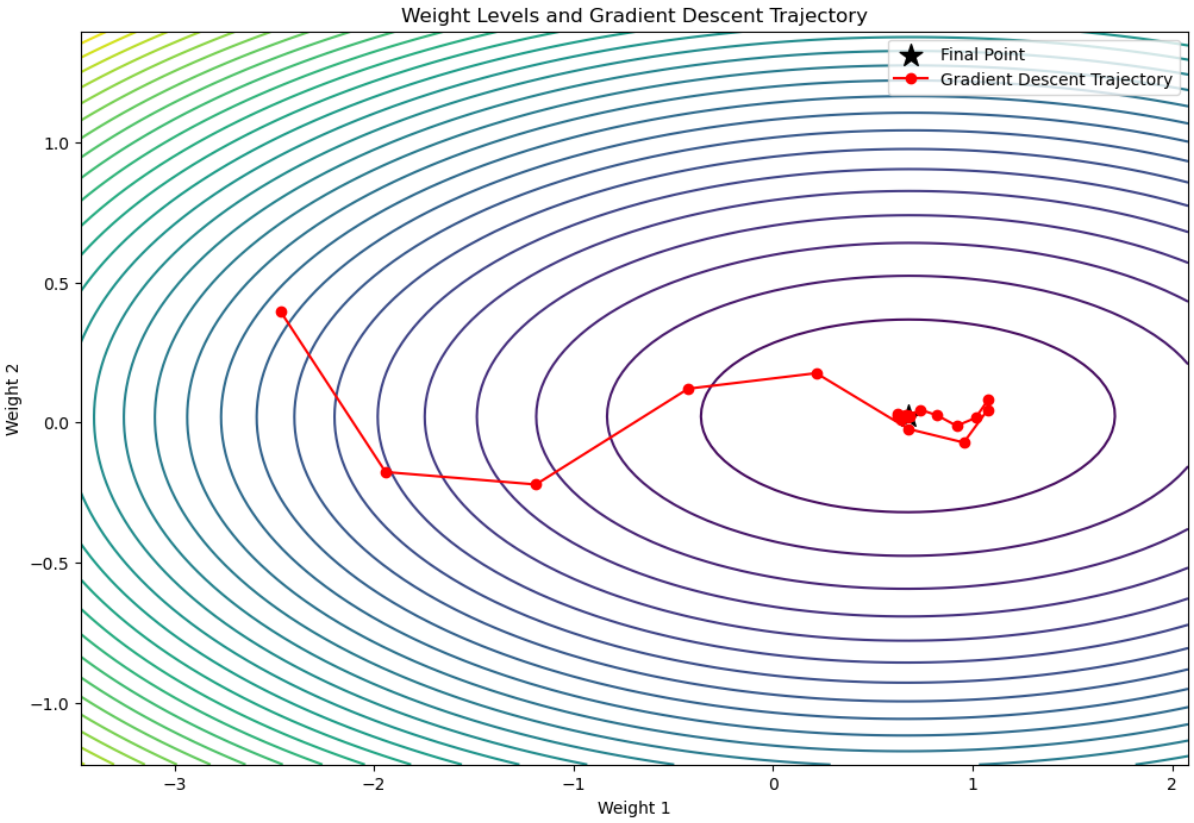
Как видно, Momentum сошелся довольно быстро. Дольше всех сходился RMSProp.

Теперь сгенерируем выборку размером 10000 и проверим работу методов на ней.

In [ ]:
```
n, m = 100000, 2

w_true = np.random.standard_normal(m)

X = np.random.uniform(-5, 5, (n, m))
X *= (np.arange(m) * 2 + 1)[np.newaxis, :]

y = X.dot(w_true) + np.random.normal(0, 1, (n))
```

