Домашнее задание №11

По аналогии с реализацией GD и SGD реализуйте метод инерции(Momentum), RMSprop и Adam.

Для произвольной задачи (можно сгенерировать синтетически матрицу X и вектор y) сравните результаты работы трех методов оптимизации.

Проведите эксперименты с этими методами для задач разных размерностей, сравните оценки сходимости и сделайте выводы по эмпирическим результатам о скорости и точности сходимости в зависимости от размерности и параметров моделей.

In [1]:
```
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
import pandas as pd
from itertools import product
from typing import Tuple
import warnings

np.random.seed(0)

warnings.filterwarnings('ignore')
%matplotlib inline
```

Реализация метода инерции(Momentum)

Momentum — это метод оптимизации алгоритмов машинного обучения, который позволяет преодолевать недостатки градиентного спуска, такие как медленное схождение и застревание в локальных минимумах.

Основные гиперпараметры RMSprop:

- α - скорость обучения
- β - инерция из предыдущего момента времени

$$v_{t+1} = \beta v_t + (1 - \beta)\nabla Q(w_t) \tag{1}$$

$$w_{t+1} = w_t - \alpha v_{t+1} \tag{2}$$

In [2]:
```
def generate_batch(X, y, batch_size):
    """ Генератор для получения батча из данных """
    for i in range(0, len(X), batch_size):
        yield X[i : i + batch_size], y[i : i + batch_size]
```

In [3]:
```
def stochastic_gradient_descent_with_momentum(
    epochs: int,
    batch_size: int,
    alpha: float,
    beta: float,
    X: np.ndarray,
    y: np.ndarray,
    w = None,
    max_iters=1000
) -> Tuple[np.ndarray, np.ndarray, np.ndarray]:
```

```
    """Функция для оптимизации весов с помощью стохастического градиентного спуска

    Args:
        epochs (int): количество эпох
        batch_size (int): размер батча
        alpha (float): длина шага
        X (np.ndarray): Матрица объектов-признаков
        y (np.ndarray): Вектор таргетов
        w (_type_, optional): Начальное значение для вектора весов. Defaults to None.

    Returns:
        Tuple[np.ndarray, np.ndarray, np.ndarray]: Возвращает полученные веса, вектор с
    """
    n, d = X.shape

    if w is None:
        w = np.random.standard_normal(d)

    w_cur = w.copy()
    v = np.zeros(d)
    w_history = [w_cur]
    err_history = []
    n_iter = 0

    for _ in range(epochs):
        p = np.random.permutation(len(X)) # случайно перемешиваем выборку
        batch_generator = generate_batch(X[p], y[p], batch_size)  # инициализируем генер

        for X_batch, y_batch in batch_generator:  # Итерируемся по полученными батчам
            y_pred = X_batch.dot(w_cur)
            err = y_pred - y_batch
            grad = 2 * X_batch.T.dot(err) / n
            v = beta * v + (1 - beta) * grad  # Расчет v
            w_cur -= alpha * v

            w_history.append(w_cur.copy())
            err_history.append(np.abs(err).mean())
            n_iter += 1
            if n_iter == max_iters:
                 return w, np.array(w_history), np.array(err_history)

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

Реализация метода RMSprop

Метод RMSprop (root mean square prop) является методом оптимизации, применяемым в алгоритмах машинного обучения для обновления параметров модели на основе градиентов функции потерь. Ключевая особенность метода RMSprop заключается в использовании экспоненциально взвешенного скользящего среднего квадрата градиентов. Это позволяет адаптировать скорость обучения для каждого параметра модели при обучении. В отличие от обычного градиентного спуска, которому требуется одно и то же значение скорости обучения для всех параметров, RMSprop регулирует скорость обучения в зависимости от изменчивости градиентов для каждого параметра.

Основные гиперпараметры RMSprop:

- α - скорость обучения
- γ - коэффициент сглаживания
- ϵ - маленькое число, добавленное для численной стабильности

$$G_t = \gamma G_{t-1} + (1-\gamma)\nabla Q(w_t)^2 \tag{3}$$

$$w_{t+1} = w_t - \frac{\alpha}{\sqrt{G_t + \epsilon}} \odot \nabla Q(w_t) \tag{4}$$

In [4]:
```python
def stochastic_gradient_descent_with_rmsprop(
    epochs: int,
    batch_size: int,
    alpha: float,
    gamma: float,
    X: np.ndarray,
    y: np.ndarray,
    w = None,
    e=1e-8,
    max_iters=1000
) -> Tuple[np.ndarray, np.ndarray, np.ndarray]:
    """Функция для оптимизации весов с помощью стохастического градиентного спуска

    Args:
        epochs (int): количество эпох
        batch_size (int): размер батча
        alpha (float): длина шага
        X (np.ndarray): Матрица объектов-признаков
        y (np.ndarray): Вектор таргетов
        w (_type_, optional): Начальное значение для вектора весов. Defaults to None.

    Returns:
        Tuple[np.ndarray, np.ndarray, np.ndarray]: Возвращает полученные веса, вектор с
    """
    n, d = X.shape

    if w is None:
        w = np.random.standard_normal(d)

    w_cur = w.copy()
    G = np.zeros(d)
    w_history = [w_cur]
    err_history = []
    n_iter = 0

    for _ in range(epochs):
        p = np.random.permutation(len(X)) # случайно перемешиваем выборку
        batch_generator = generate_batch(X[p], y[p], batch_size)  # инициализируем генер

        for X_batch, y_batch in batch_generator:  # Итерируемся по полученными батчам
            y_pred = X_batch.dot(w_cur)
            err = y_pred - y_batch
            grad = 2 * X_batch.T.dot(err) / n
            G = gamma * G + (1 - gamma) * (grad)**2 # Расчет G
            w_cur -= (alpha / ((G + e)**0.5)) * grad

            w_history.append(w_cur.copy())
            err_history.append(np.abs(err).mean())
            n_iter += 1
            if n_iter == max_iters:
                 return w, np.array(w_history), np.array(err_history)

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

Реализация метода Adam

Adam (Adaptive Moment Estimation) — это адаптивный алгоритм обучения, предназначенный для

улучшения скорости обучения глубоких нейронных сетей и быстрого достижения сходимости. Он настраивает скорость обучения каждого параметра на основе истории градиента, что помогает нейронной сети учиться эффективно.

Основные гиперпараметры Адама:

- η — размер шага для оптимизации;
- β_1 — скорость затухания для импульса;
- β_2 — скорость затухания для квадратов градиентов;
- ϵ — малое значение для предотвращения деления на ноль. Вместе эти параметры позволяют Адаму быстрее сходиться, оставаясь численно стабильным

$$
\begin{aligned}
m_t &= \beta_1 * m_{t-1} + (1-\beta_1) * \nabla w_t \\
v_t &= \beta_2 * v_{t-1} + (1-\beta_2) * (\nabla w_t)^2 \\
\hat{m}_t &= \frac{m_t}{1-{\beta_1}^t} \\
\hat{v}_t &= \frac{v_t}{1-{\beta_2}^t} \\
w_{t+1} &= w_t - \frac{\eta}{\sqrt{\hat{v}_t + \epsilon}} * \hat{m}_t
\end{aligned}
$$

