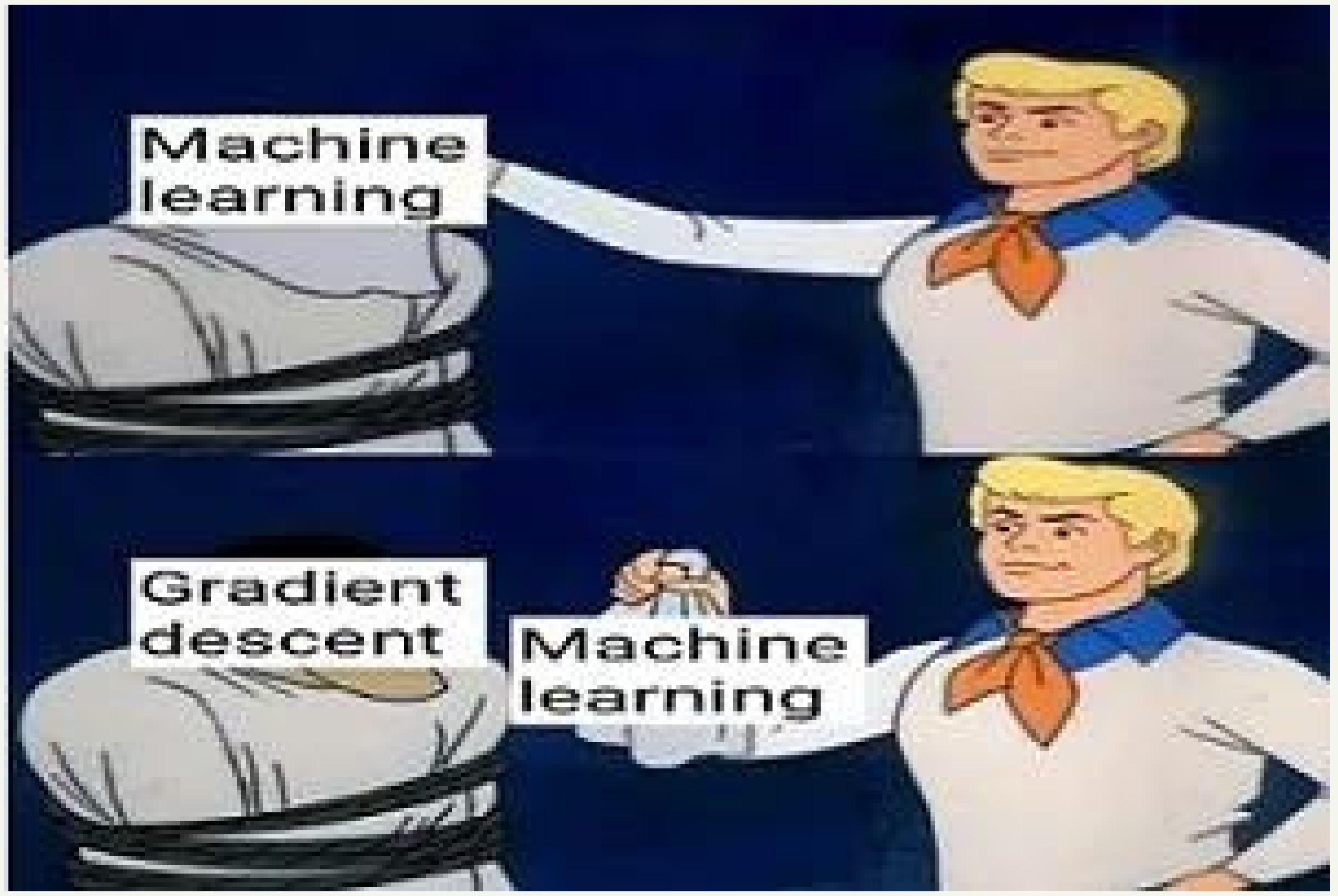
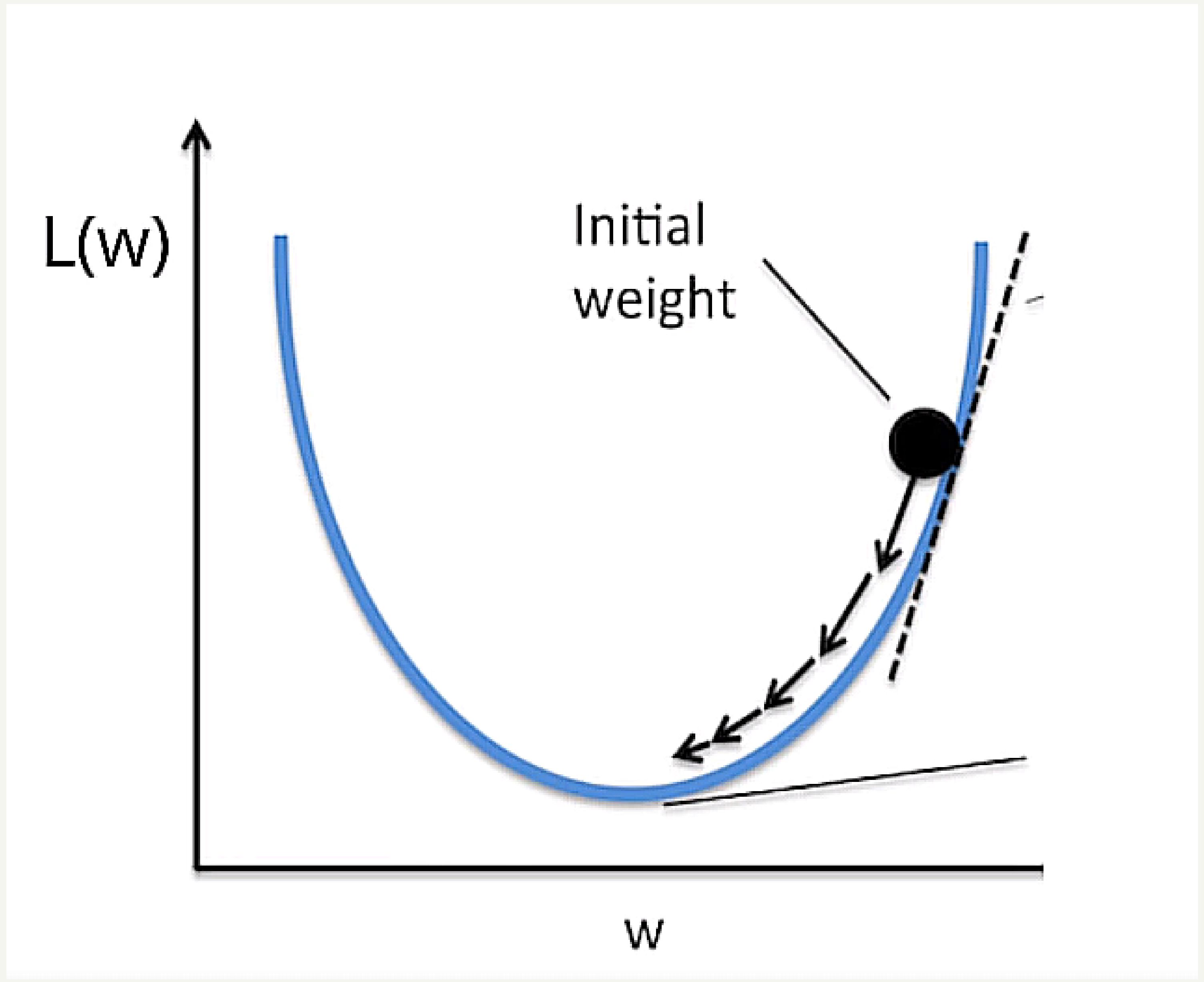
Deep Learning -Otimizers

A project presented by **Dhruv Yadav**





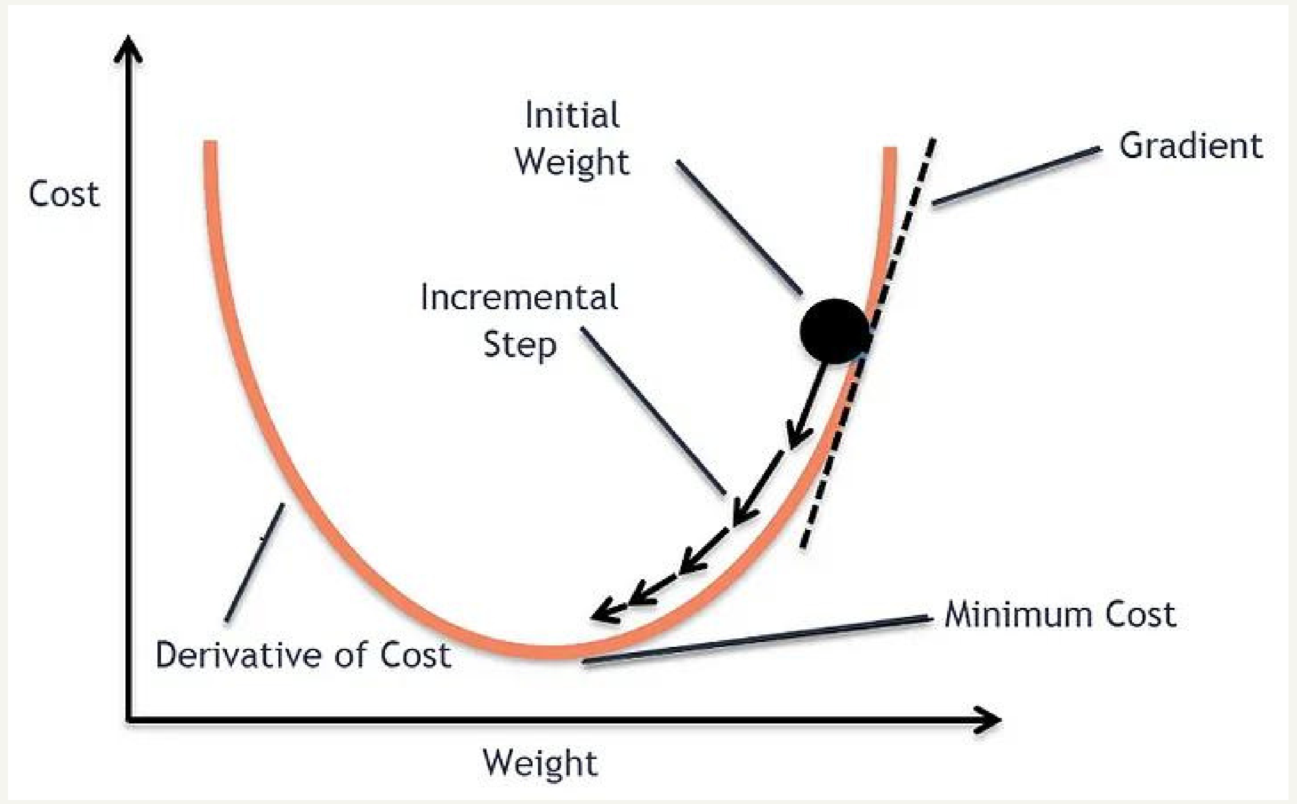
Introduction

Gradient Descent

Why we need Gradient Descent?

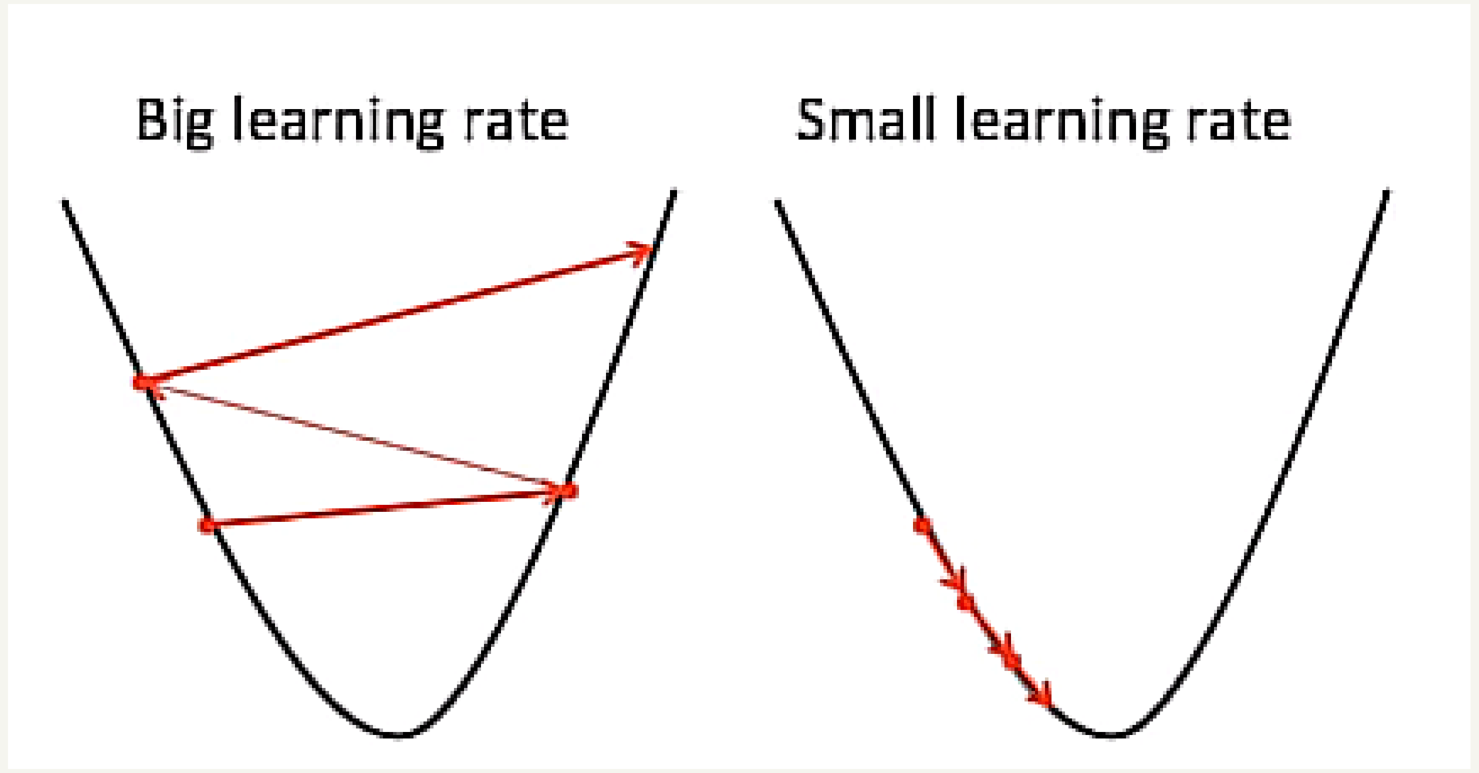
Basis of all optimizers

- Whenever we have a function which we want to minimize or maximize, we take its derivative and set it to 0.
- This works well for simpler functions but fails for more complex ones.
- **Idea-** repeatedly take small steps in the direction of the gradient to find a new w(weight).
- At each step, L(w) decreases provided the step size is small enough; eventually converging to the minimum.



$$w = \text{random value}$$
$$w = w - \eta \nabla L(w)$$

Hyperparameters



Learning Rate

Learning rate must be small enough so that cost does not blow up but large enough so we don't have to wait for longer than necessary.