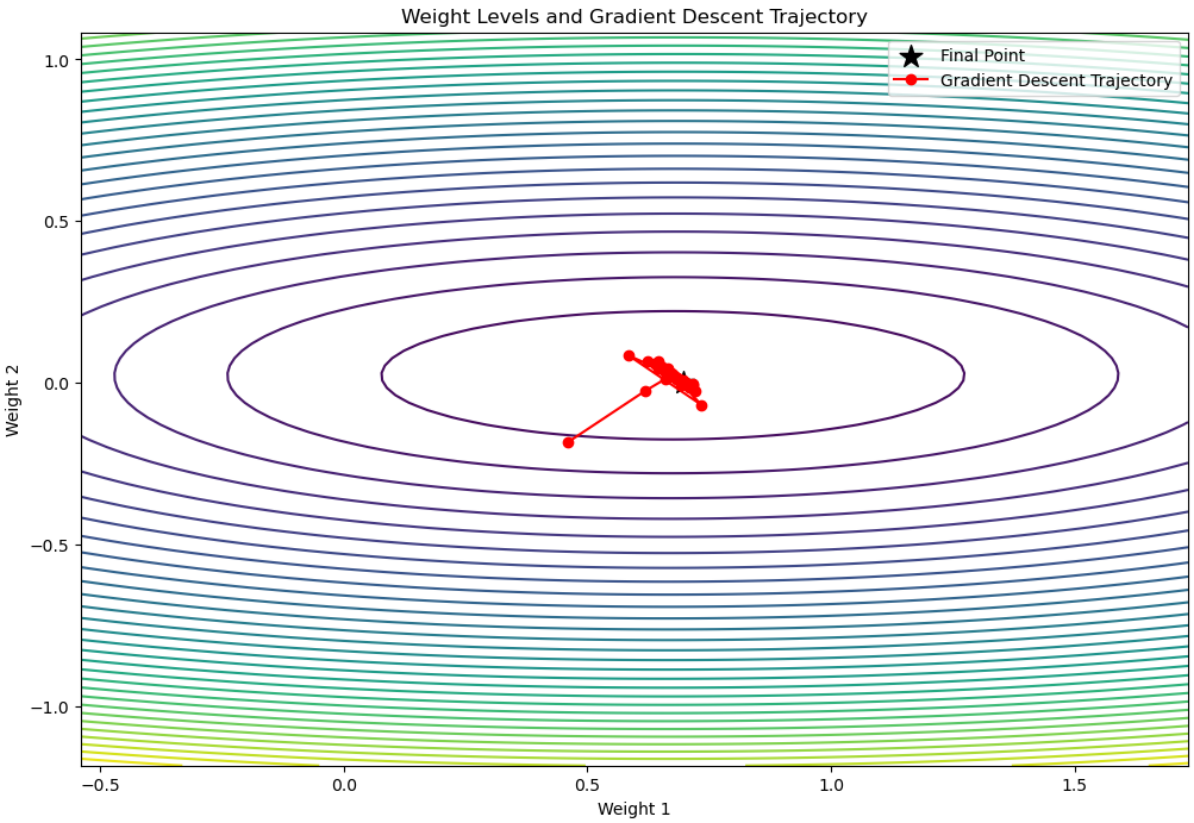
In [ ]:
```
w, w_history, momentum_errors = momentum_optimizer(X=X, y=y, momentum=0.6, epochs=100)
print(w_history[-1])
print()
plot_weight_levels(X, y, w_history)
```

```
[0.67547058 0.02324001]

[0.67547058 0.02324001]
```



In [ ]:
```
w, w_history, rmsprop_errors = rmsprop_optimizer(X=X, y=y, epochs=100)
print(w_history[-1])
print()
plot_weight_levels(X, y, w_history)
```