In [5]:
```python
def stochastic_gradient_descent_with_adam(
    epochs: int,
    batch_size: int,
    alpha: float,
    beta1: float,
    beta2: float,
    X: np.ndarray,
    y: np.ndarray,
    w = None,
    e=1e-8,
    max_iters=1000
) -> Tuple[np.ndarray, np.ndarray, np.ndarray]:
    """Функция для оптимизации весов с помощью стохастического градиентного спуска

    Args:
        epochs (int): количество эпох
        batch_size (int): размер батча
        alpha (float): длина шага
        X (np.ndarray): Матрица объектов-признаков
        y (np.ndarray): Вектор таргетов
        w (_type_, optional): Начальное значение для вектора весов. Defaults to None.

    Returns:
        Tuple[np.ndarray, np.ndarray, np.ndarray]: Возвращает полученные веса, вектор с
    """
    n, d = X.shape

    if w is None:
        w = np.random.standard_normal(d)

    w_cur = w.copy()
    m = np.zeros(d)
    v = np.zeros(d)
    w_history = [w_cur]
    err_history = []
    n_iter = 0
```

```
    for _ in range(epochs):
        p = np.random.permutation(len(X)) # случайно перемешиваем выборку
        batch_generator = generate_batch(X[p], y[p], batch_size)  # инициализируем генер

        for X_batch, y_batch in batch_generator:  # Итерируемся по полученными батчам
            y_pred = X_batch.dot(w_cur)
            err = y_pred - y_batch
            grad = 2 * X_batch.T.dot(err) / n
            m = beta1 * m + (1 - beta1) * grad # Расчет m
            v = beta2 * v + (1 - beta2) * grad**2 # Расчет v
            m_hat = m / (1 - beta1) # Расчет m с шапкой
            v_hat = v / (1 - beta2) # Расчет v с шапкой
            w_cur -= (alpha / ((v_hat + e)**0.5)) * m_hat

            w_history.append(w_cur.copy())
            err_history.append(np.abs(err).mean())
            n_iter += 1
            if n_iter == max_iters:
                 return w, np.array(w_history), np.array(err_history)

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

Сравнение результатов работы трех методов оптимизации

Сгенерируем двумерную синтетическую матрицу X и двумерный вектор y размерами 500 элементов. В этом и последующих экспериментах для ускорения и правдоподобности(как быдто у нас реальная задача обучения нейронной сети) их мы зафиксирует максимальное число эпох и итераций равными 1000, а размер батча примем равным 32.

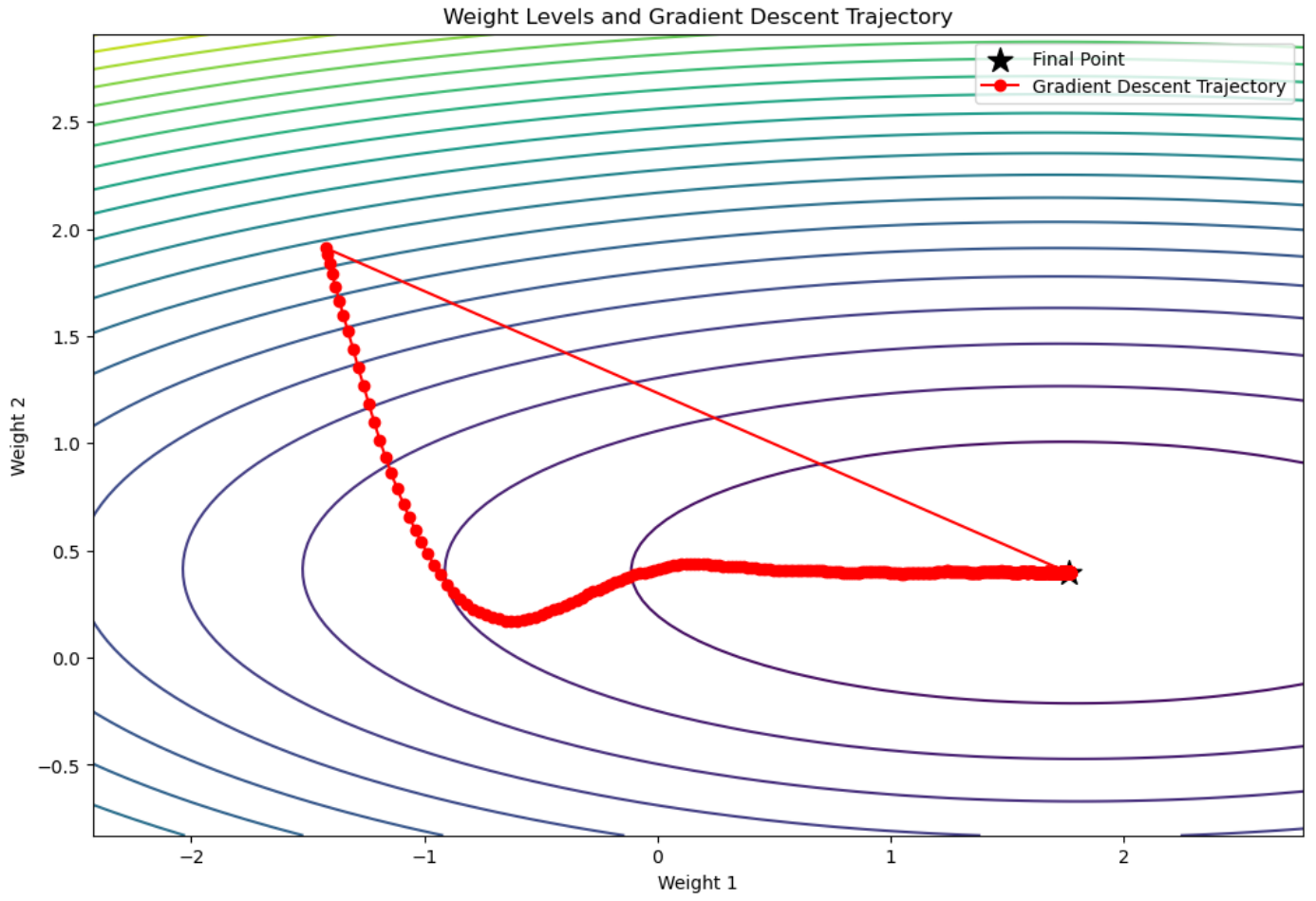
In [6]:
```
n, d = 500, 2

w_true = np.random.standard_normal(d)

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

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

In [7]:
```
# Код для визуализации линий уровня и траектории градиентного спуска
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')
```

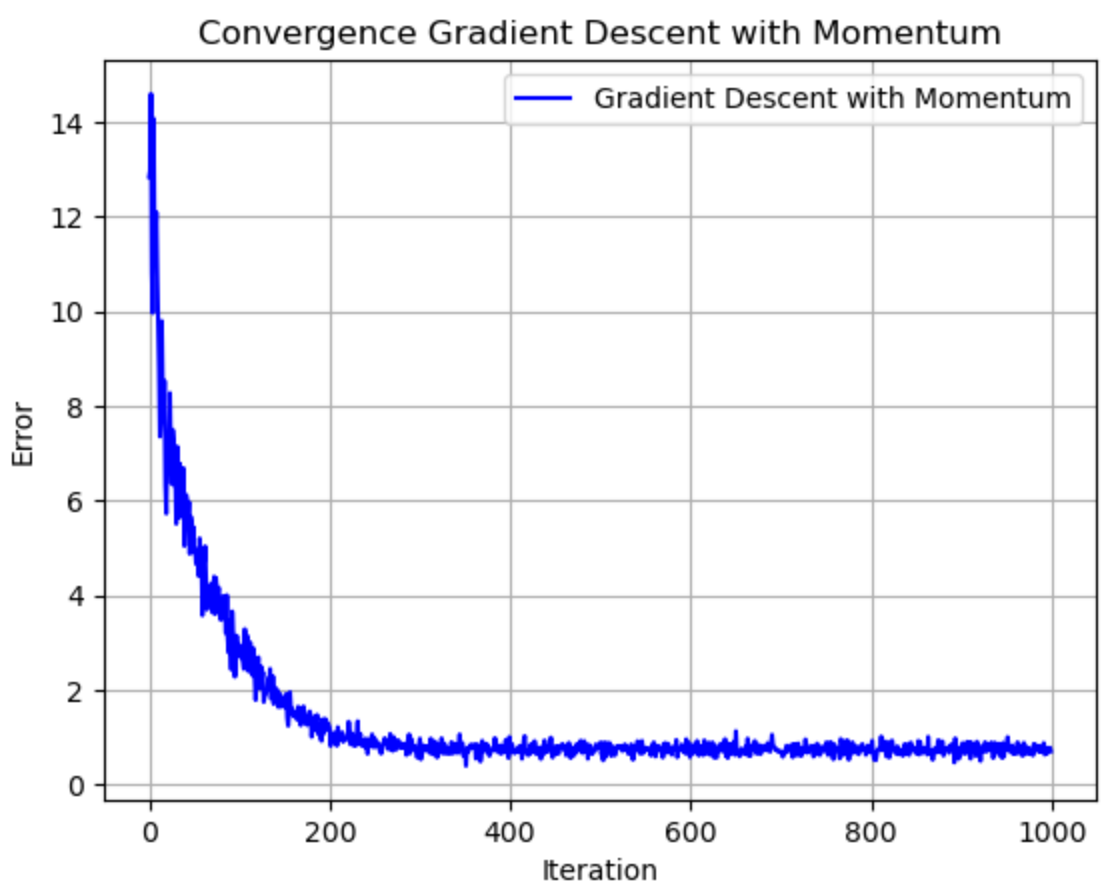
```
    plt.xlabel('Weight 1')
    plt.ylabel('Weight 2')
    plt.legend()
    plt.show()
```

In [8]:
```
# Код для сравнения зависимости величины ошибки от количества итераций
def plot_convergence(momentum_error, rmsprop_error, adam_error):
    plt.figure(figsize=(10, 6))

    plt.plot(momentum_error, label='Gradient Descent with Momentum', color='blue')

    plt.plot(rmsprop_error, label='Gradient Descent with RMSprop', color='red', linestyl

    plt.plot(adam_error, label='Gradient Descent with Adam', color='green', linestyle='-

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

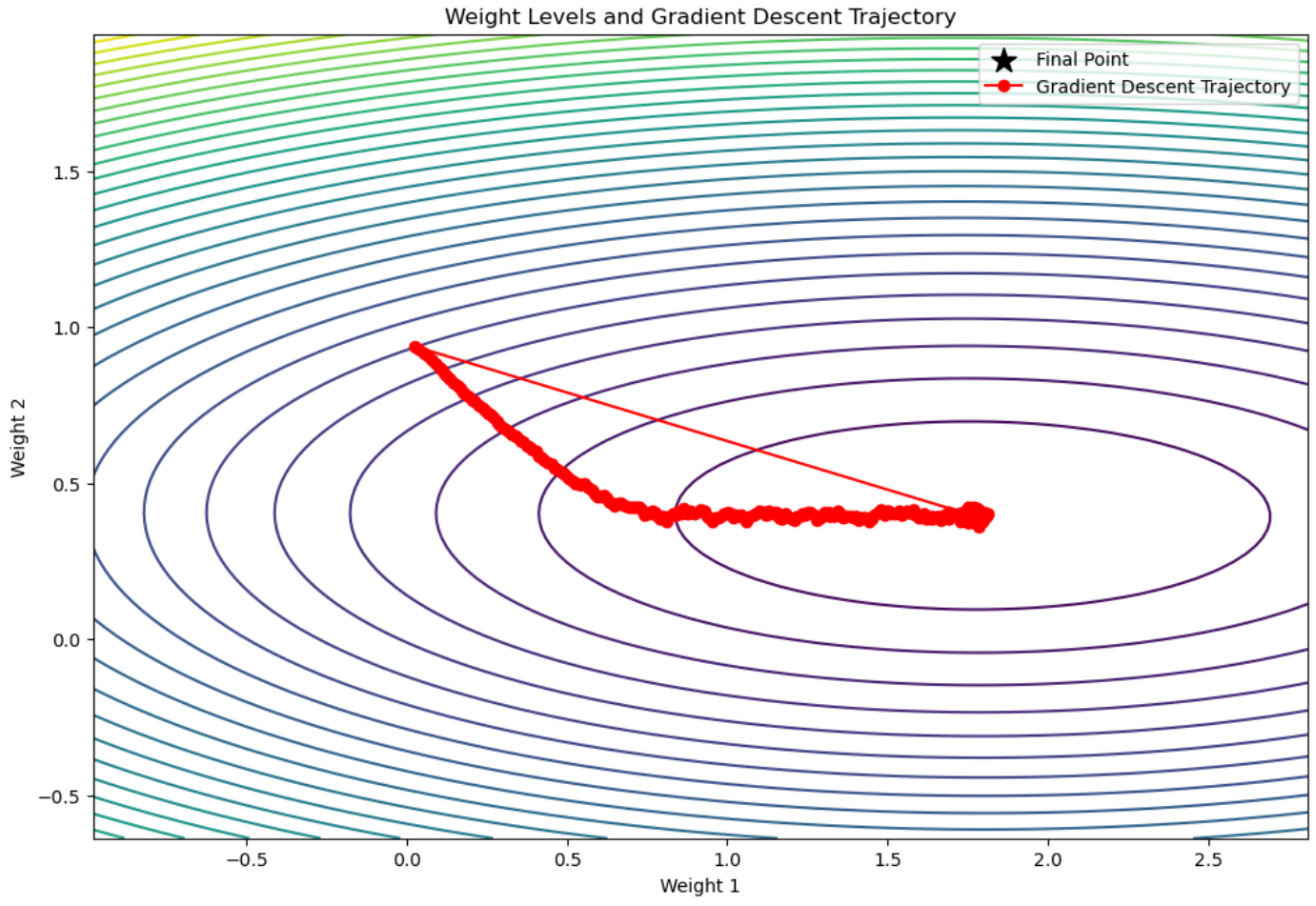
In [9]:
```
w_momentum, w_history_momentum, w_error_momentum = stochastic_gradient_descent_with_mome
plot_weight_levels(X, y, w_history_momentum)
```



In [10]:
```
plt.plot(w_error_momentum, label='Gradient Descent with Momentum', color='blue', linesty