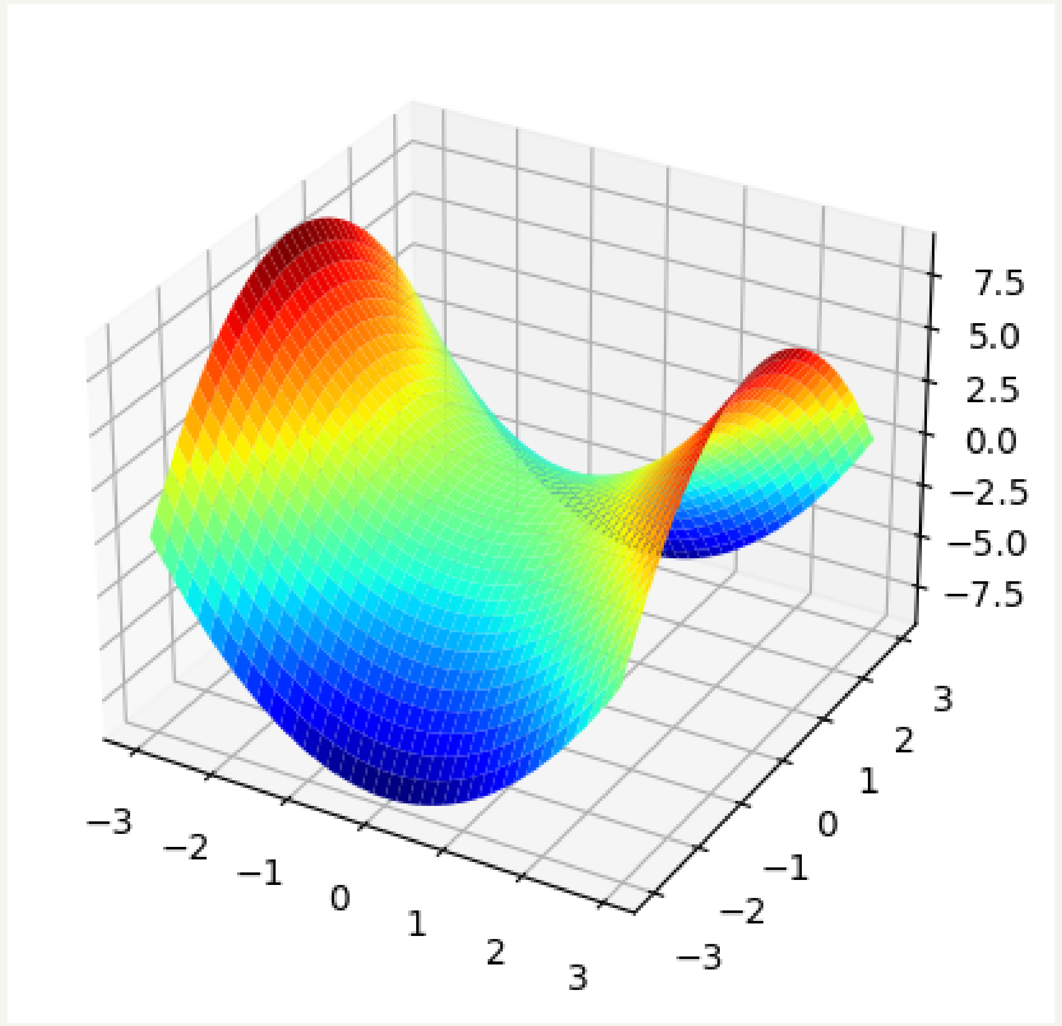
Multivariate Gradient Descent

Code Demonstration

Intuition

The bivariate (two variables) quadratic function, $J(\theta 1,\theta 2)$ which we are going to perform-

$$J(\theta_1, \theta_2) = \theta_1^2 - \theta_2^2$$



When applying gradient descent to this function, our objective still remains the same, except that now we have two parameters, θ1 and θ2 to optimise.

Update Rule

Rule for single variate

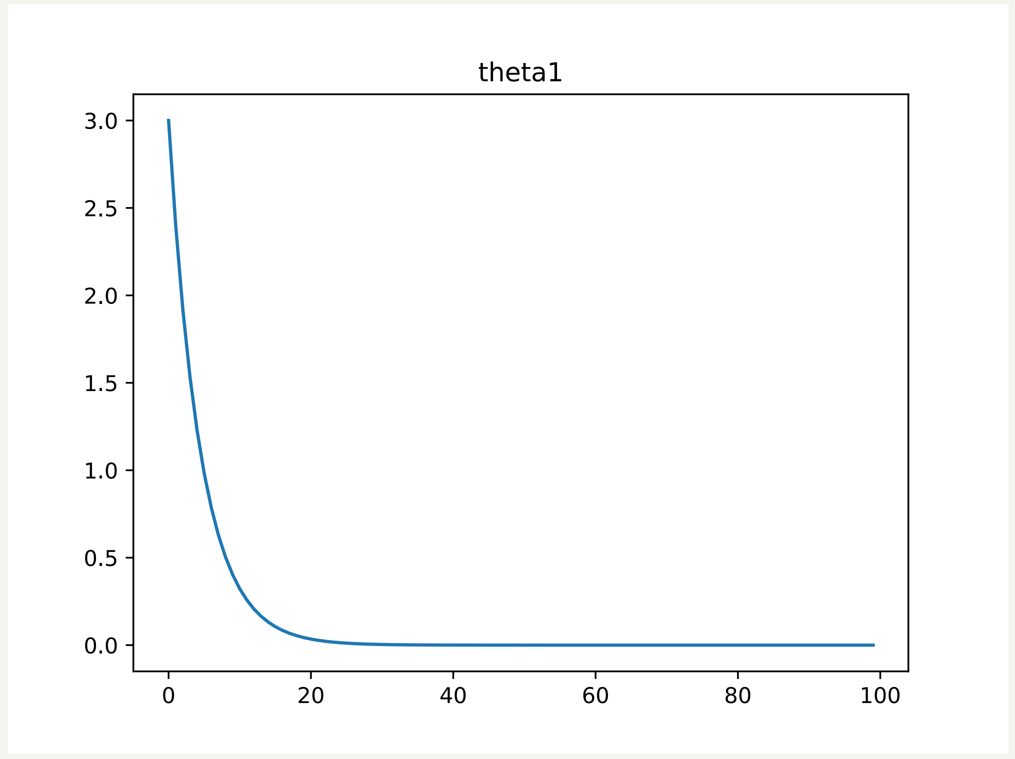
This was the update rule for univariate gradient descent:

$$\theta := \theta - \alpha \frac{dJ(\theta)}{d\theta}$$

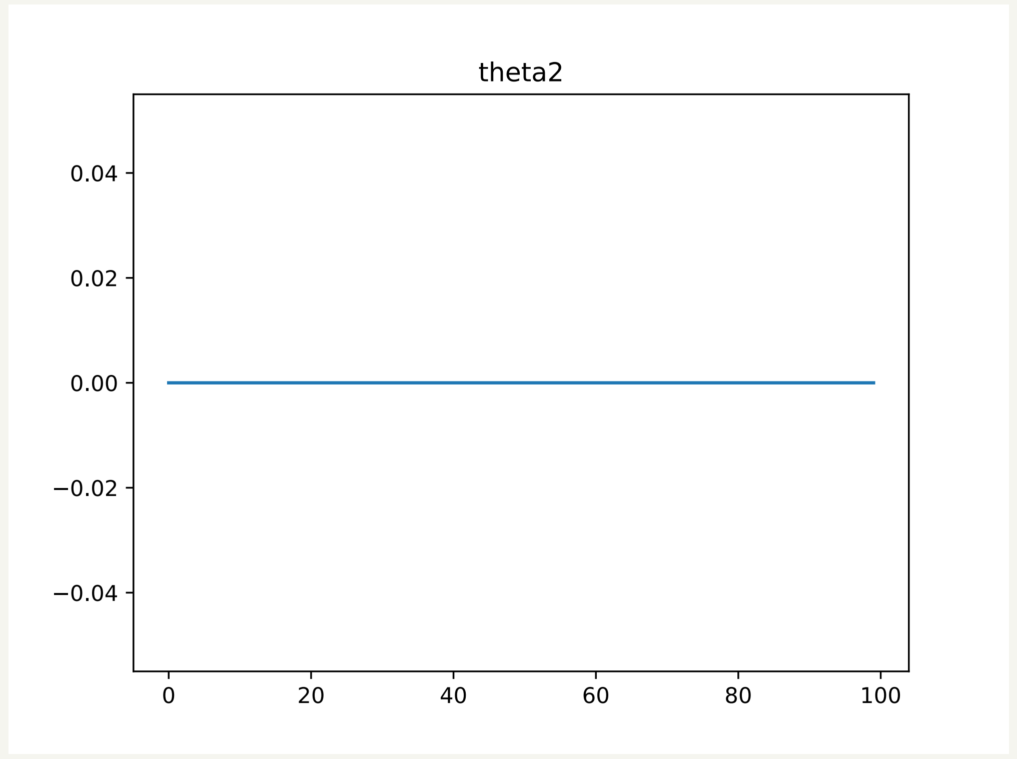
Where α is the learning rate and $dJ(\theta)/d\theta$ is the derivative of $J(\theta)$ — i.e. the slope of a tangent line that touches the $J(\theta)$ at given θ.

Now that we have two variables, we need to supply an update rule for each:

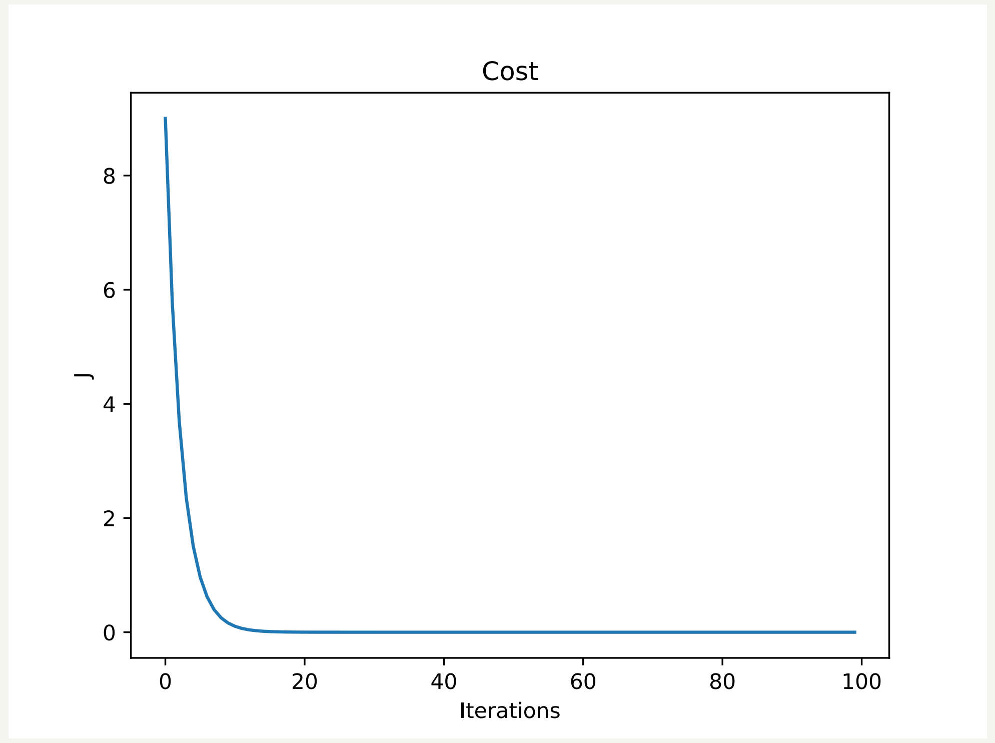
$$\theta_1 := \theta_1 - \alpha \frac{\partial J(\theta_1, \theta_2)}{\partial \theta_1}$$
$$\theta_2 := \theta_2 - \alpha \frac{\partial J(\theta_1, \theta_2)}{\partial \theta_2}$$