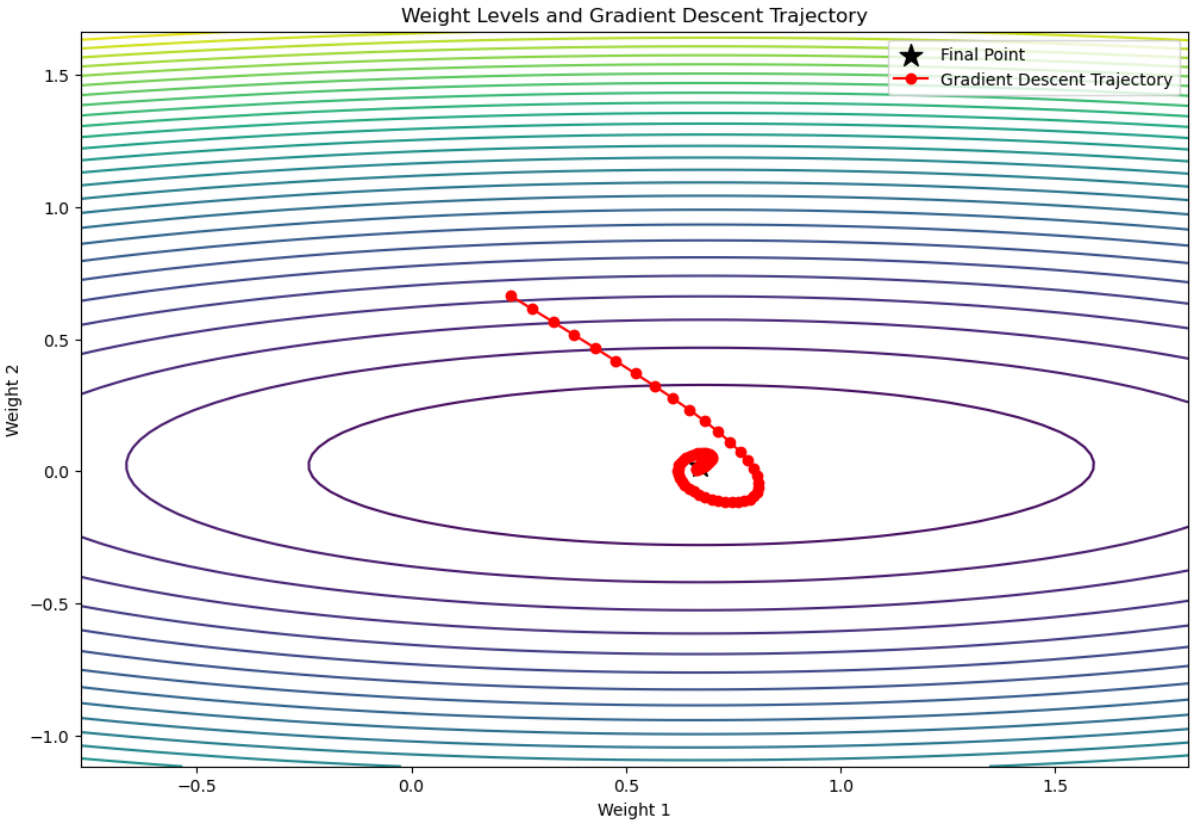
```
[0.69693174 0.00142556]

[0.69693174 0.00142556]
```



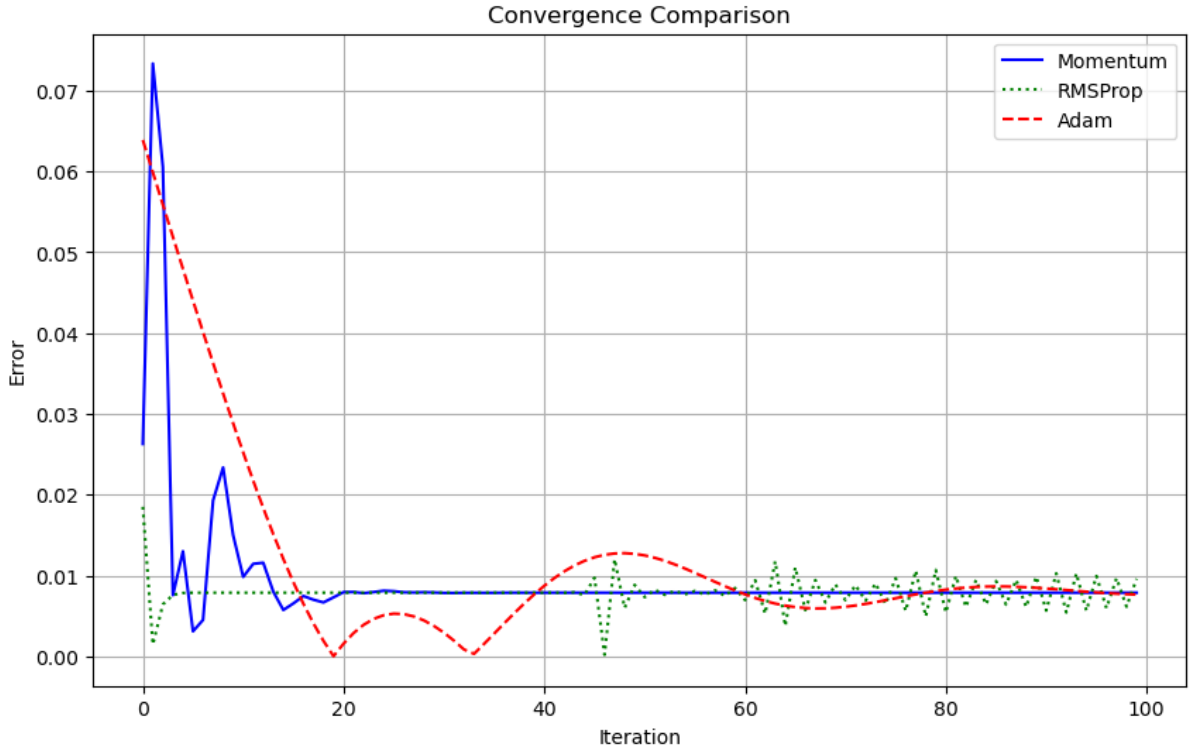
In [ ]:
```
w, w_history, adam_errors = adam_optimizer(X=X, y=y, epochs=100)
print(w_history[-1])
```

```
print()
plot_weight_levels(X, y, w_history)
```

```
[0.67335838 0.02113427]

[0.67335838 0.02113427]
```