plt.title('Convergence Gradient Descent with Momentum')
plt.xlabel('Iteration')
plt.ylabel('Error')
plt.legend()
plt.grid(True)
plt.show()
```

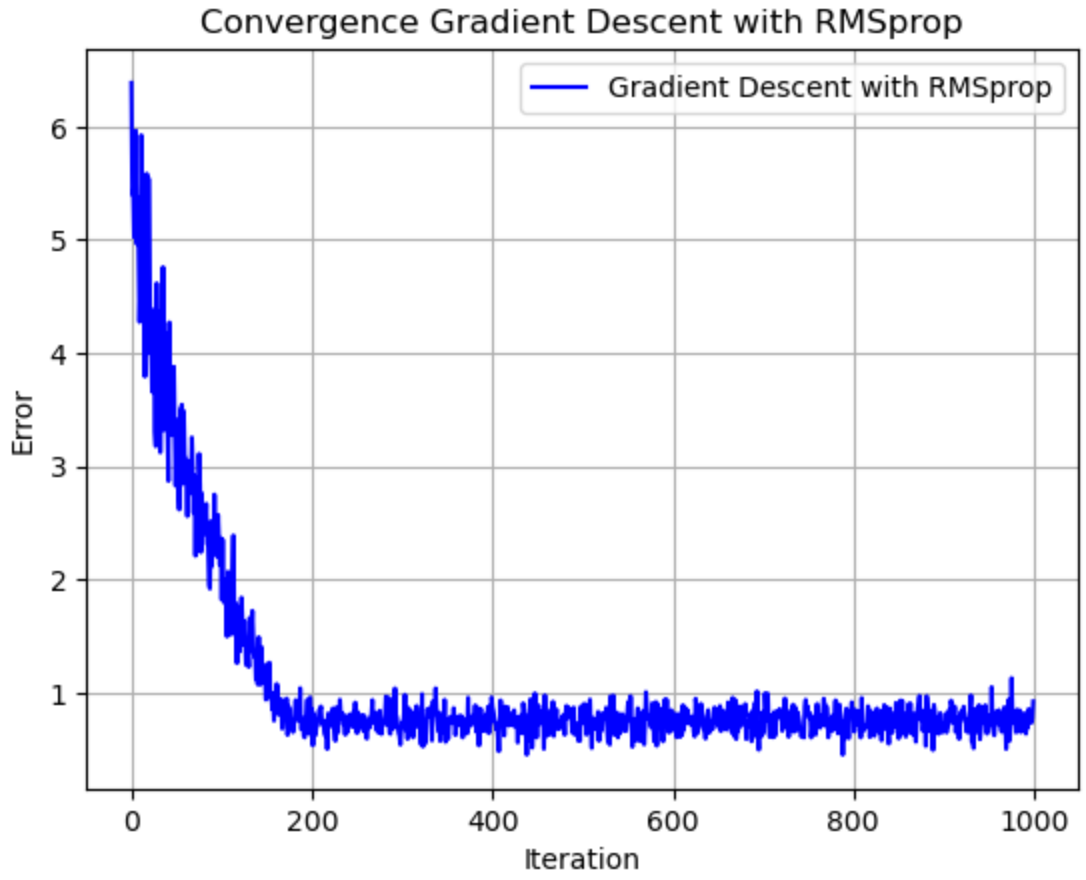


In [11]:
```
w_rmsprop, w_history_rmsprop, w_error_rmsprop = stochastic_gradient_descent_with_rmsprop
plot_weight_levels(X, y, w_history_rmsprop)
```

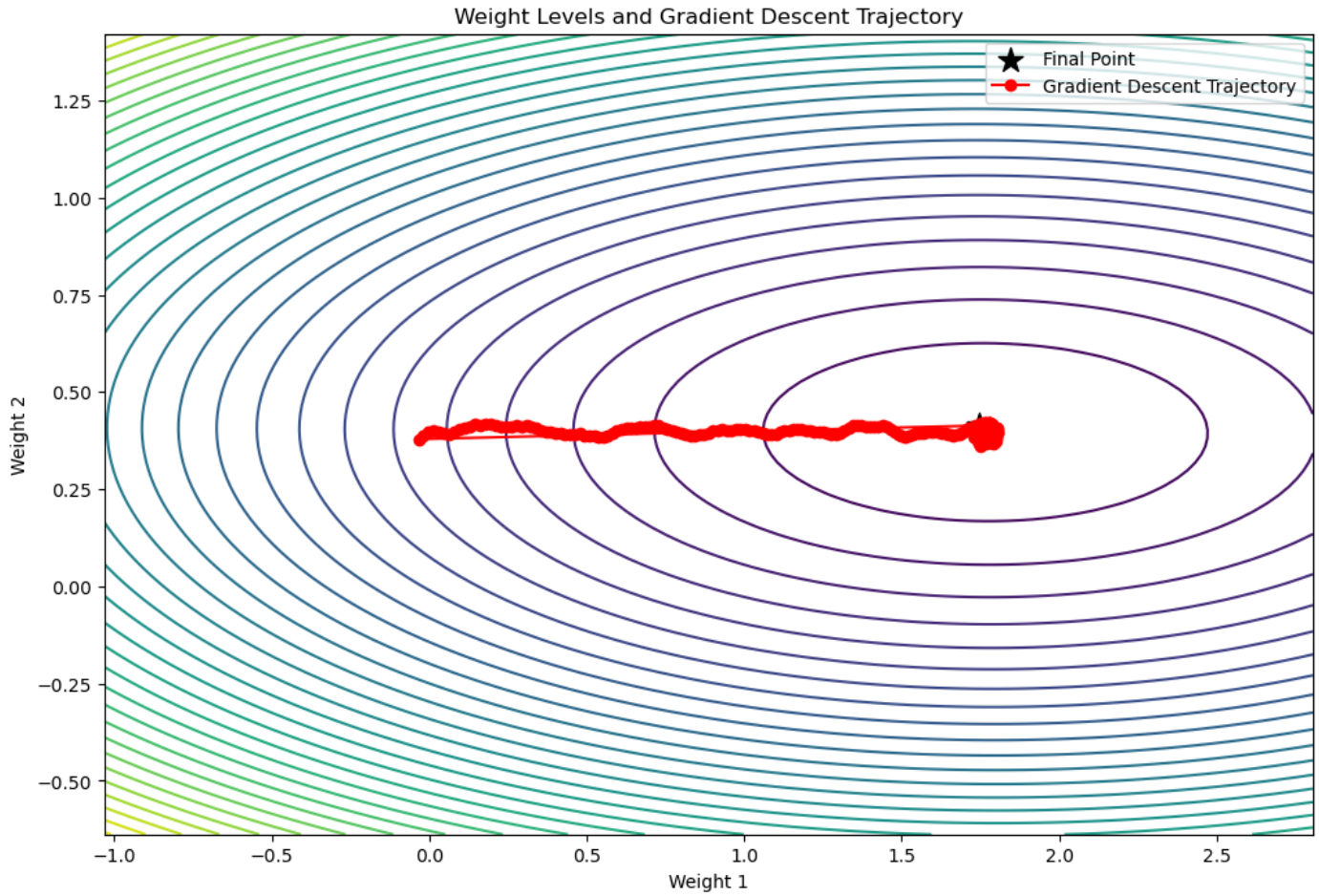


In [12]:
```
plt.plot(w_error_rmsprop, label='Gradient Descent with RMSprop', color='blue', linestyle
```

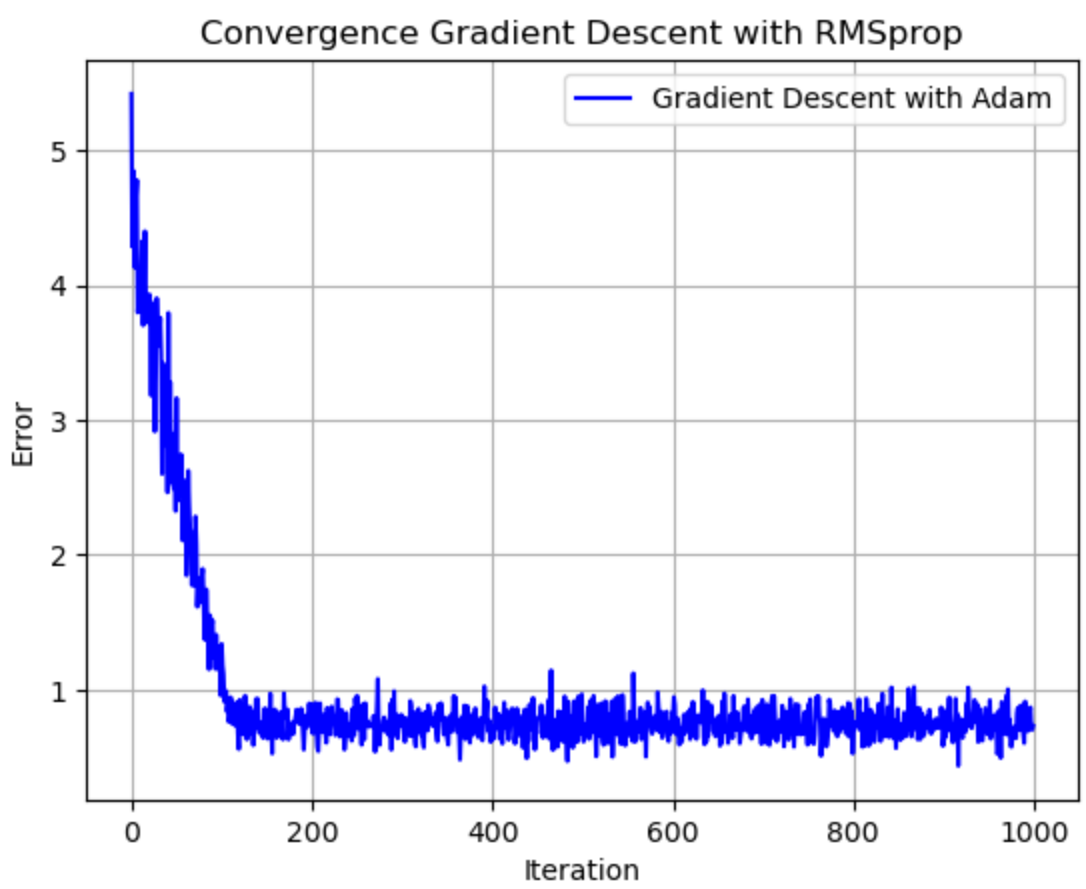
```
plt.title('Convergence Gradient Descent with RMSprop')
plt.xlabel('Iteration')
plt.ylabel('Error')
plt.legend()
plt.grid(True)
plt.show()
```



In [13]:
```
w_adam, w_history_adam, w_error_adam = stochastic_gradient_descent_with_adam(1000, 32, 1
plot_weight_levels(X, y, w_history_adam)
```

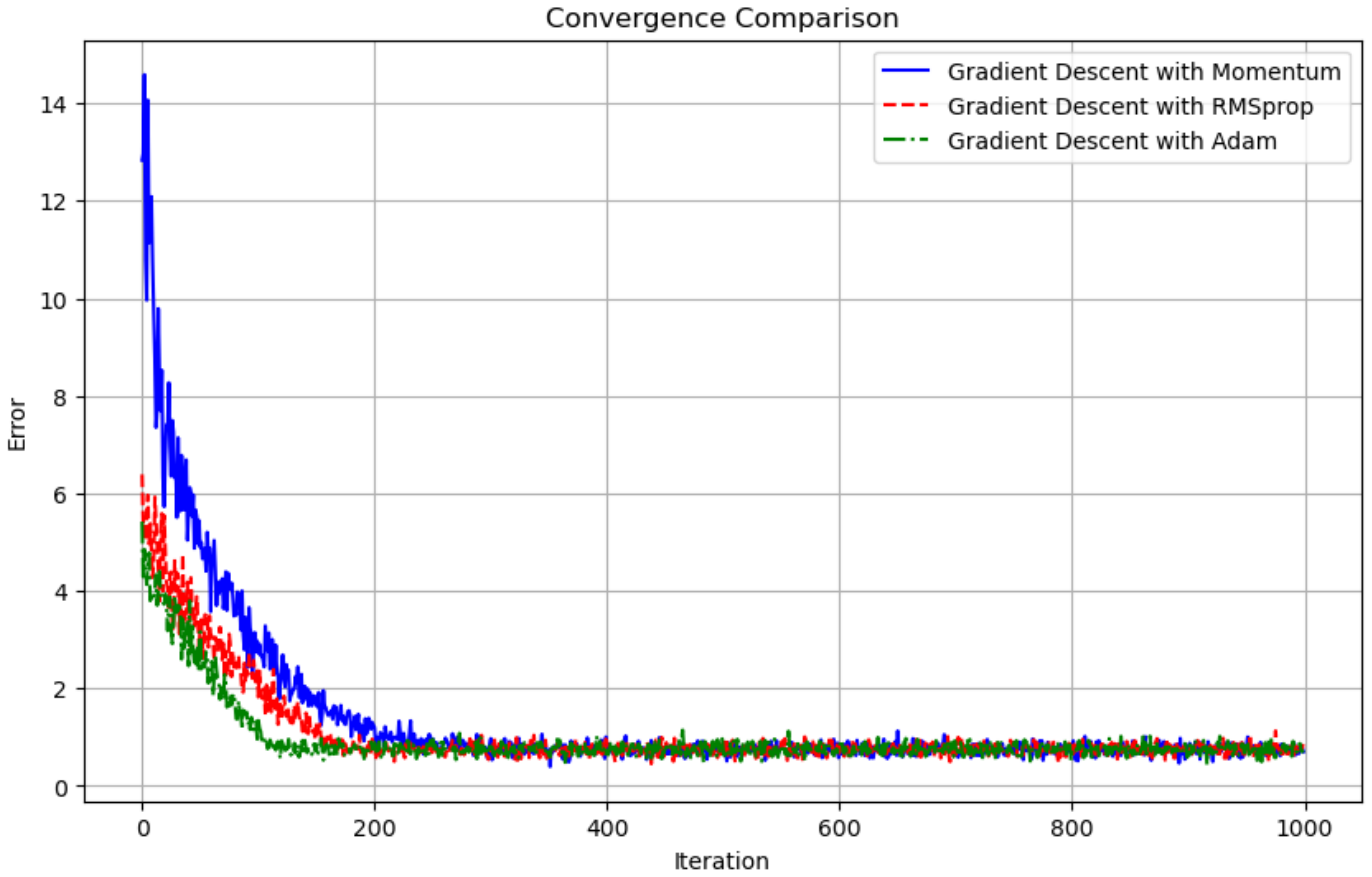


In [14]:
```
plt.plot(w_error_adam, label='Gradient Descent with Adam', color='blue', linestyle='-')