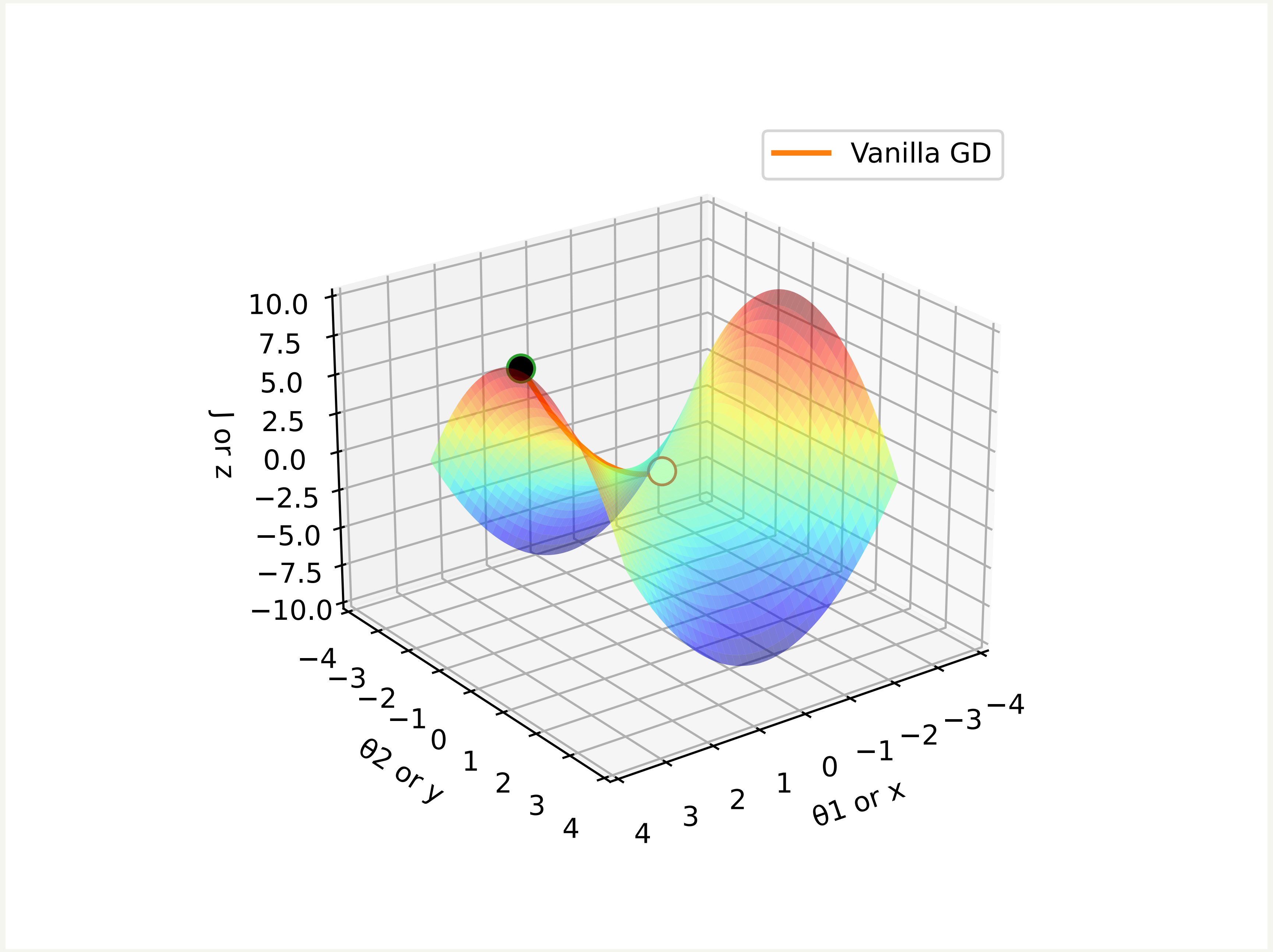
theta1



theta2

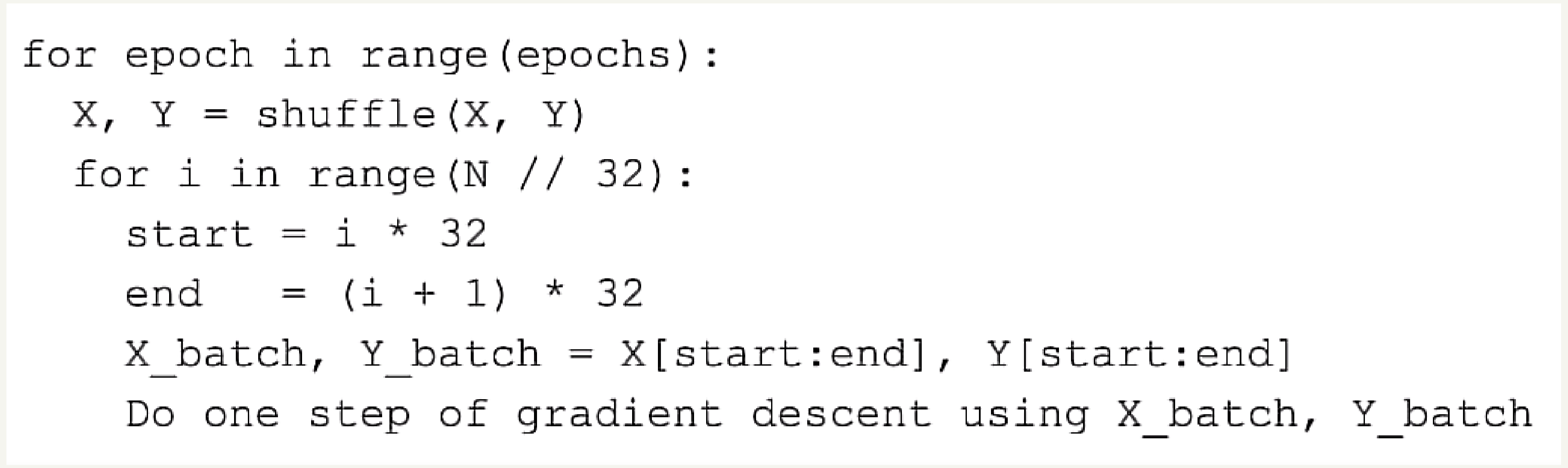


J(cost)


Vanilla GD
J or z
10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
θ2 or y
θ1 or x
−4 −3 −2 −1 0 1 2 3 4

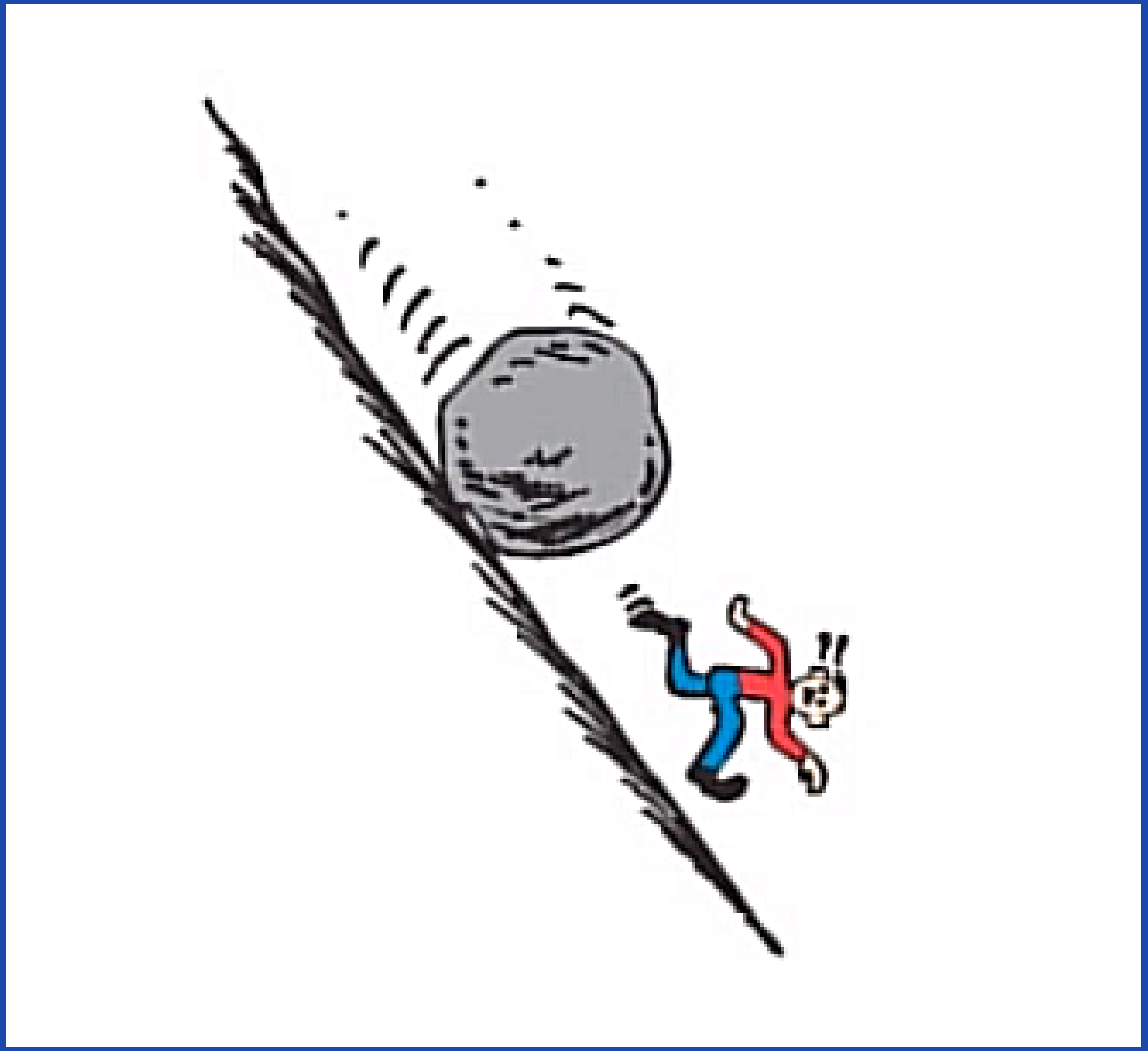
Stochastic Gradient Descent

How does it work?



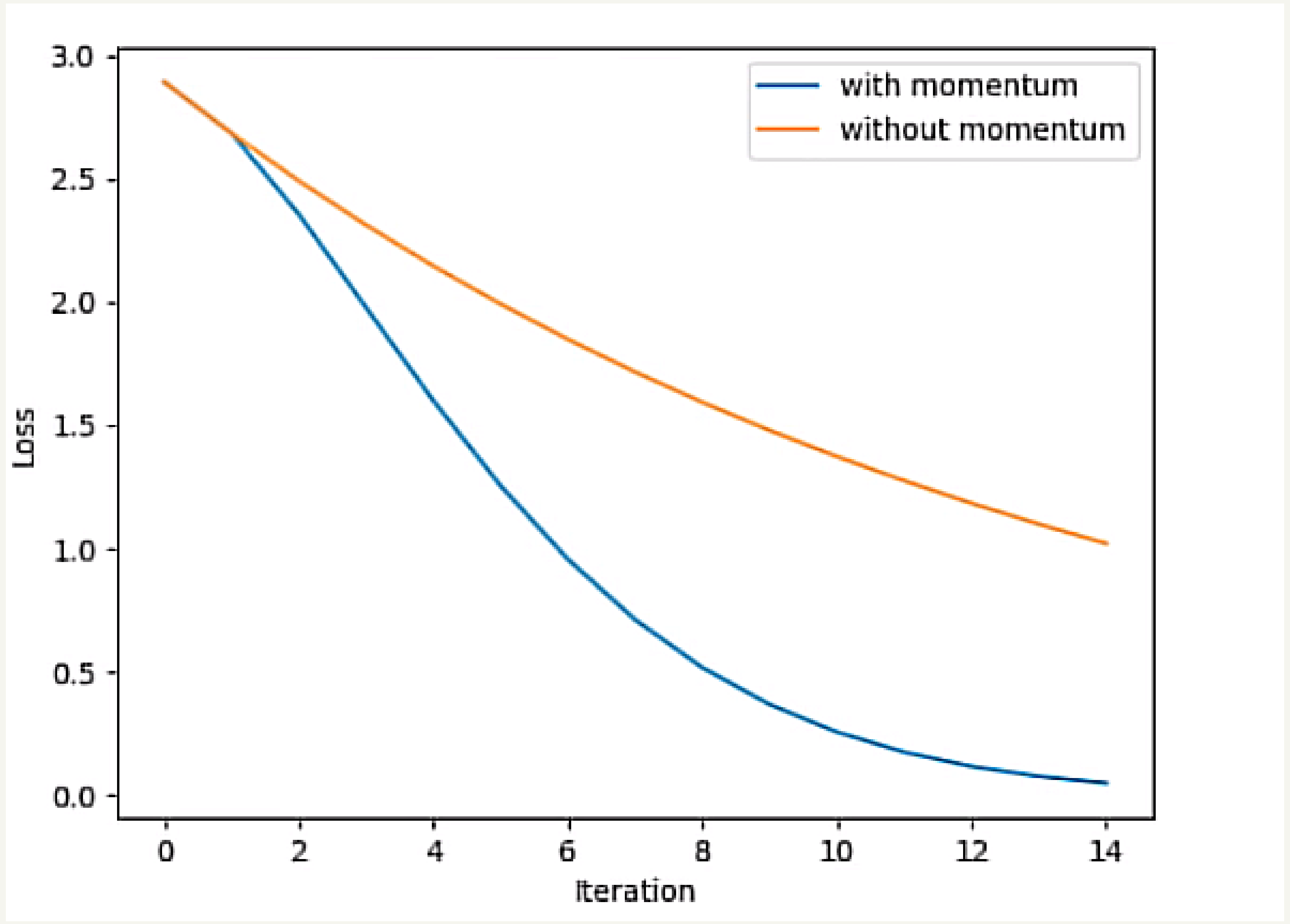
```
for epoch in range(epochs):
  X, Y = shuffle(X, Y)
  for i in range(N // 32):
    start = i * 32
    end   = (i + 1) * 32
    X_batch, Y_batch = X[start:end], Y[start:end]
    Do one step of gradient descent using X_batch, Y_batch
```