In [ ]:
```
plot_convergence(abs(momentum_errors), abs(rmsprop_errors), abs(adam_errors))
```



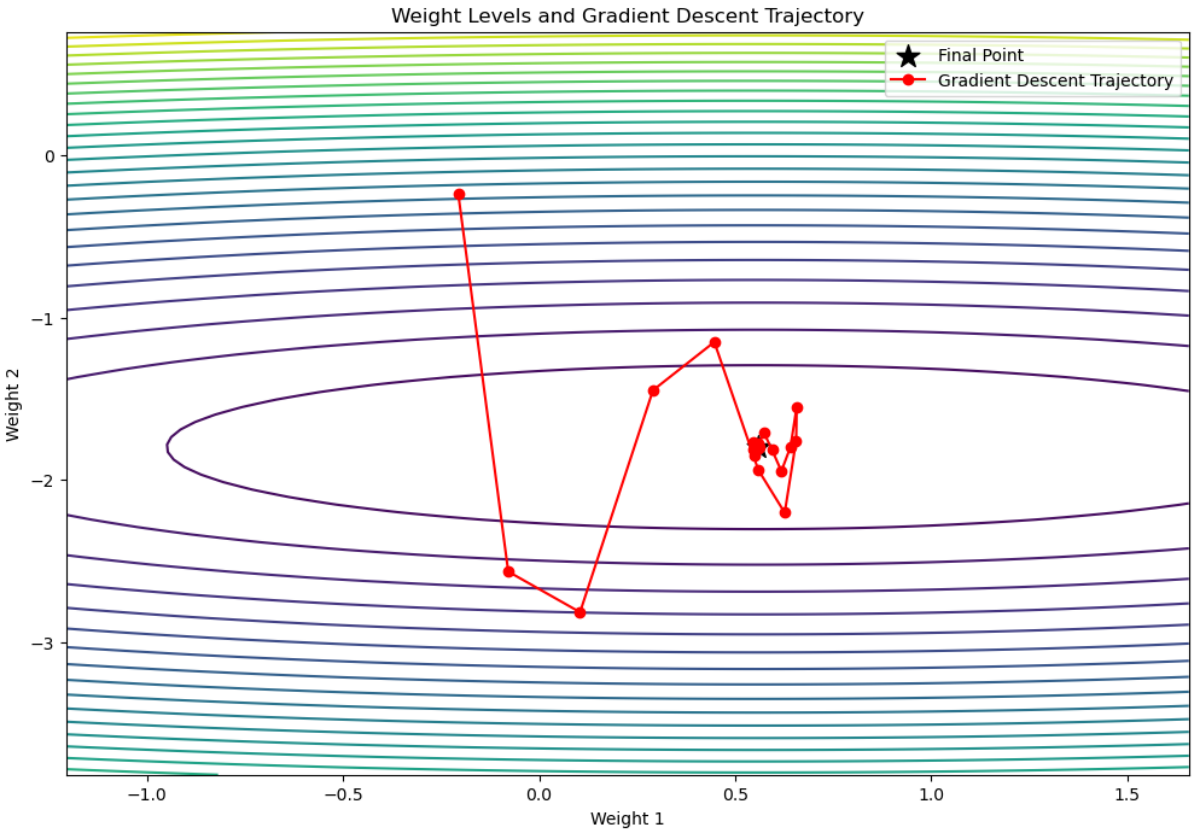
В этот раз RMSProp сошелся самым первым, однако после определенной эпохи он становится нестабильным. Самым долгим по сходимости оказался метод Adam.