plt.title('Convergence Gradient Descent with RMSprop')
plt.xlabel('Iteration')
plt.ylabel('Error')
plt.legend()
plt.grid(True)
plt.show()
```



Нанесем теперь все 3 кривые на 1 график и визуально сравним их.

In [15]:
```
plot_convergence(w_error_momentum, w_error_rmsprop, w_error_adam)
```



Вывод: как видно из графиков градиентный спуск с оптимизатором Adam сходится быстрее остальных за меньше чеисло итераций. Наиболее устойчивой и "непетляющей" к минимуму

траекторией обладает метод RMSprop.

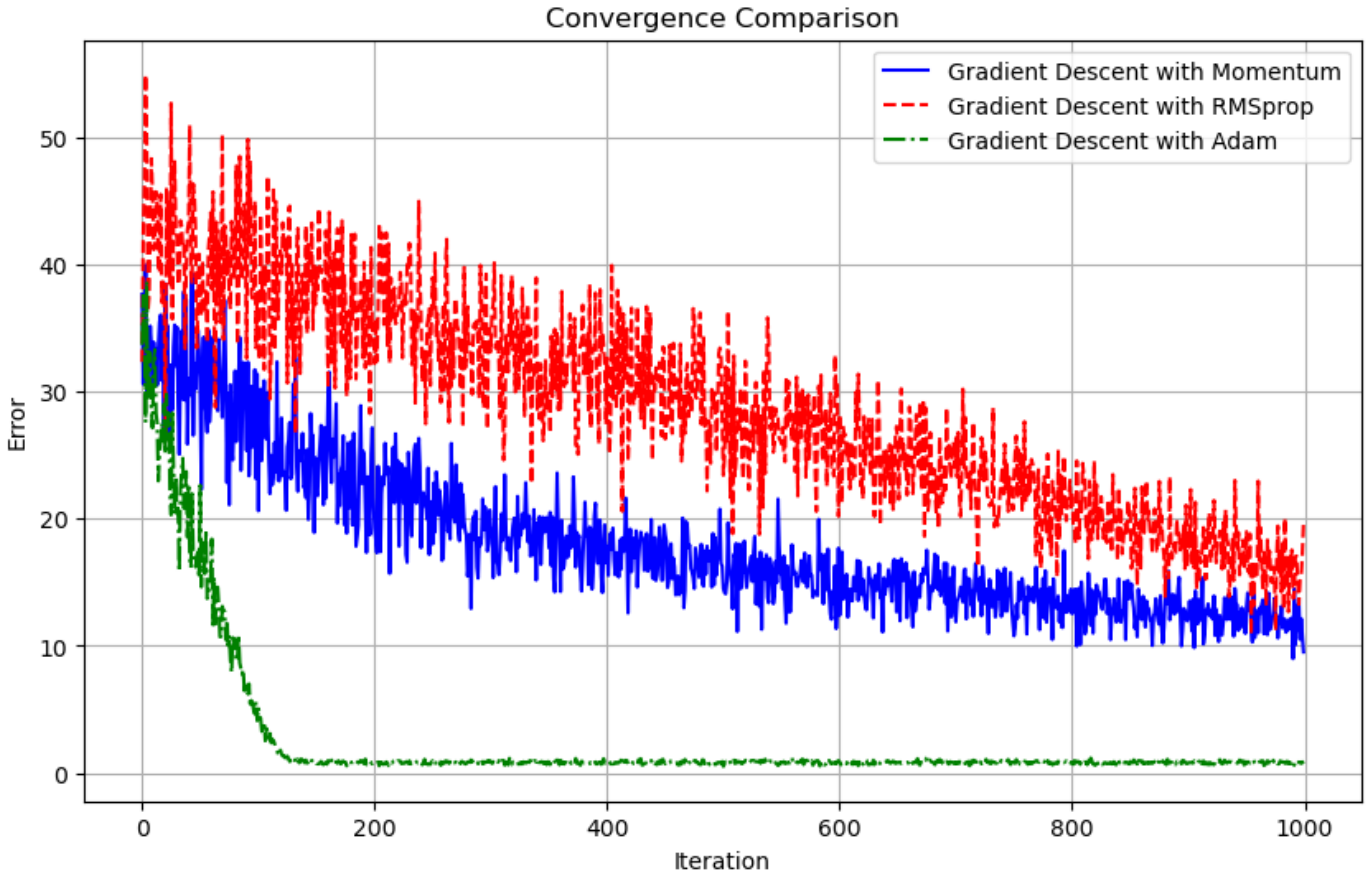
Эксперименты с оптимизаторами для задач разных размерностей

Проведем эксперименты с выборкой размера 10000 и количеством признаков 5

In [16]:
```
n, d = 10000, 5

w_true = np.random.standard_normal(d)