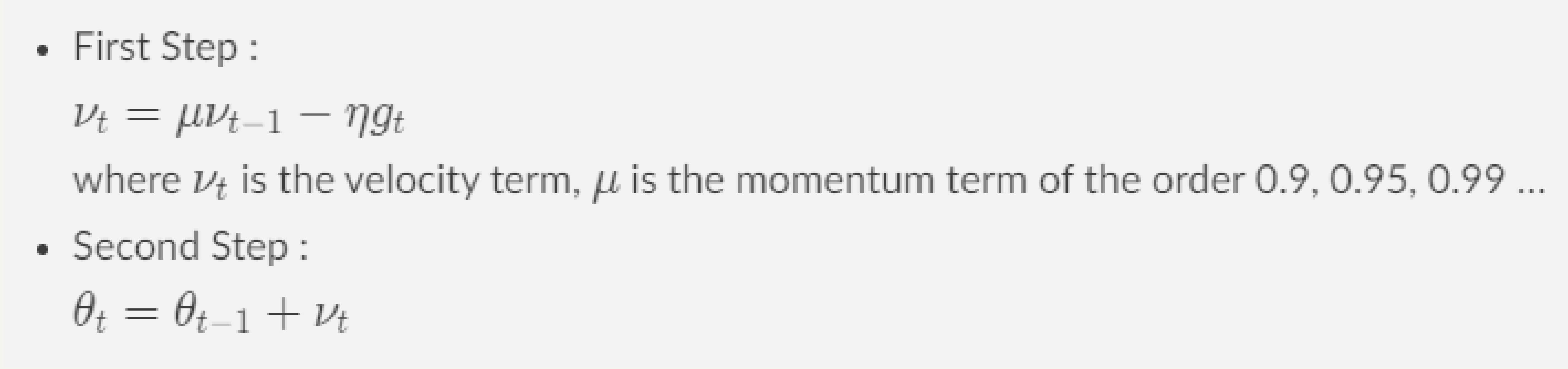
- SGD works by updating the model's parameters based on the gradients of the cost function computed on a small subset of the training data, called a "mini-batch". This makes the algorithm much faster than traditional gradient descent, which computes the gradients on the entire training set.
- One advantage of SGD is that it is more robust to noisy or non-convex cost functions. By randomly sampling mini-batches, the algorithm can escape local minima and converge to a global minimum.
- Another advantage of SGD is that it allows for online learning, which means the model can be updated in real-time as new data becomes available.



Momentum

- Momentum adds a "velocity" term to the gradient descent update rule, which allows the algorithm to "smooth out" the search path and move more quickly in directions where the gradient is consistently pointing.
- **Advantages**: Momentum gradient descent has faster convergence, greater stability in the face of noise or complex landscapes, and reduced sensitivity to hyperparameters.
- **Limitations**: It can sometimes overshoot the minimum or get stuck in suboptimal local minima.





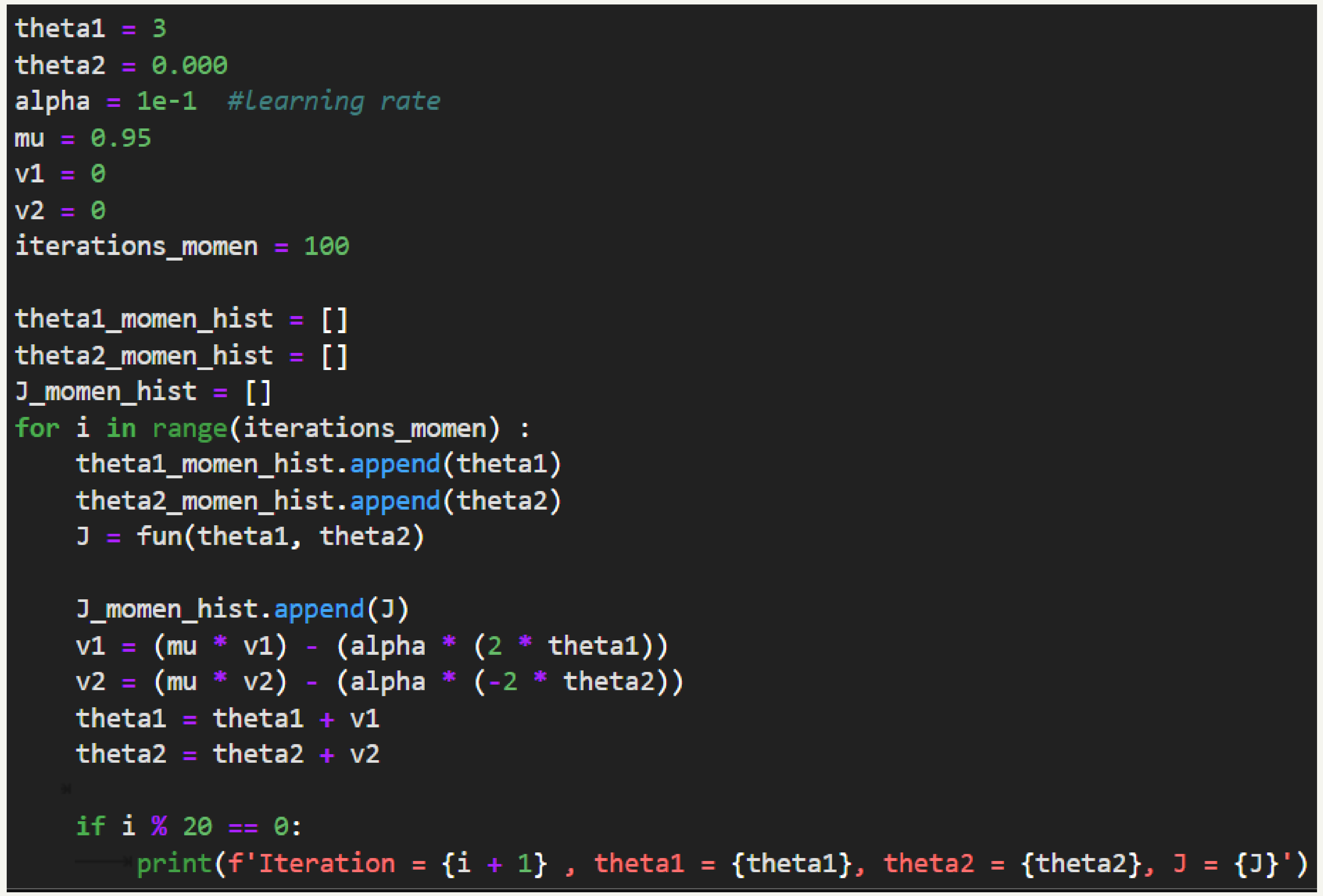
- First Step :

  $$\nu_t = \mu\nu_{t-1} - \eta g_t$$

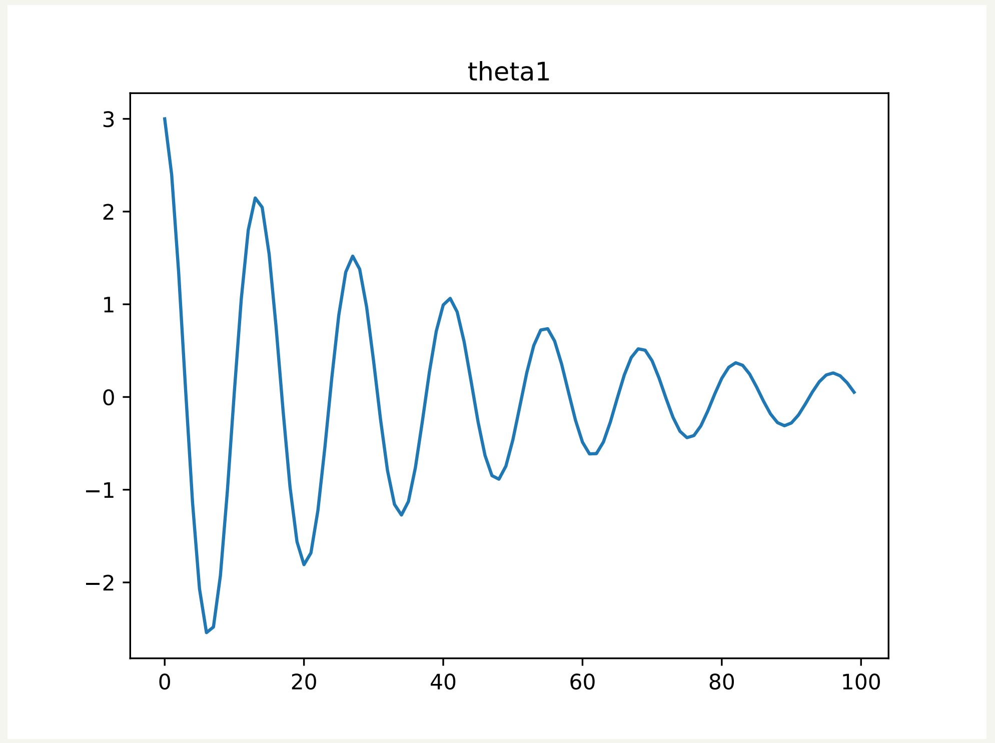
  where ν_t is the velocity term, μ is the momentum term of the order 0.9, 0.95, 0.99 ...
- Second Step :