Теперь сгенерируем выборку размером 100000 и проверим работу методов на ней.

In [ ]:
```
n, m = 100000, 2

w_true = np.random.standard_normal(m)

X = np.random.uniform(-5, 5, (n, m))
X *= (np.arange(m) * 2 + 1)[np.newaxis, :]

y = X.dot(w_true) + np.random.normal(0, 1, (n))
```

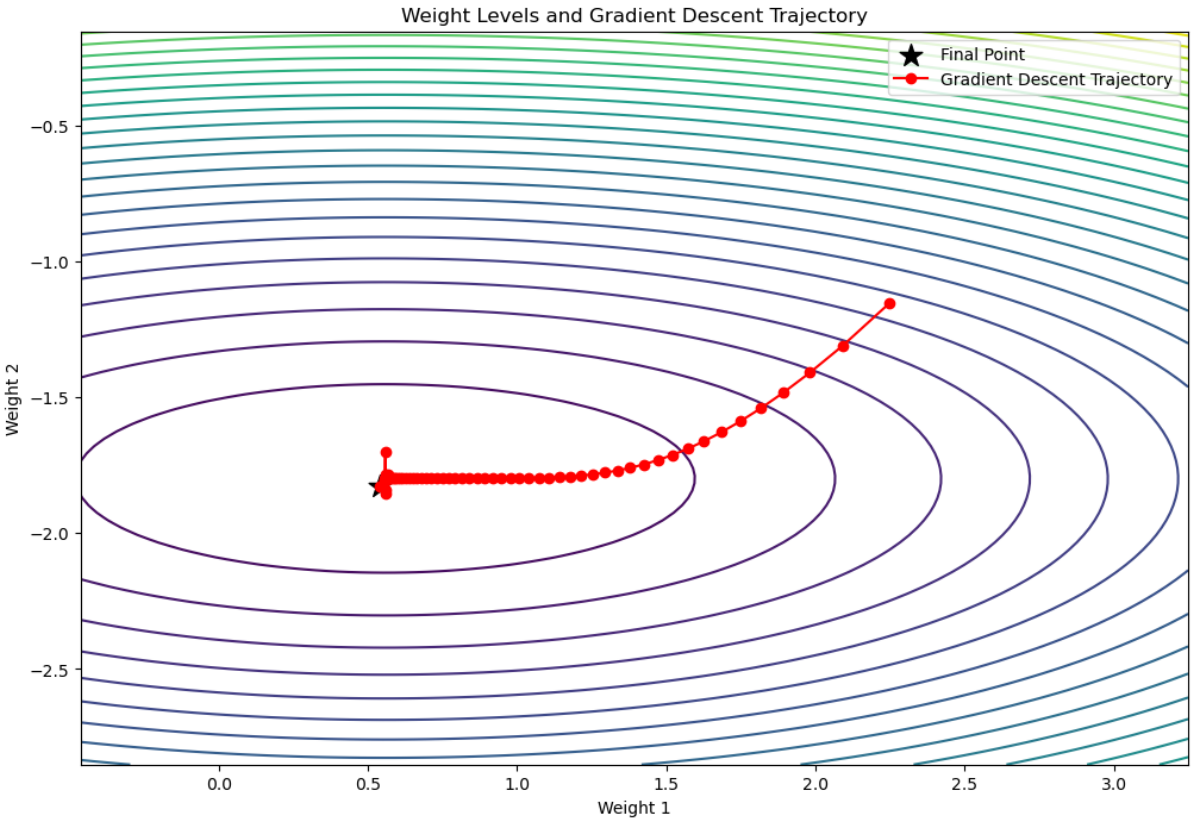
In [ ]:
```
w, w_history, momentum_errors = momentum_optimizer(X=X, y=y, momentum=0.6, epochs=100)
print(w_history[-1])
print()
plot_weight_levels(X, y, w_history)
```

```
[ 0.55824603 -1.79810092]

[ 0.55824603 -1.79810092]
```