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

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

In [17]:
```
w_momentum, w_history_momentum, w_error_momentum = stochastic_gradient_descent_with_mome
w_rmsprop, w_history_rmsprop, w_error_rmsprop = stochastic_gradient_descent_with_rmsprop
w_adam, w_history_adam, w_error_adam = stochastic_gradient_descent_with_adam(1000, 32, 1
```

In [18]:
```
plot_convergence(w_error_momentum, w_error_rmsprop, w_error_adam)
```

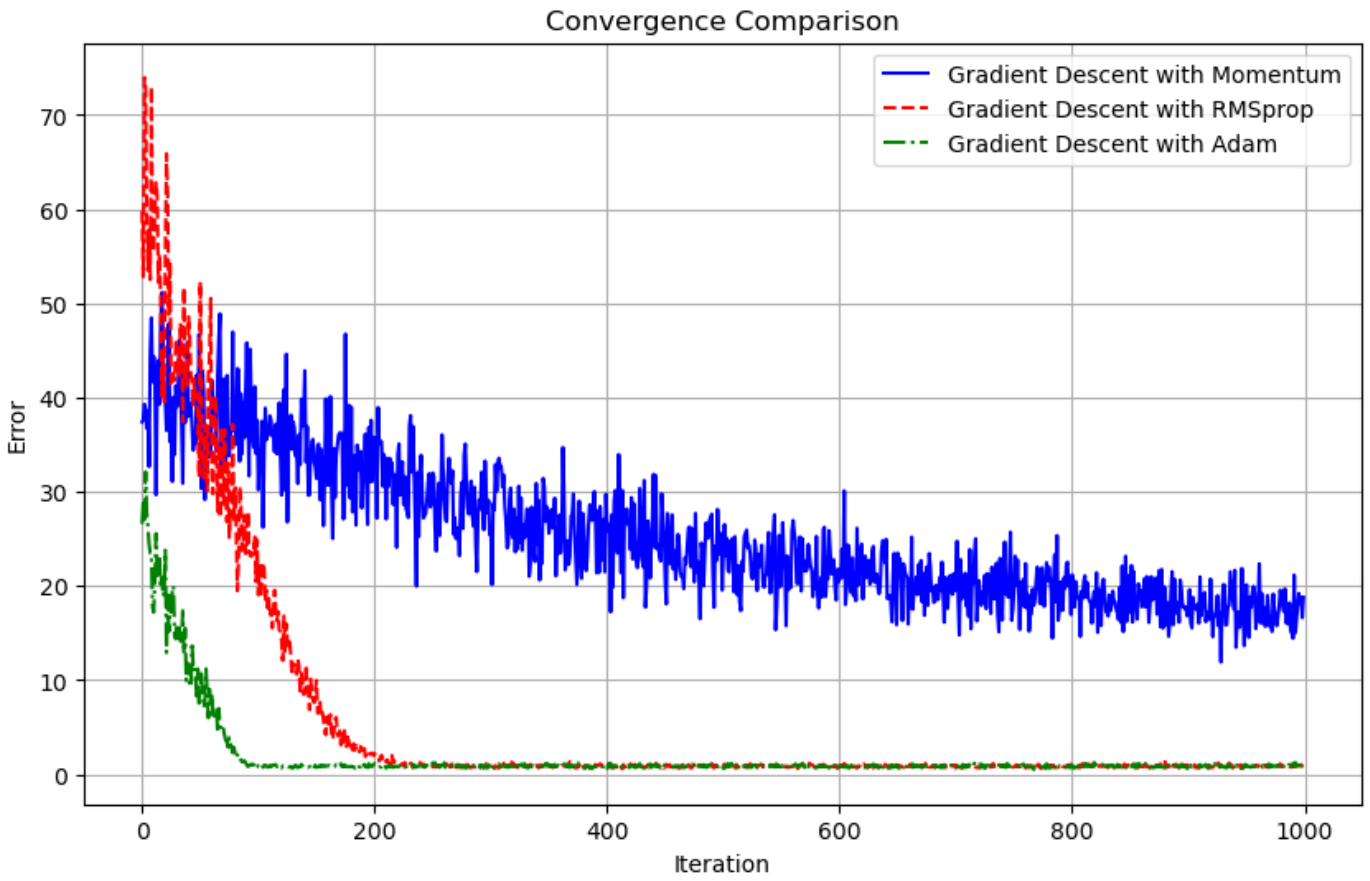


Теперь увеличим размер выборки до 150000 и количества признаков до 7

In [19]:
```
n, d = 150000, 7

w_true = np.random.standard_normal(d)

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

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

In [20]:
```
w_momentum, w_history_momentum, w_error_momentum = stochastic_gradient_descent_with_mome
w_rmsprop, w_history_rmsprop, w_error_rmsprop = stochastic_gradient_descent_with_rmsprop
w_adam, w_history_adam, w_error_adam = stochastic_gradient_descent_with_adam(1000, 32, 1
```

In [21]:
```
plot_convergence(w_error_momentum, w_error_rmsprop, w_error_adam)
```



Вывод: как видно из графиков при увеличении размера выборки и количества признаков градиентный спуск с оптимизатором Adam сходится быстрее остальных за меньше чеисло итераций. Также он наиболее обладает устойчивой и "непетляющей" траекторией спуска. Метод инерции при таких параметрах и размере выборки не сходится за данное количество эпох и итераций.

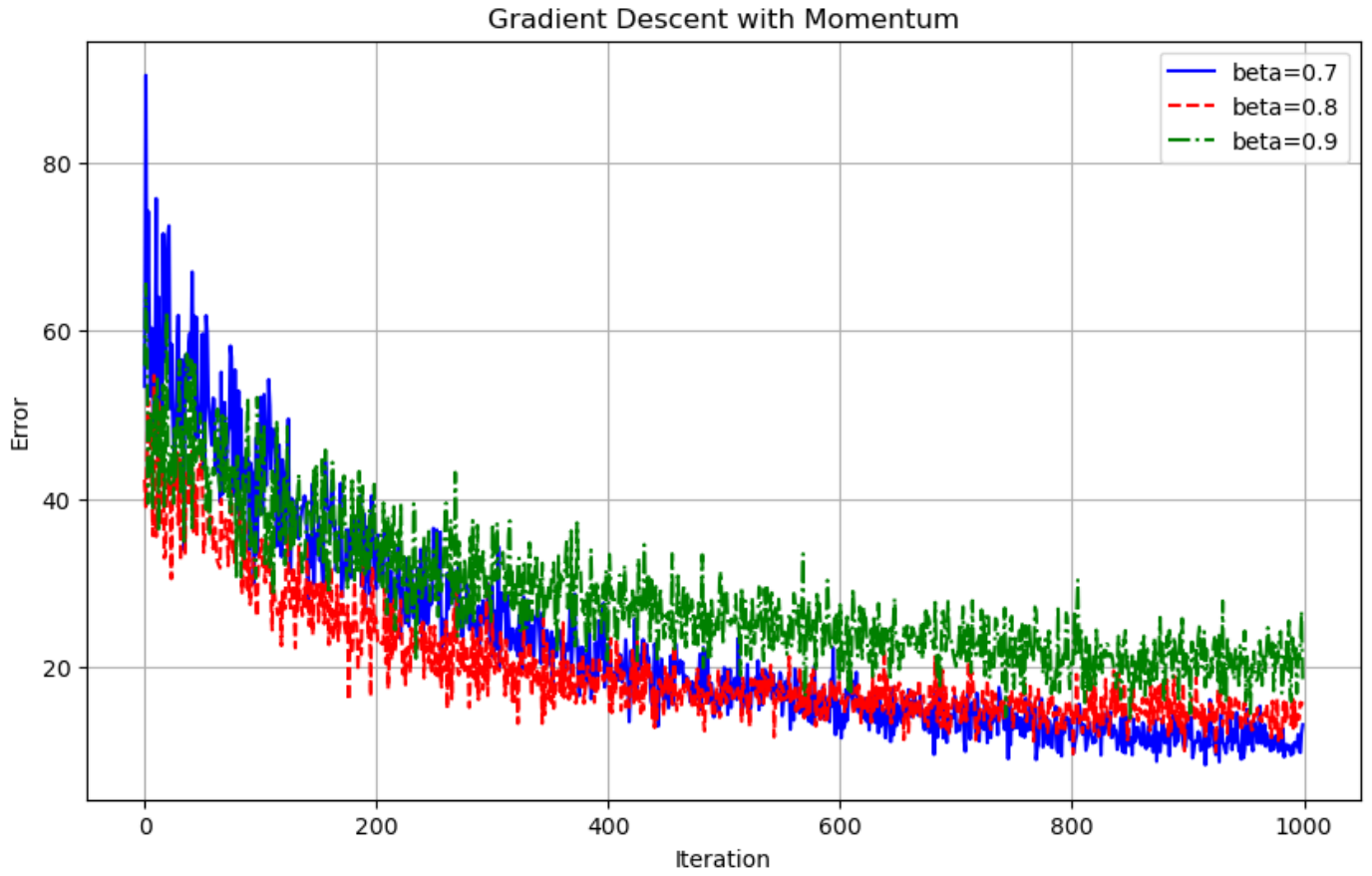
Эксперименты с параметрами для метода инерции(Momentum)

Проведем эксперименты с подбором параметра для каждого из оптимизаторов. Начнем с метода инерции, у него 2 параметра, их и подберем.

In [22]:
```
momentum_betas = { 'momentum_0_7': [], 'momentum_0_8': [], 'momentum_0_9': [] }