  $$\theta_t = \theta_{t-1} + \nu_t$$

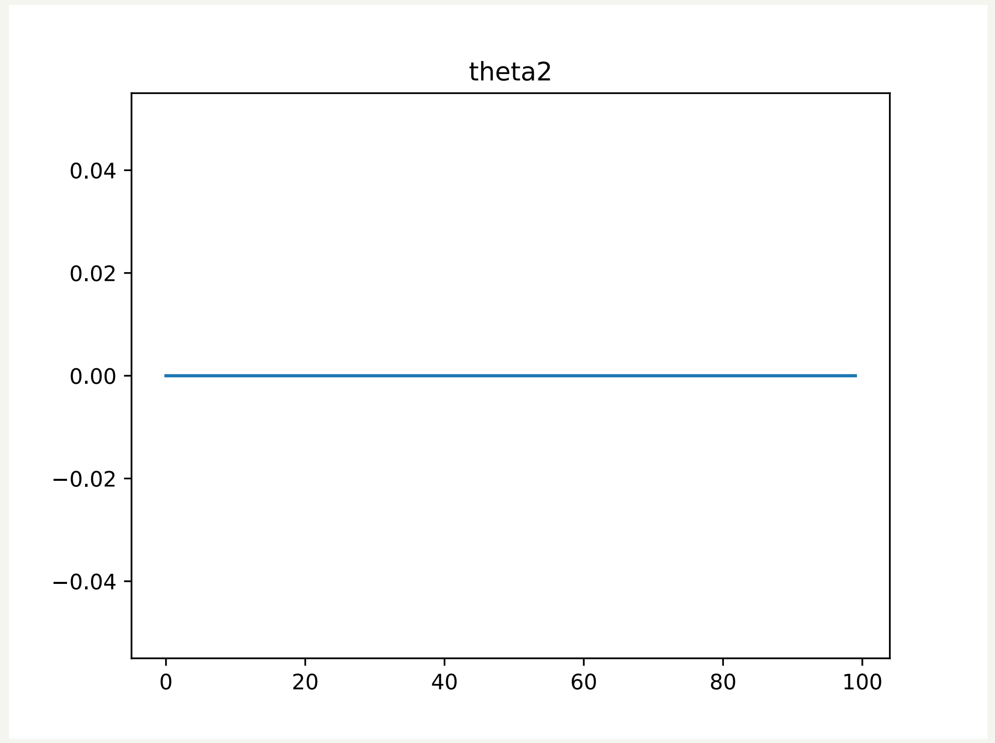
Code



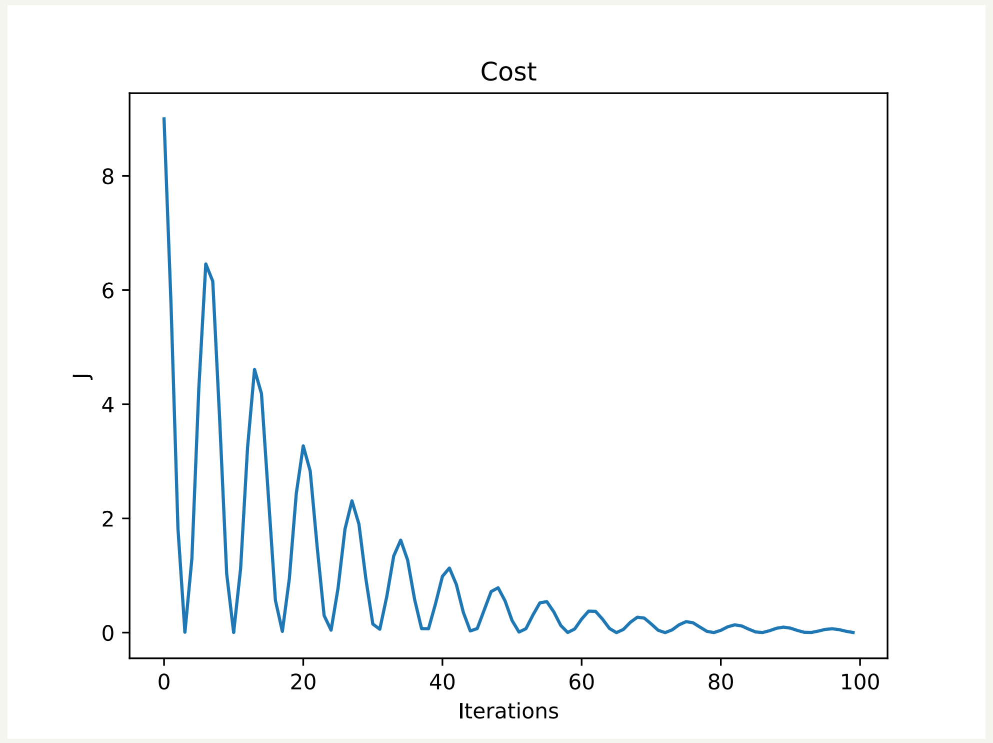
```python
theta1 = 3
theta2 = 0.000
alpha = 1e-1  #learning rate
mu = 0.95
v1 = 0
v2 = 0
iterations_momen = 100

theta1_momen_hist = []
theta2_momen_hist = []
J_momen_hist = []
for i in range(iterations_momen) :
    theta1_momen_hist.append(theta1)
    theta2_momen_hist.append(theta2)
    J = fun(theta1, theta2)

    J_momen_hist.append(J)
    v1 = (mu * v1) - (alpha * (2 * theta1))
    v2 = (mu * v2) - (alpha * (-2 * theta2))
    theta1 = theta1 + v1
    theta2 = theta2 + v2

    if i % 20 == 0:
        print(f'Iteration = {i + 1} , theta1 = {theta1}, theta2 = {theta2}, J = {J}')
```



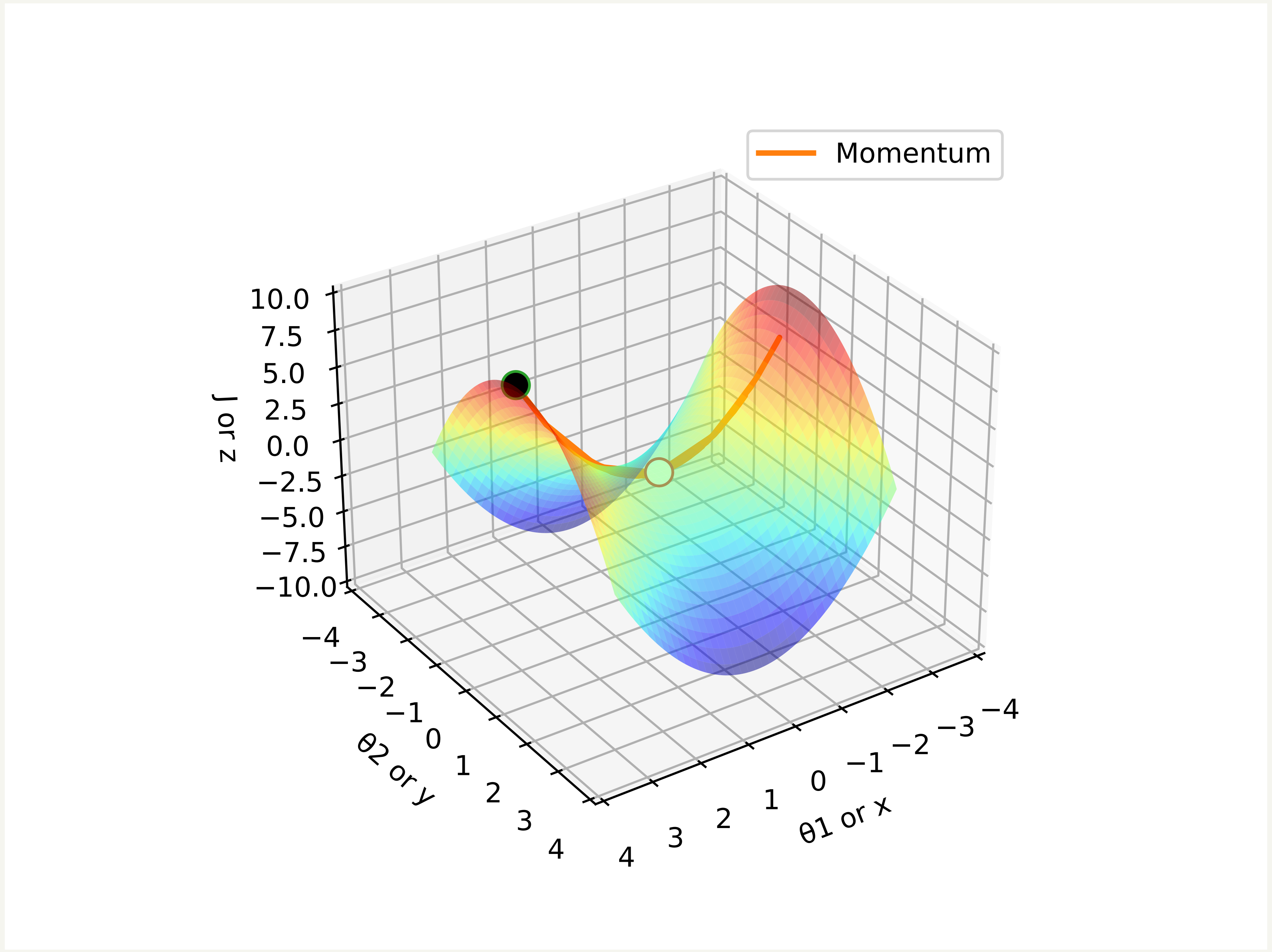
theta1



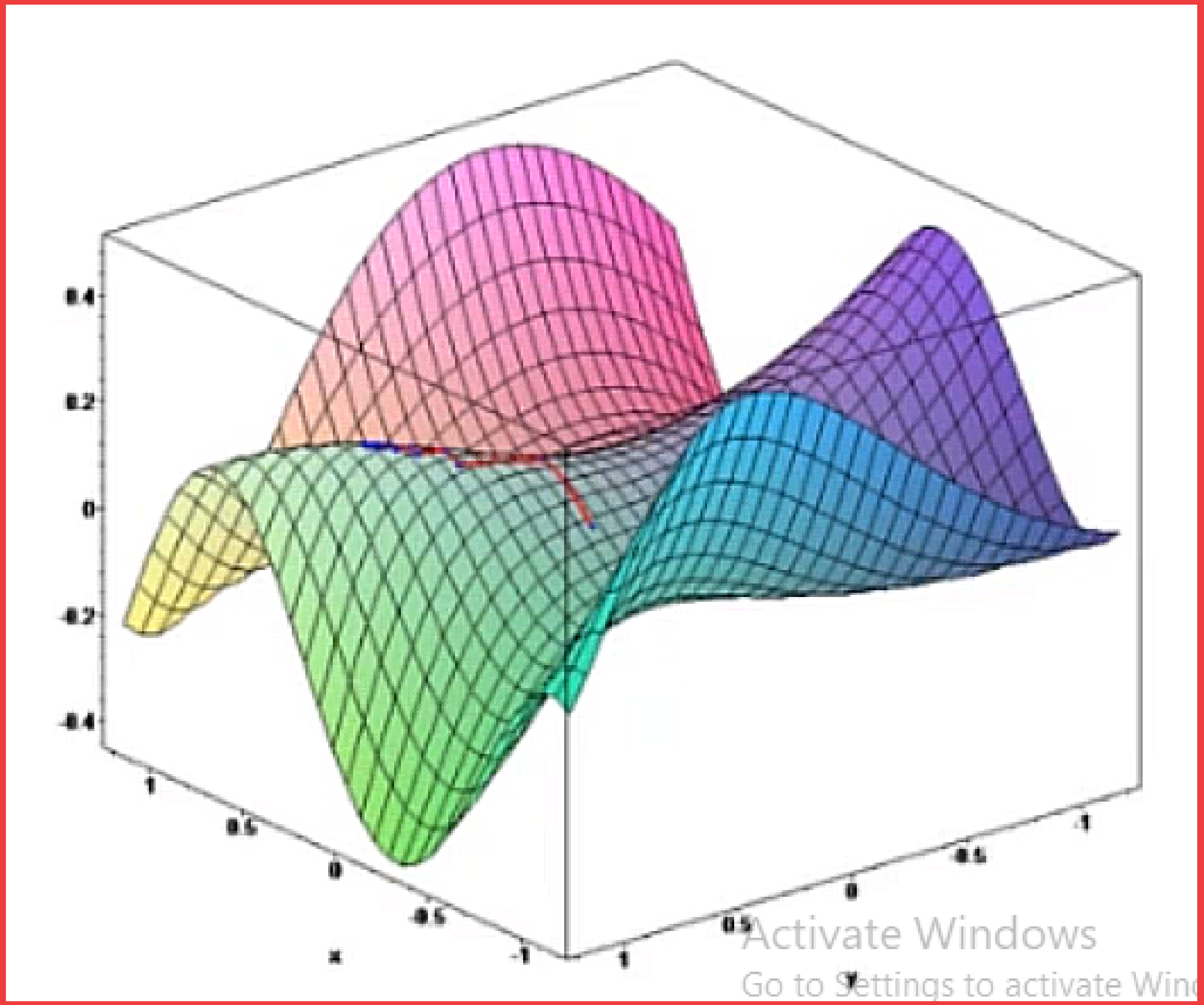
theta2

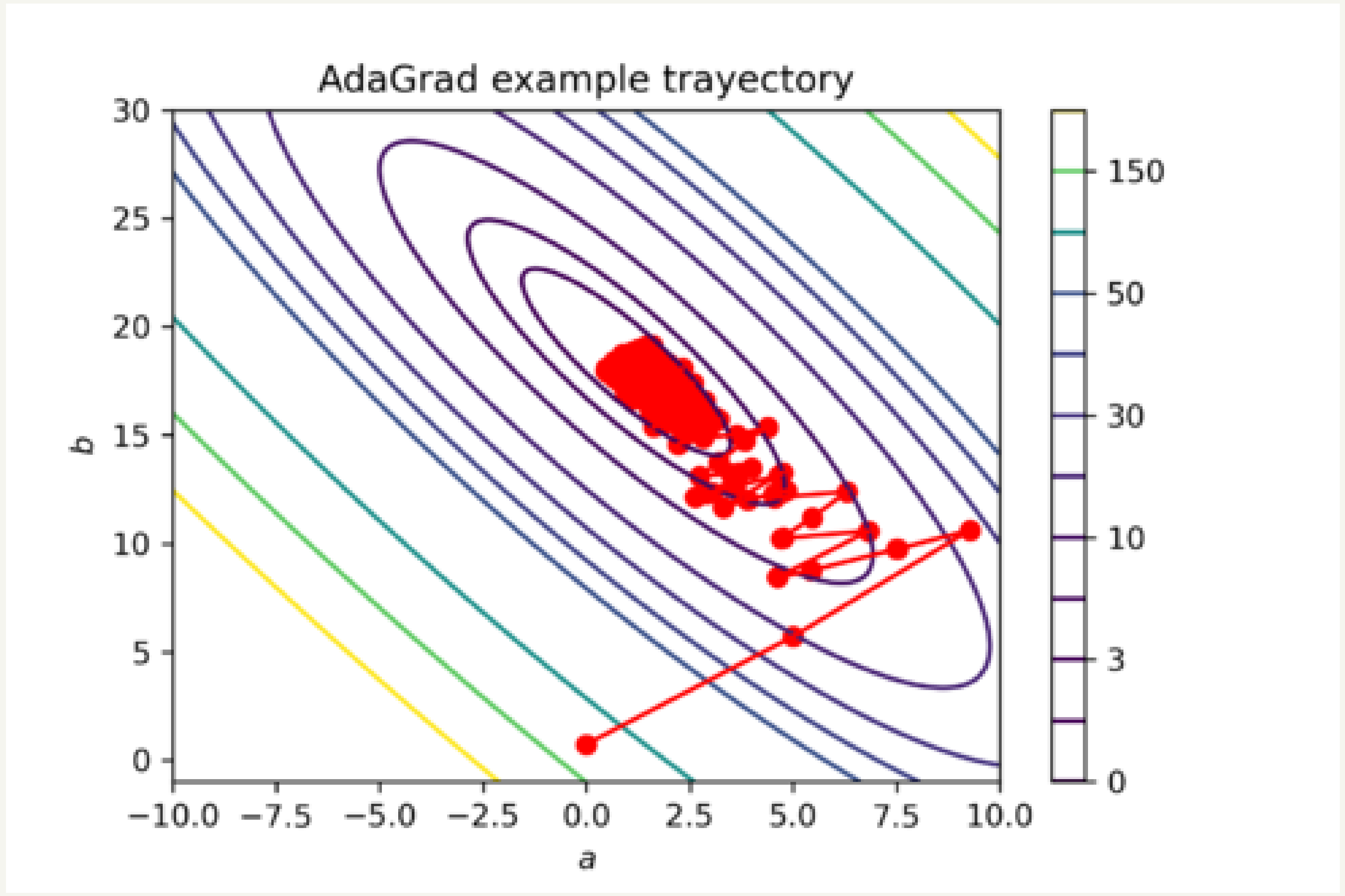


J(cost)