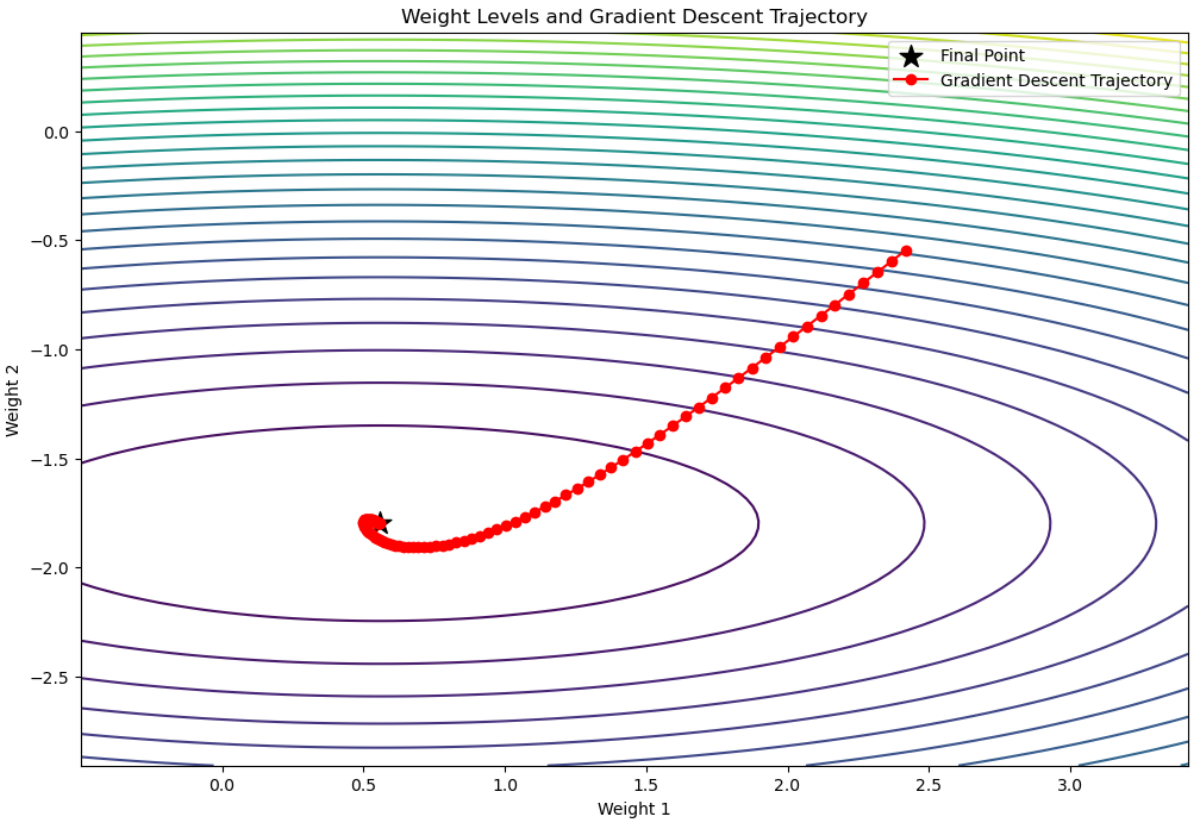
In [ ]:
```
w, w_history, rmsprop_errors = rmsprop_optimizer(X=X, y=y, epochs=100)
print(w_history[-1])
print()
plot_weight_levels(X, y, w_history)
```

```
[ 0.53856256 -1.82683591]

[ 0.53856256 -1.82683591]
```