for beta in [0.7, 0.8, 0.9]:
    if beta == 0.7:
        w_momentum, w_history_momentum, w_error_momentum = stochastic_gradient_descent_w
        momentum_betas['momentum_0_7'] = w_error_momentum
    elif beta == 0.8:
        w_momentum, w_history_momentum, w_error_momentum = stochastic_gradient_descent_w
        momentum_betas['momentum_0_8'] = w_error_momentum
    else:
        w_momentum, w_history_momentum, w_error_momentum = stochastic_gradient_descent_w
        momentum_betas['momentum_0_9'] = w_error_momentum
```

In [23]:
```
plt.figure(figsize=(10, 6))

plt.plot(momentum_betas['momentum_0_7'], label='beta=0.7', color='blue')
```

```
plt.plot(momentum_betas['momentum_0_8'], label='beta=0.8', color='red', linestyle='--')

plt.plot(momentum_betas['momentum_0_9'], label='beta=0.9', color='green', linestyle='-.'

plt.title('Gradient Descent with Momentum')
plt.xlabel('Iteration')
plt.ylabel('Error')
plt.legend()
plt.grid(True)
plt.show()
```

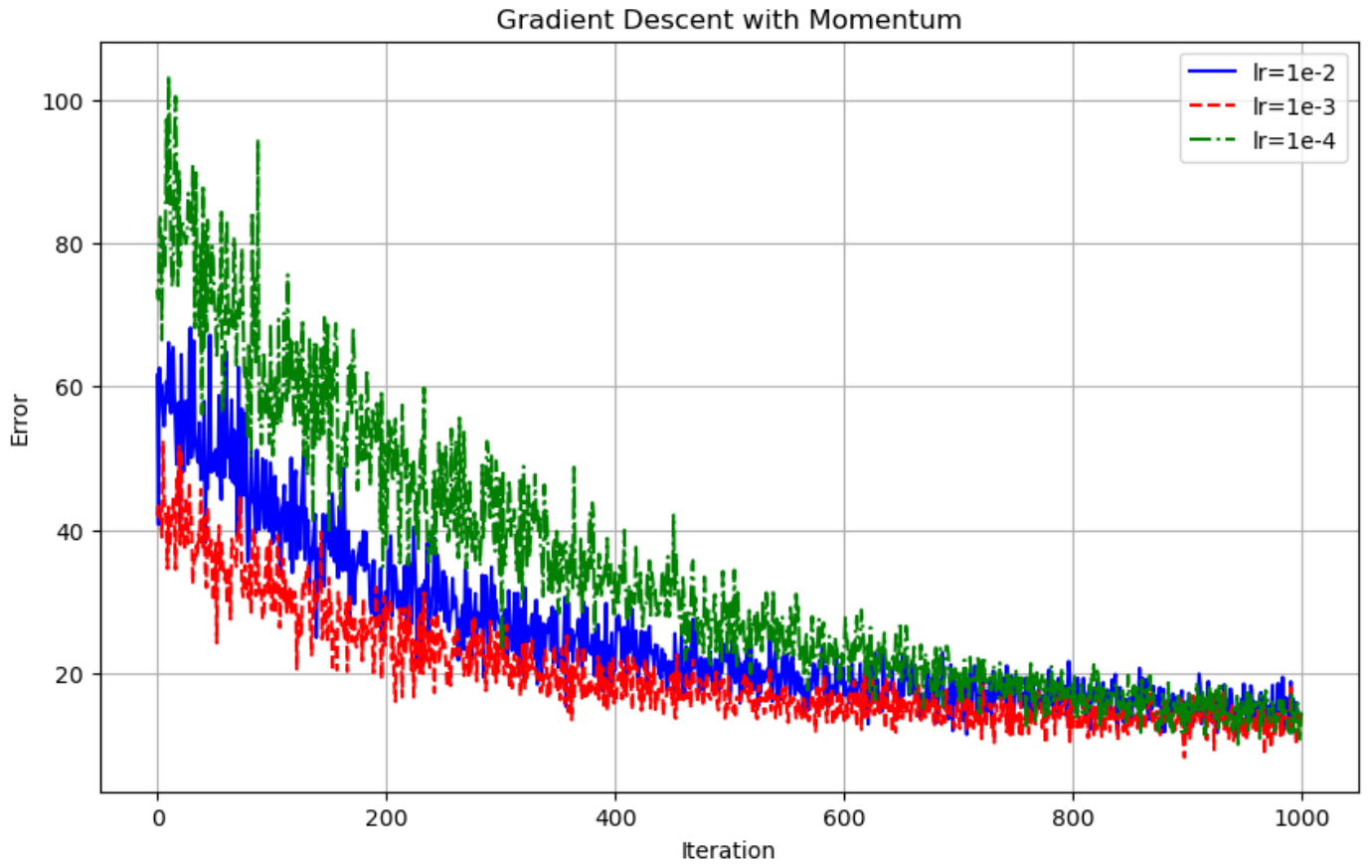


Вывод: проведя подбор параметра β на большой выборке можно утверждать, что эмпирически метод инерции быстрее и точнее сходится для параметра β = 0.7.

In [24]:
```
momentum_lrs = { 'lr_1e-2': [], 'lr_1e-3': [], 'lr_1e-4': [] }

for lr in [1e-2, 1e-3, 1e-4]:
    if lr == 1e-2:
        w_momentum, w_history_momentum, w_error_momentum = stochastic_gradient_descent_w
        momentum_lrs['lr_1e-2'] = w_error_momentum
    elif lr == 1e-3:
        w_momentum, w_history_momentum, w_error_momentum = stochastic_gradient_descent_w
        momentum_lrs['lr_1e-3'] = w_error_momentum
    else:
        w_momentum, w_history_momentum, w_error_momentum = stochastic_gradient_descent_w
        momentum_lrs['lr_1e-4'] = w_error_momentum
```

In [25]:
```
plt.figure(figsize=(10, 6))

plt.plot(momentum_lrs['lr_1e-2'], label='lr=1e-2', color='blue')

plt.plot(momentum_lrs['lr_1e-3'], label='lr=1e-3', color='red', linestyle='--')

plt.plot(momentum_lrs['lr_1e-4'], label='lr=1e-4', color='green', linestyle='-.')
```

```
plt.title('Gradient Descent with Momentum')
plt.xlabel('Iteration')
plt.ylabel('Error')
plt.legend()
plt.grid(True)
plt.show()
```



Вывод: из полученных выше результатов следует, что learning rate = 1e-3 - является наиболее оптимальным с точки зрения сходимости кривой убывания величины ошибки в зависимости от итерации. Поскольку при 1000 итераций величина ошибки приближается к стабильному значению, в отличии от конкурентов.

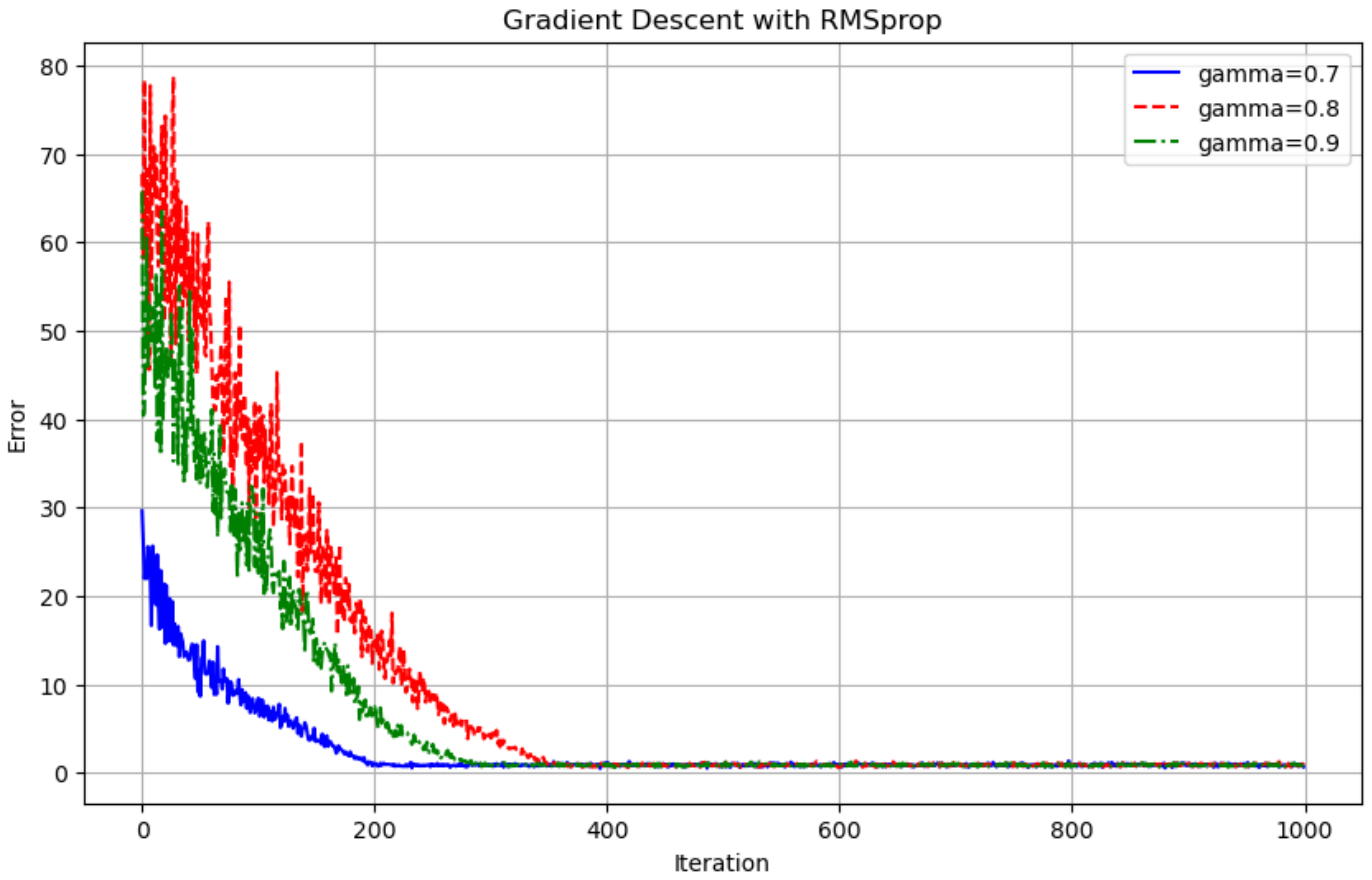
Эксперименты с параметрами для метода RMSprop

У данного метода 3 параметра. Но имеет значения подбирать только 2: величину шага и параметр γ - коэффициент сглаживания.

In [26]:
```
rmsprop_gammas = { 'rmsprop_0_7': [], 'rmsprop_0_8': [], 'rmsprop_0_9': [] }