Momentum
10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
J or z
−4
−3
−2
−1
0
1
2
3
4
θ2 or y
4
3
2
1
0
−1
−2
−3
−4
θ1 or x

AdaGrad





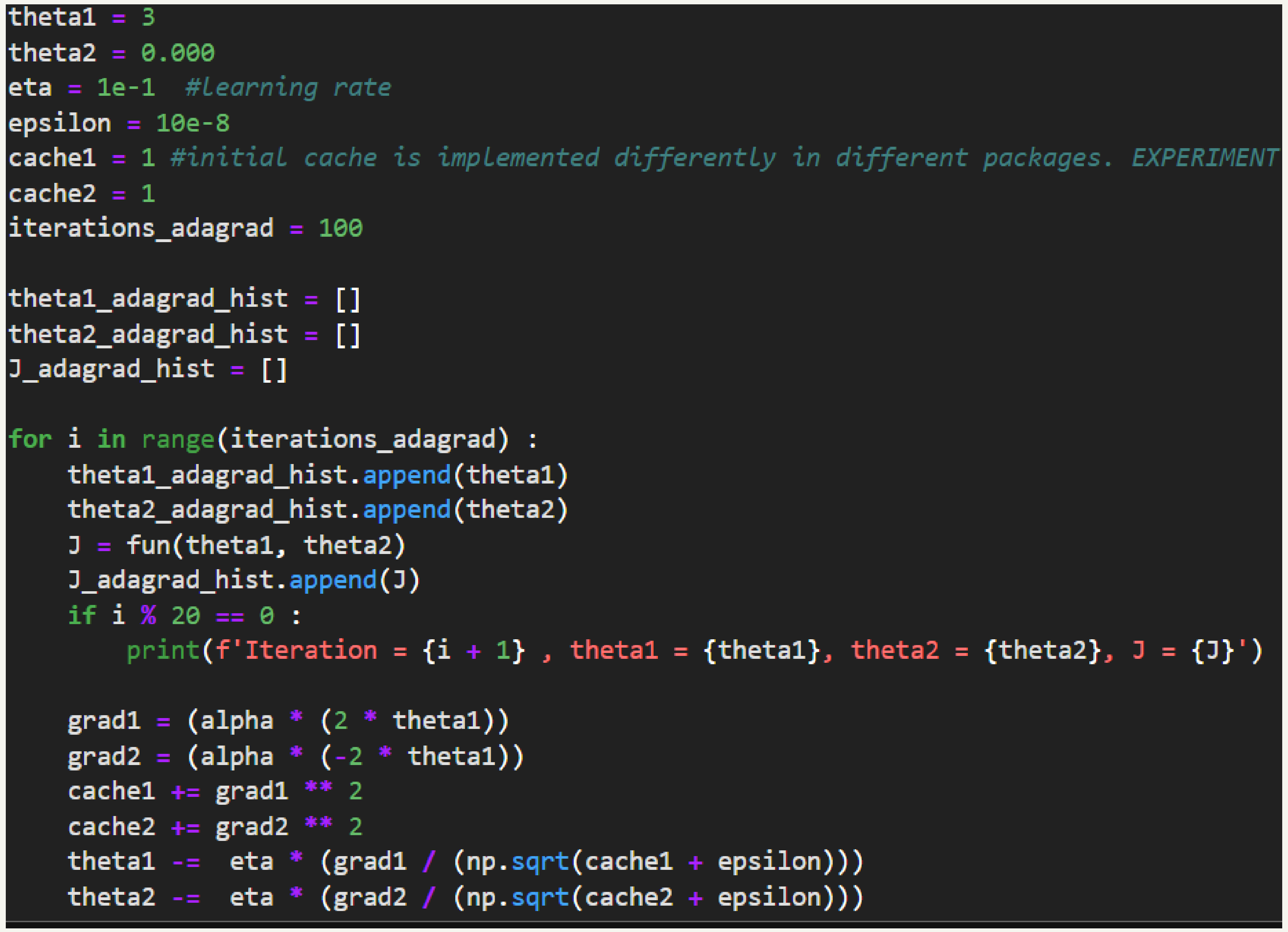
- Modified version of gradient descent that adapts the learning rate to each parameter based on its history. This means that it reduces the learning rate for frequently occurring features, and increases it for infrequently occurring features.
- **Advantages**: Adagrad can handle sparse data more efficiently than other optimization techniques, and can adapt to the data more effectively.
- **Limitations**: Its sensitivity to initial learning rate can cause it to converge too quickly or too slowly.

$$cache = cache + gradient^2$$

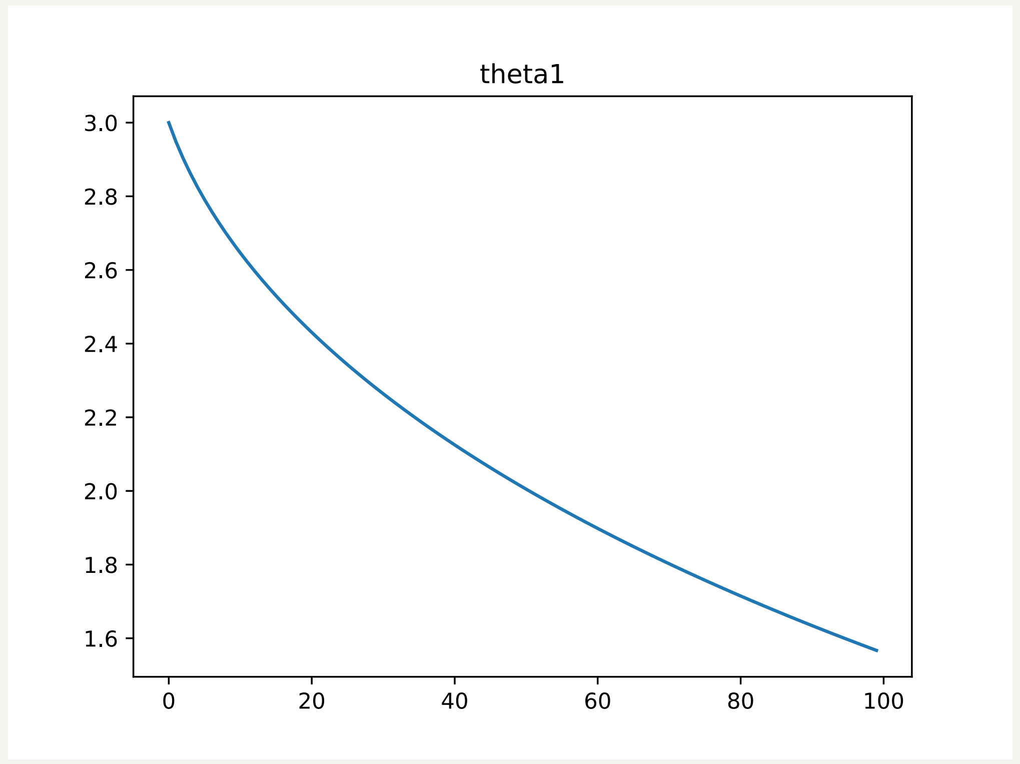
$$\theta \leftarrow \theta - \eta \frac{\nabla J}{\sqrt{cache + \varepsilon}}$$

- Each parameter has its own cache.
- Cache accumulates squared gradients, hence always positive.
- To avoid dividing by zero in second eqn, we add a small number ϵ of the order 10^(-8) or 10^(-10).

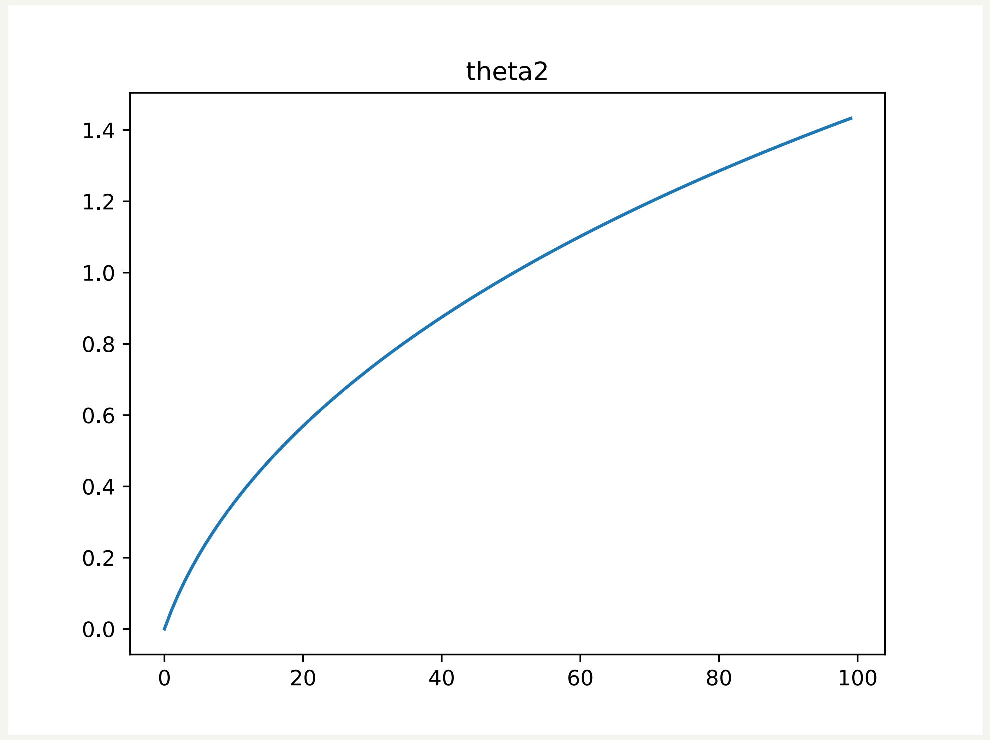
Code



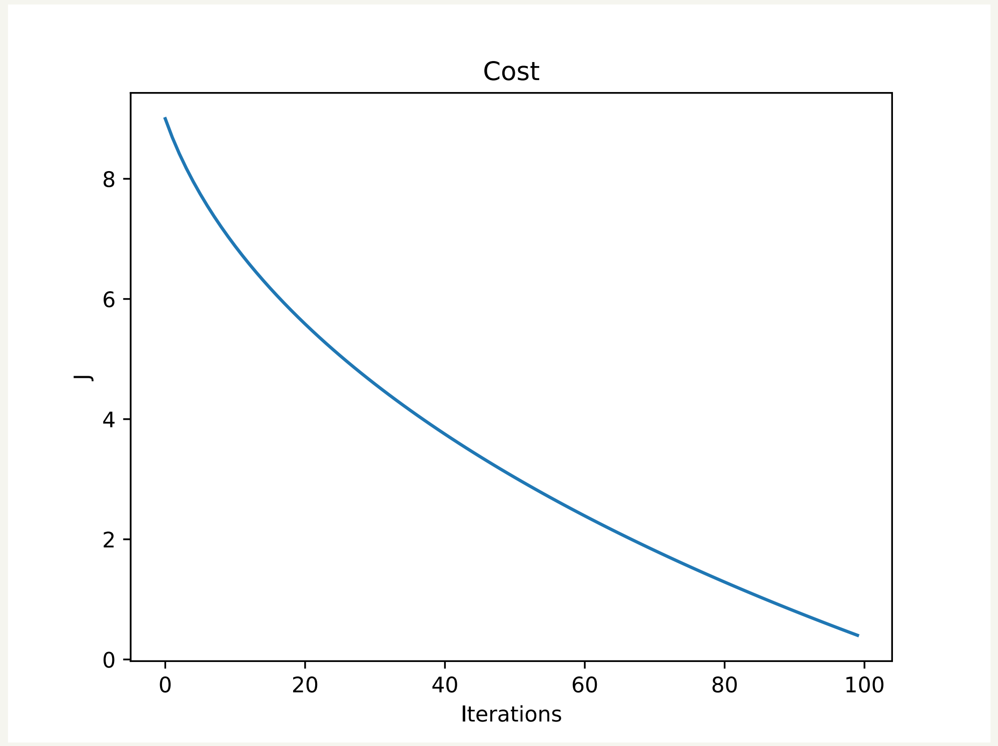
```python
theta1 = 3
theta2 = 0.000
eta = 1e-1  #learning rate
epsilon = 10e-8
cache1 = 1 #initial cache is implemented differently in different packages. EXPERIMENT
cache2 = 1
iterations_adagrad = 100

theta1_adagrad_hist = []
theta2_adagrad_hist = []
J_adagrad_hist = []

for i in range(iterations_adagrad) :
    theta1_adagrad_hist.append(theta1)
    theta2_adagrad_hist.append(theta2)
    J = fun(theta1, theta2)
    J_adagrad_hist.append(J)
    if i % 20 == 0 :
        print(f'Iteration = {i + 1} , theta1 = {theta1}, theta2 = {theta2}, J = {J}')

    grad1 = (alpha * (2 * theta1))
    grad2 = (alpha * (-2 * theta1))
    cache1 += grad1 ** 2
    cache2 += grad2 ** 2
    theta1 -=  eta * (grad1 / (np.sqrt(cache1 + epsilon)))
    theta2 -=  eta * (grad2 / (np.sqrt(cache2 + epsilon)))
```