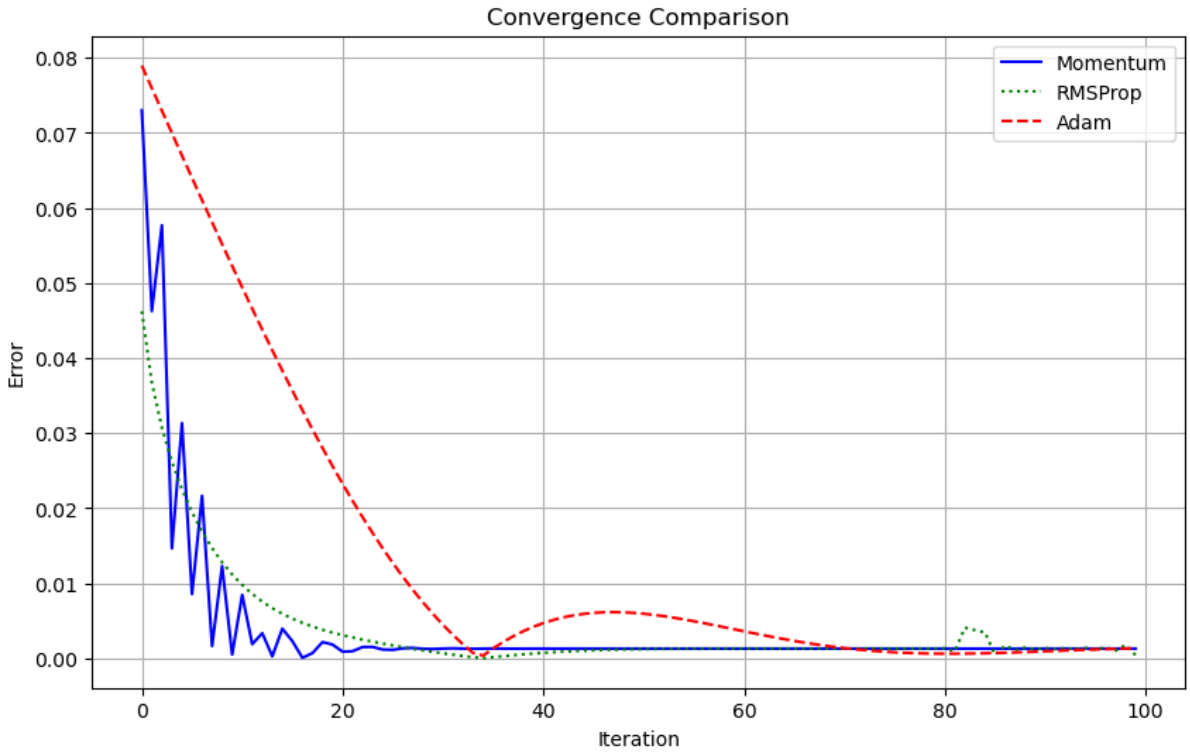


In [ ]:
```
w, w_history, adam_errors = adam_optimizer(X=X, y=y, epochs=100)
print(w_history[-1])
print()
plot_weight_levels(X, y, w_history)
```

```
[ 0.55695582 -1.79927416]

[ 0.55695582 -1.79927416]
```



In [ ]:
```
plot_convergence(abs(momentum_errors), abs(rmsprop_errors), abs(adam_errors))
```



В этот раз RMSProp сошелся примерно одновременно с Momentum. Самым долгим по сходимости снова оказался метод Adam.

Вывод

Из проведенных эксперементов можно сделать следующие выводы:

- В целом все три метода показывают неплохую сходимость, но быстрее всех сходились Momentum и RMSProp. Adam показал не самый лучший результат. Однако если рассмотреть применение этого оптимизатора в более сложных моделях возможно, что Adam покажет хорошие результаты, благодаря своей адаптивной природе и комбинации метода инерции и RMSprop.
- Метод инерции (Momentum) может обеспечивать более быструю сходимость в начале обучения, особенно при отсутствии больших колебаний в данных. Однако, на последних этапах обучения Adam может обеспечивать лучшую устойчивость и точность.