for gamma in [0.7, 0.8, 0.9]:
    if gamma == 0.7:
        w_momentum, w_history_momentum, w_error_momentum = stochastic_gradient_descent_w
        rmsprop_gammas['rmsprop_0_7'] = w_error_momentum
    elif gamma == 0.8:
        w_momentum, w_history_momentum, w_error_momentum = stochastic_gradient_descent_w
        rmsprop_gammas['rmsprop_0_8'] = w_error_momentum
    else:
        w_momentum, w_history_momentum, w_error_momentum = stochastic_gradient_descent_w
        rmsprop_gammas['rmsprop_0_9'] = w_error_momentum
```

In [27]:
```
plt.figure(figsize=(10, 6))
```

```
plt.plot(rmsprop_gammas['rmsprop_0_7'], label='gamma=0.7', color='blue')

plt.plot(rmsprop_gammas['rmsprop_0_8'], label='gamma=0.8', color='red', linestyle='--')

plt.plot(rmsprop_gammas['rmsprop_0_9'], label='gamma=0.9', color='green', linestyle='-.'

plt.title('Gradient Descent with RMSprop')
plt.xlabel('Iteration')
plt.ylabel('Error')
plt.legend()
plt.grid(True)
plt.show()
```

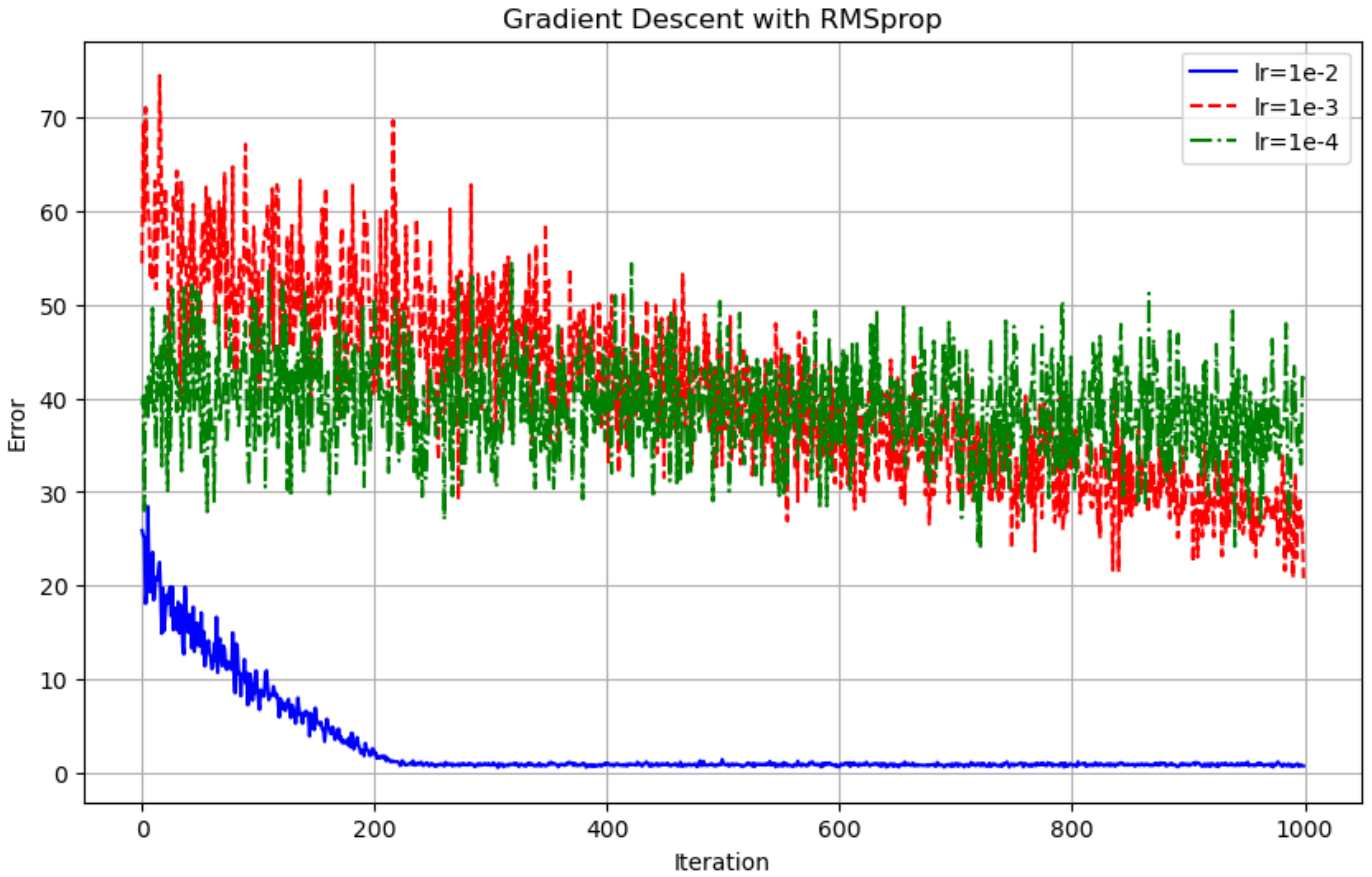


Вывод: проведя подбор параметра γ на большой выборке можно утверждать, что эмпирически RMSprop быстрее и точнее сходится для параметра γ, равного 0.8 - около 200 итераций для этого требуется.

In [28]:
```
rmsprop_lrs = { 'lr_1e-2': [], 'lr_1e-3': [], 'lr_1e-4': [] }

for lr in [1e-2, 1e-3, 1e-4]:
    if lr == 1e-2:
        w_momentum, w_history_momentum, w_error_momentum = stochastic_gradient_descent_w
        rmsprop_lrs['lr_1e-2'] = w_error_momentum
    elif lr == 1e-3:
        w_momentum, w_history_momentum, w_error_momentum = stochastic_gradient_descent_w
        rmsprop_lrs['lr_1e-3'] = w_error_momentum
    else:
        w_momentum, w_history_momentum, w_error_momentum = stochastic_gradient_descent_w
        rmsprop_lrs['lr_1e-4'] = w_error_momentum
```

In [29]:
```
plt.figure(figsize=(10, 6))

plt.plot(rmsprop_lrs['lr_1e-2'], label='lr=1e-2', color='blue')

plt.plot(rmsprop_lrs['lr_1e-3'], label='lr=1e-3', color='red', linestyle='--')
```

```
plt.plot(rmsprop_lrs['lr_1e-4'], label='lr=1e-4', color='green', linestyle='-.')