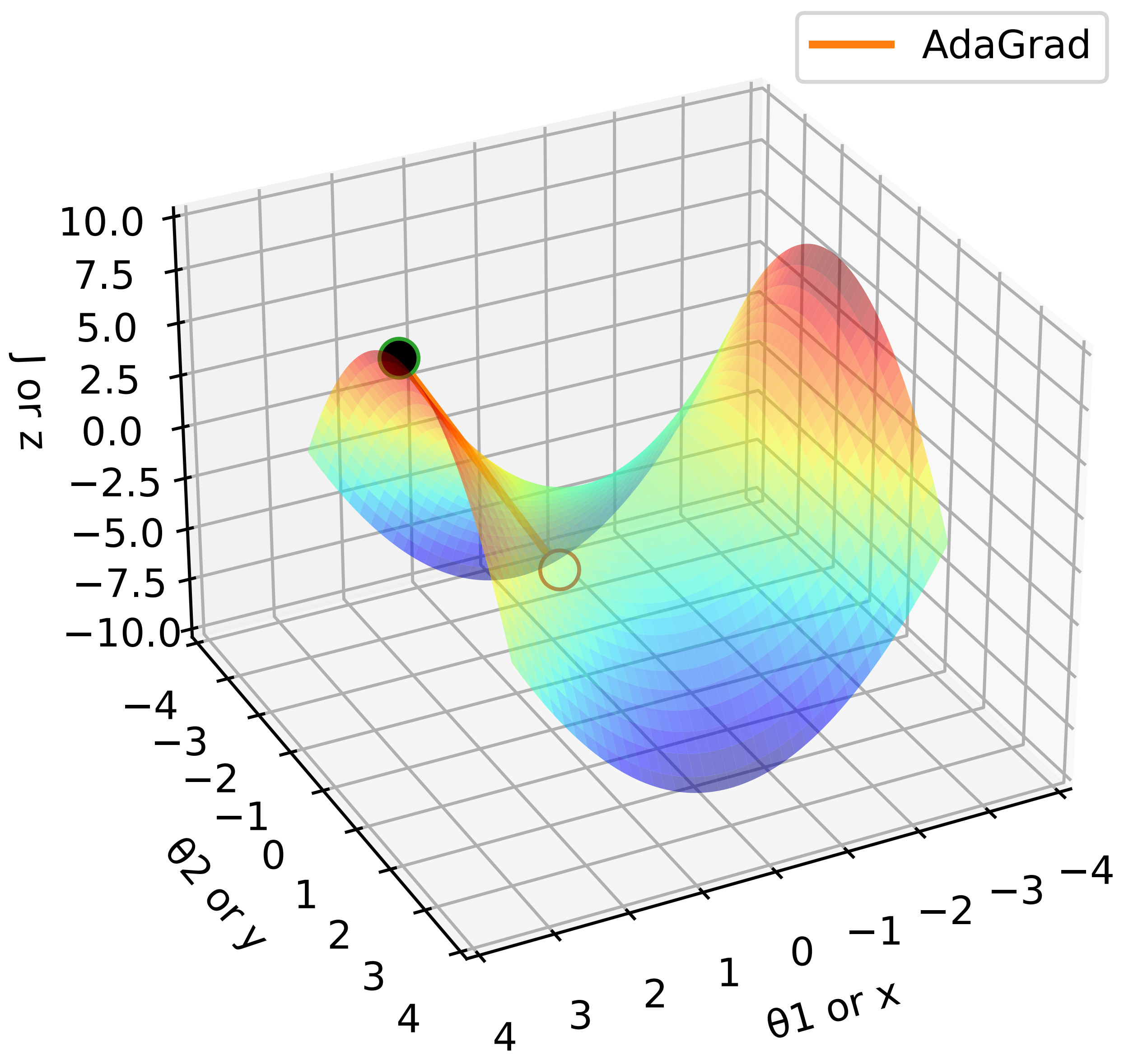
theta1



theta2



J(cost)


AdaGrad
J or z
10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
θ2 or y
−4
−3
−2
−1
0
1
2
3
4
θ1 or x
4
3
2
1
0
−1
−2
−3
−4

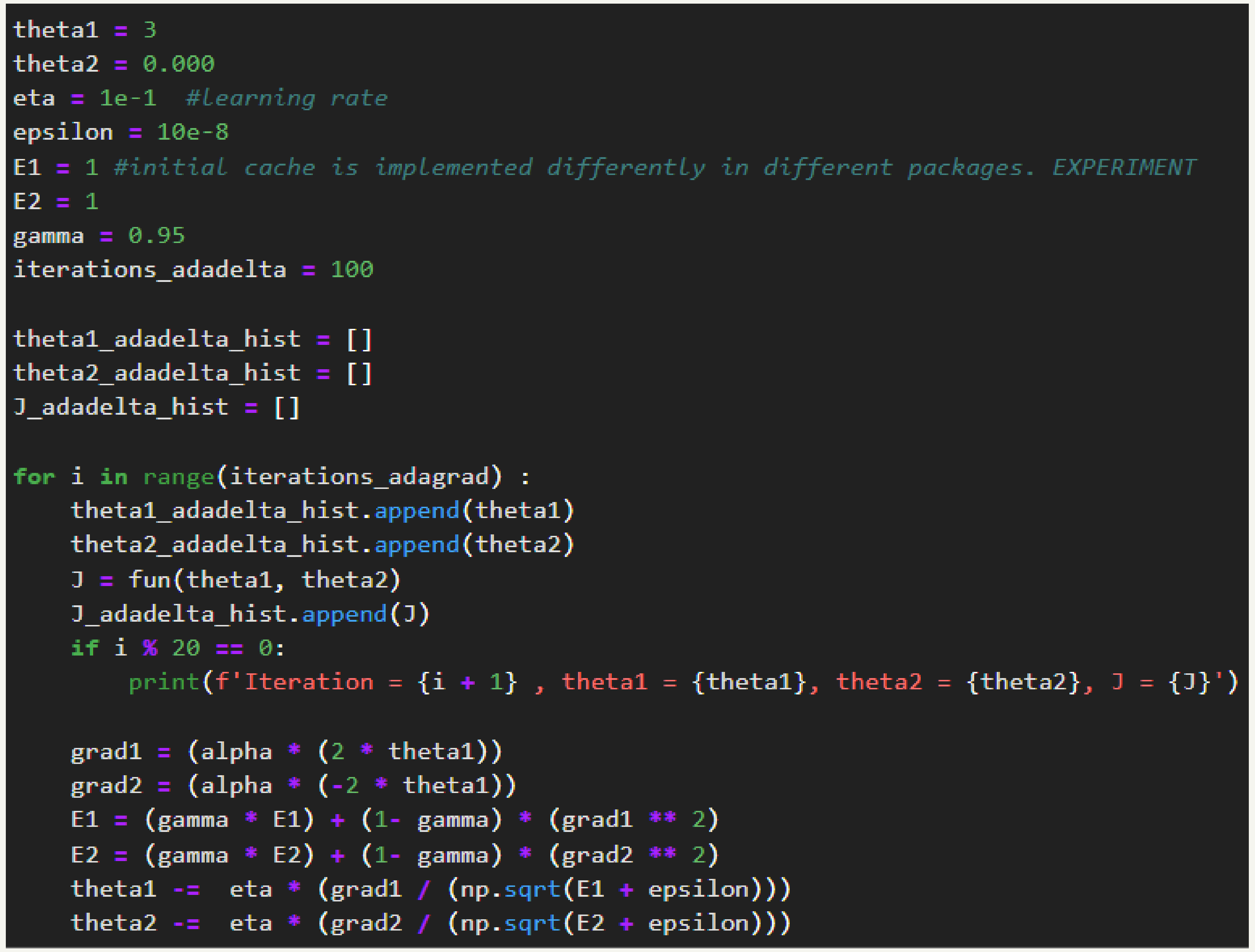
AdaDelta

- Modified version of gradient descent that adapts the learning rate to each parameter based on its history. This means that it reduces the learning rate for frequently occurring features, and increases it for infrequently occurring features.
- Adadelta works by maintaining two exponentially decaying averages - one for the gradients and one for the updates - and using these averages to compute the learning rate for each parameter in the model. This helps Adadelta to adapt to the data more effectively and avoid some of the problems that Adagrad has with initial learning rates.
- **Advantages**: Adadelta is able to handle very noisy gradients and converge more quickly than other optimization techniques, even without an initial learning rate.
- **Limitations**: Its sensitivity to the hyperparameter decay, which controls the decay rate of the moving averages.

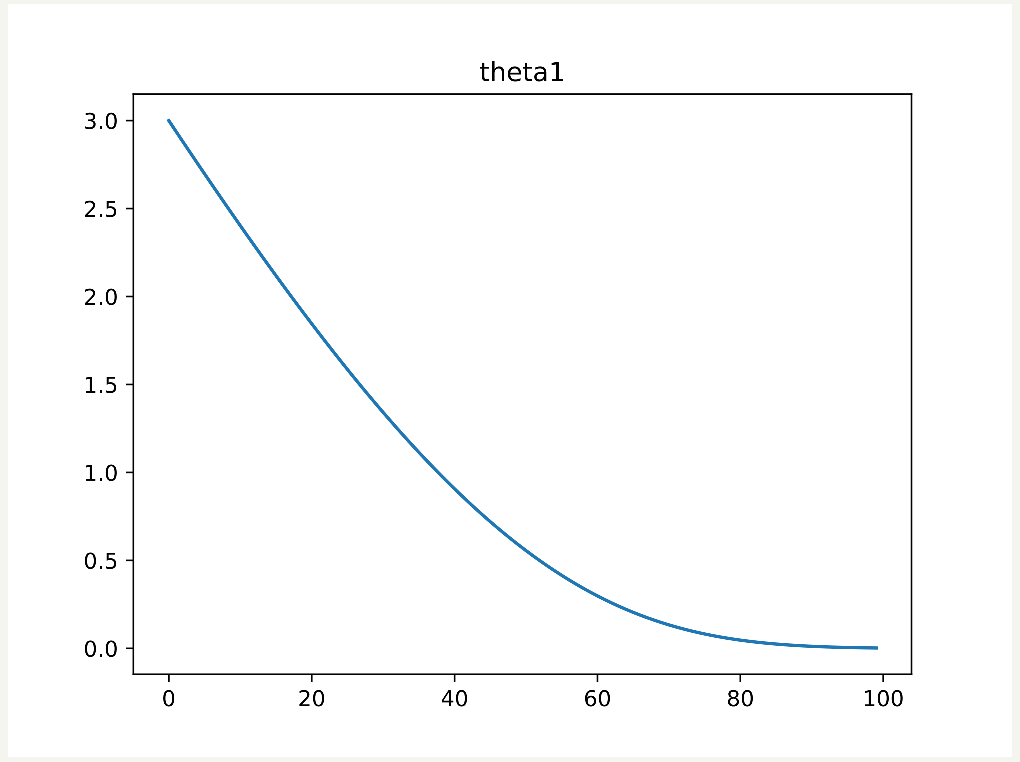
$$cache = decay * cache + (1 - decay) * gradient^2$$

- Adagrad suffered from too quick convergence of learning rate to 0, hence to make cache grow less fast, we decrease on each update.