plt.title('Gradient Descent with RMSprop')
plt.xlabel('Iteration')
plt.ylabel('Error')
plt.legend()
plt.grid(True)
plt.show()
```



Вывод: из полученных выше результатов следует, что learning rate = 1e-2 - является наиболее оптимальным с точки зрения сходимости кривой убывания величины ошибки в зависимости от итерации. Поскольку при 300 итераций величина ошибки приближается к стабильному значению, в отличии от конкурентов: При learning rate = 1e-4 метод вообще не сходится, а для learning rate = 1e-3 требуется более 1000 итераций.

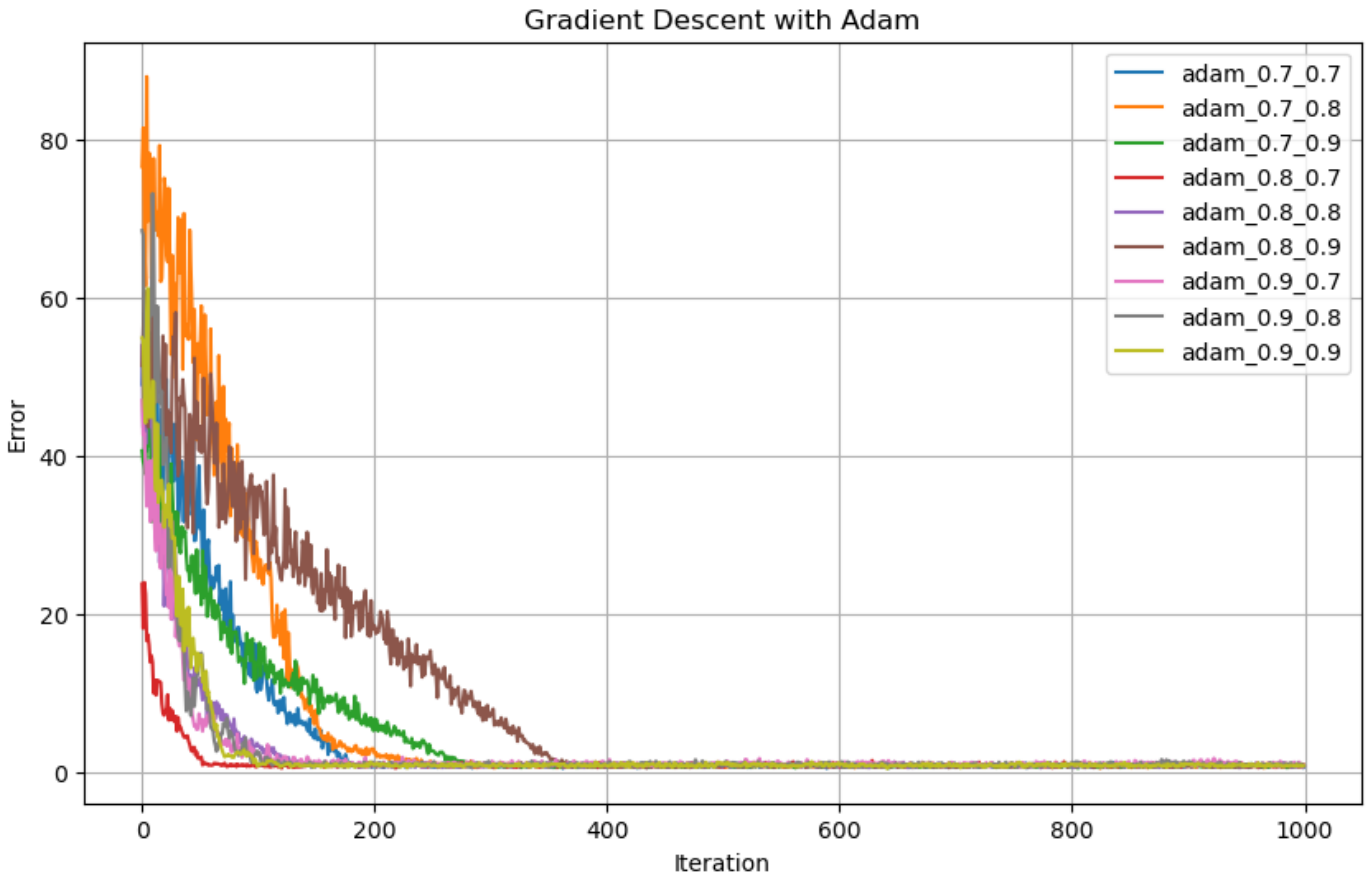
Эксперименты с параметрами для метода Adam

Метод Adam имеет 3 подбираемых параметра: величину шага, параметр β_1 - скорость затухания для импульса и параметр β_2 - скорость затухания для квадратов градиентов. Параметр e подбирать не вижу большого смысла, поскольку он нужен для того, чтобы знаменатель дроби не обращался в 0.

In [30]:
```
momentum_betas = {}

for pairs in list(product([0.7, 0.8, 0.9], [0.7, 0.8, 0.9])):
    beta_1, beta_2 = pairs
    w_momentum, w_history_momentum, w_error_momentum = stochastic_gradient_descent_with_
    momentum_betas[f'adam_{beta_1}_{beta_2}'] = w_error_momentum
```

In [31]:
```
plt.figure(figsize=(10, 6))

for mb in momentum_betas:
```

```
    plt.plot(momentum_betas[mb], label=mb)

plt.title('Gradient Descent with Adam')
plt.xlabel('Iteration')
plt.ylabel('Error')
plt.legend()
plt.grid(True)
plt.show()
```

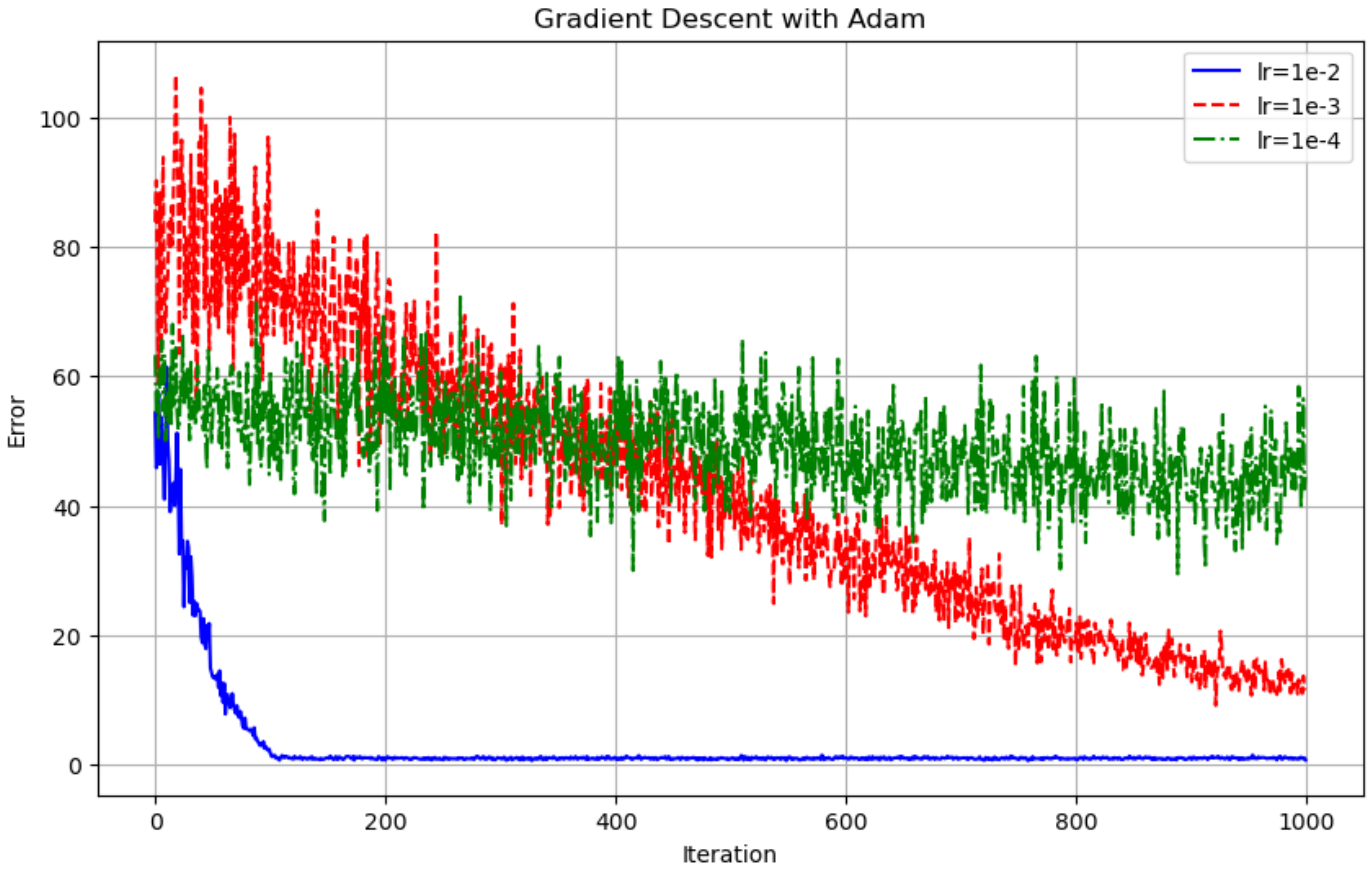


Вывод: проведя подбор параметров β_1 и β_2 на большой выборке можно утверждать, что эмпирически метод Adam быстрее и точнее сходится для параметра $\beta_1 = 0.8$ и $\beta_2 = 0.7$.

In [32]:
```
adam_lrs = { 'lr_1e-2': [], 'lr_1e-3': [], 'lr_1e-4': [] }

for lr in [1e-2, 1e-3, 1e-4]:
    if lr == 1e-2:
        w_momentum, w_history_momentum, w_error_momentum = stochastic_gradient_descent_w
        adam_lrs['lr_1e-2'] = w_error_momentum
    elif lr == 1e-3:
        w_momentum, w_history_momentum, w_error_momentum = stochastic_gradient_descent_w
        adam_lrs['lr_1e-3'] = w_error_momentum
    else:
        w_momentum, w_history_momentum, w_error_momentum = stochastic_gradient_descent_w
        adam_lrs['lr_1e-4'] = w_error_momentum
```

In [33]:
```
plt.figure(figsize=(10, 6))

plt.plot(adam_lrs['lr_1e-2'], label='lr=1e-2', color='blue')

plt.plot(adam_lrs['lr_1e-3'], label='lr=1e-3', color='red', linestyle='--')

plt.plot(adam_lrs['lr_1e-4'], label='lr=1e-4', color='green', linestyle='-.')

plt.title('Gradient Descent with Adam')
plt.xlabel('Iteration')
```

```
plt.ylabel('Error')
plt.legend()
plt.grid(True)
plt.show()
```



Вывод: из полученных выше результатов следует, что learning rate = 1e-2 - является наиболее оптимальным с точки зрения сходимости кривой убывания величины ошибки в зависимости от итерации. Поскольку при 100 итерациях величина ошибки приближается к стабильному значению - 0, в отличии от конкурентов: При learning rate = 1e-4 метод вообще не сходится, а для learning rate = 1e-3 требуется более 1000 итераций.