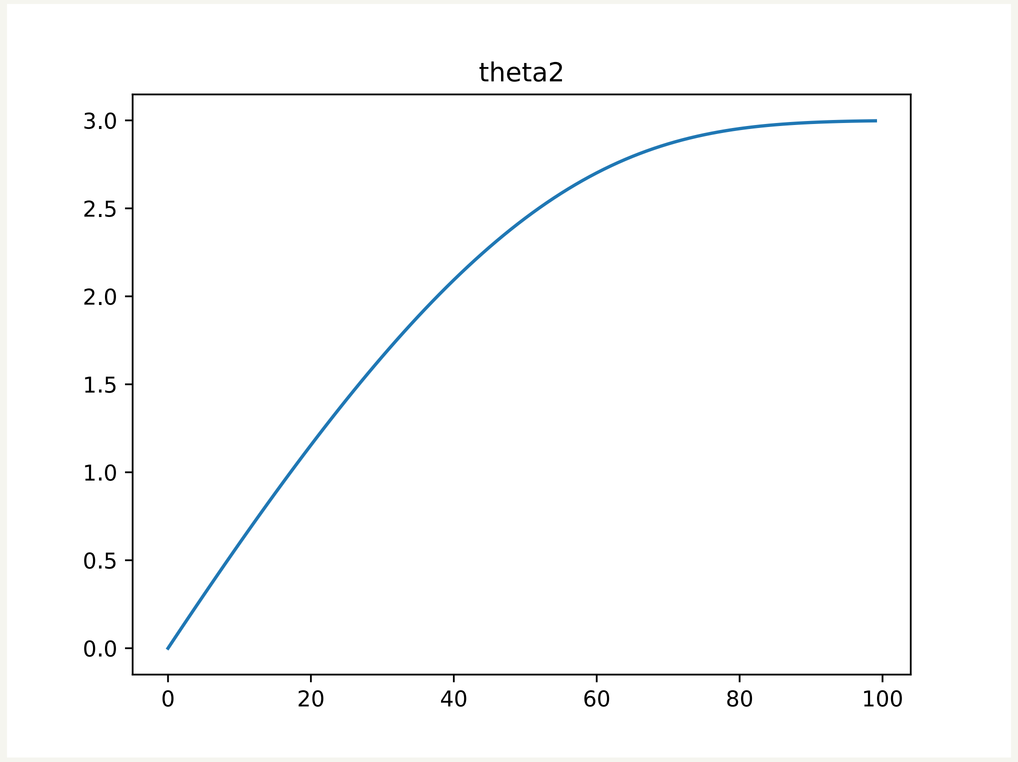
Code



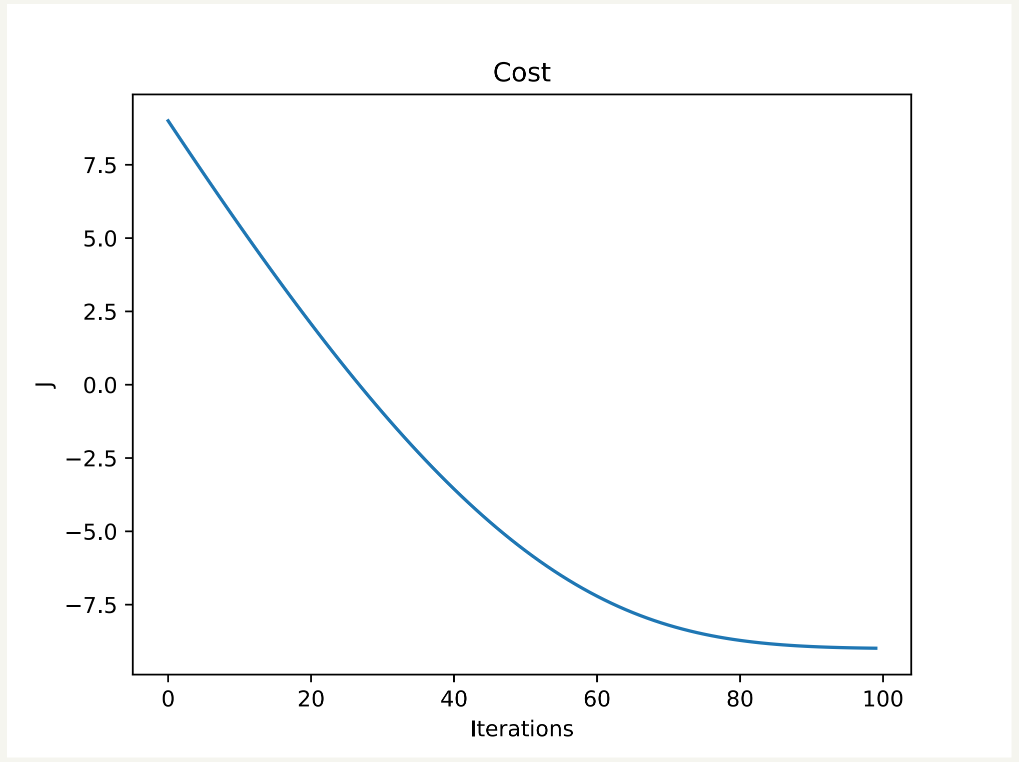
```python
theta1 = 3
theta2 = 0.000
eta = 1e-1  #learning rate
epsilon = 10e-8
E1 = 1 #initial cache is implemented differently in different packages. EXPERIMENT
E2 = 1
gamma = 0.95
iterations_adadelta = 100

theta1_adadelta_hist = []
theta2_adadelta_hist = []
J_adadelta_hist = []

for i in range(iterations_adagrad) :
    theta1_adadelta_hist.append(theta1)
    theta2_adadelta_hist.append(theta2)
    J = fun(theta1, theta2)
    J_adadelta_hist.append(J)
    if i % 20 == 0:
        print(f'Iteration = {i + 1} , theta1 = {theta1}, theta2 = {theta2}, J = {J}')

    grad1 = (alpha * (2 * theta1))
    grad2 = (alpha * (-2 * theta1))
    E1 = (gamma * E1) + (1- gamma) * (grad1 ** 2)
    E2 = (gamma * E2) + (1- gamma) * (grad2 ** 2)
    theta1 -=  eta * (grad1 / (np.sqrt(E1 + epsilon)))
    theta2 -=  eta * (grad2 / (np.sqrt(E2 + epsilon)))
```



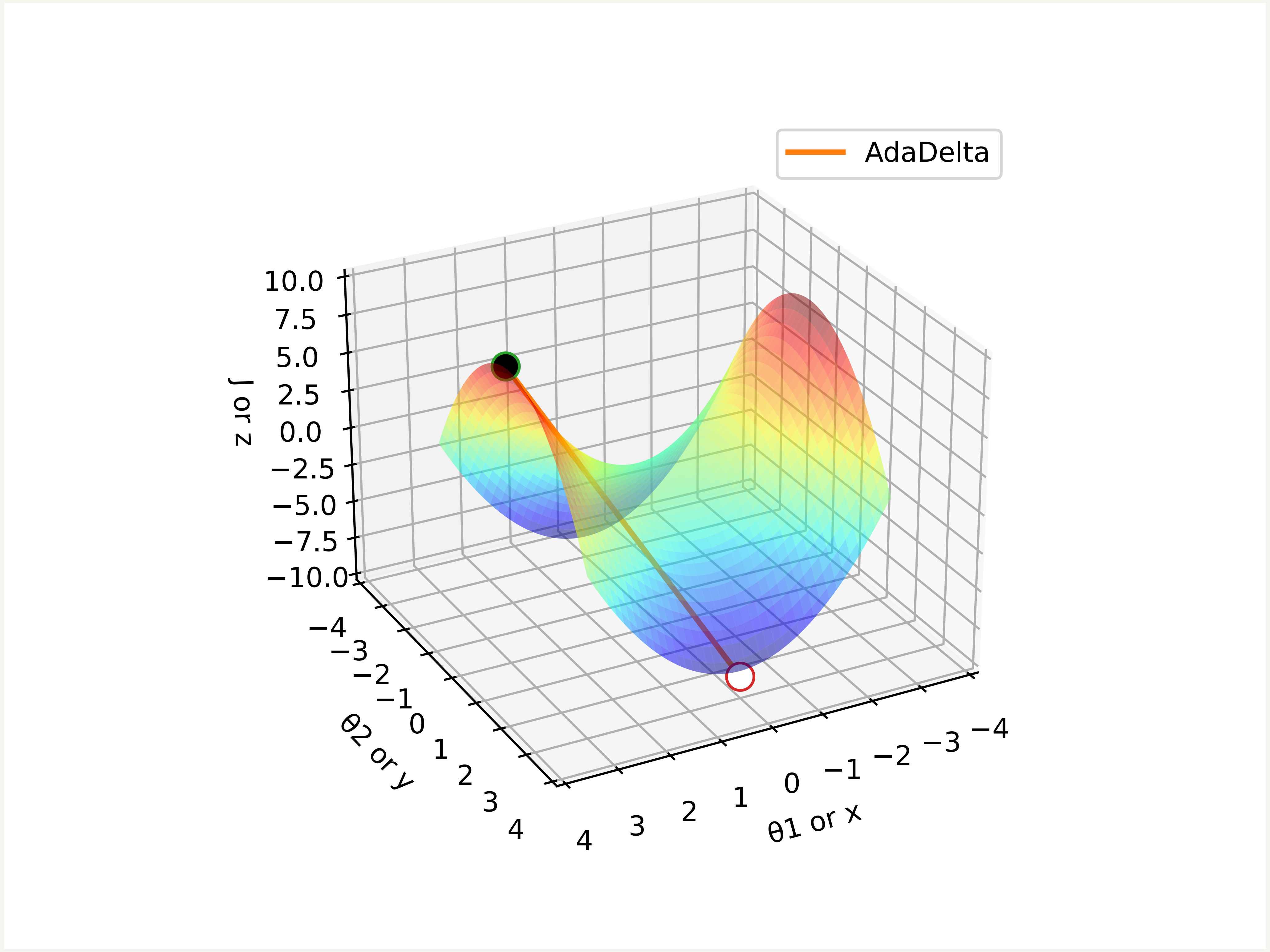
theta1

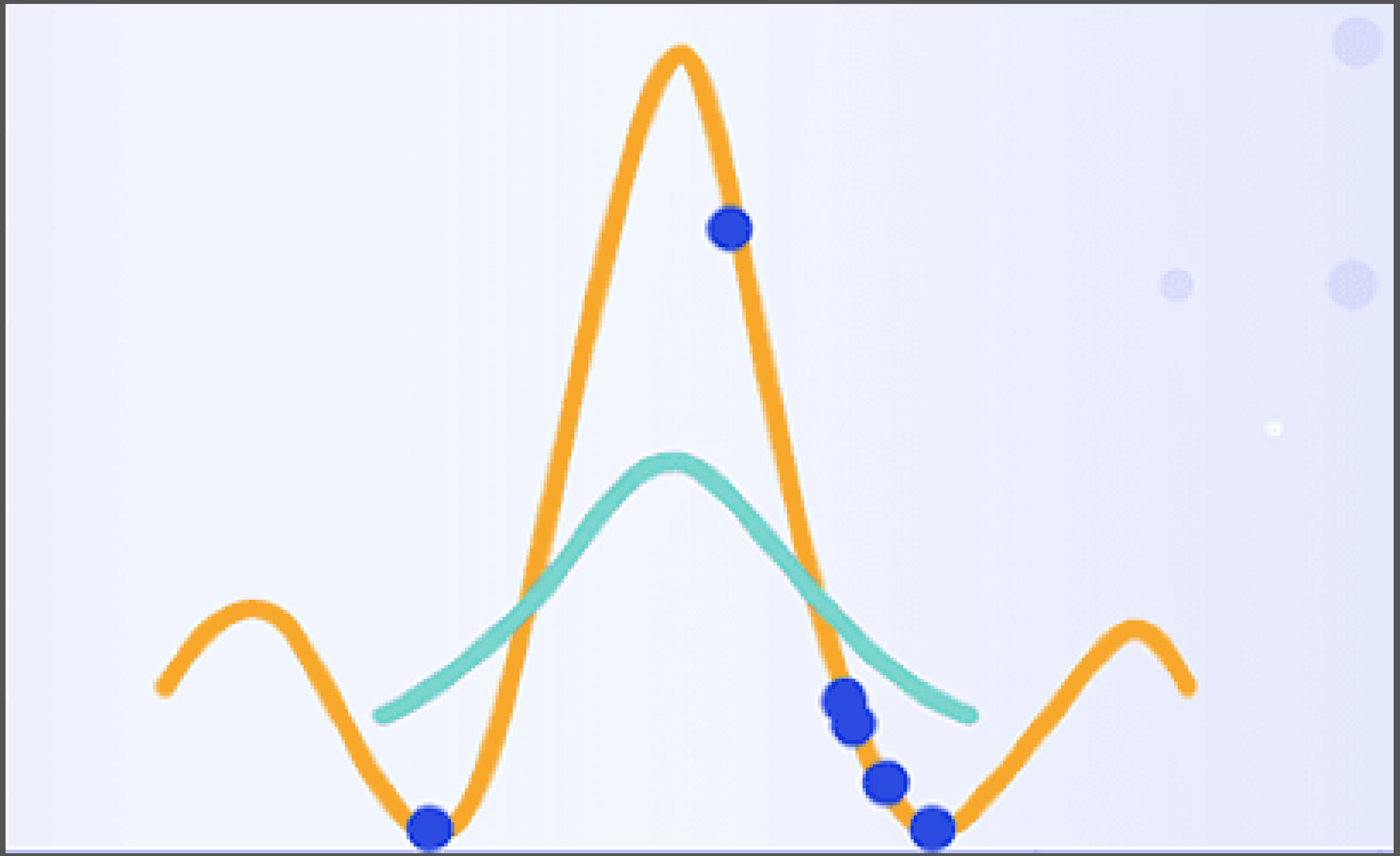


theta2

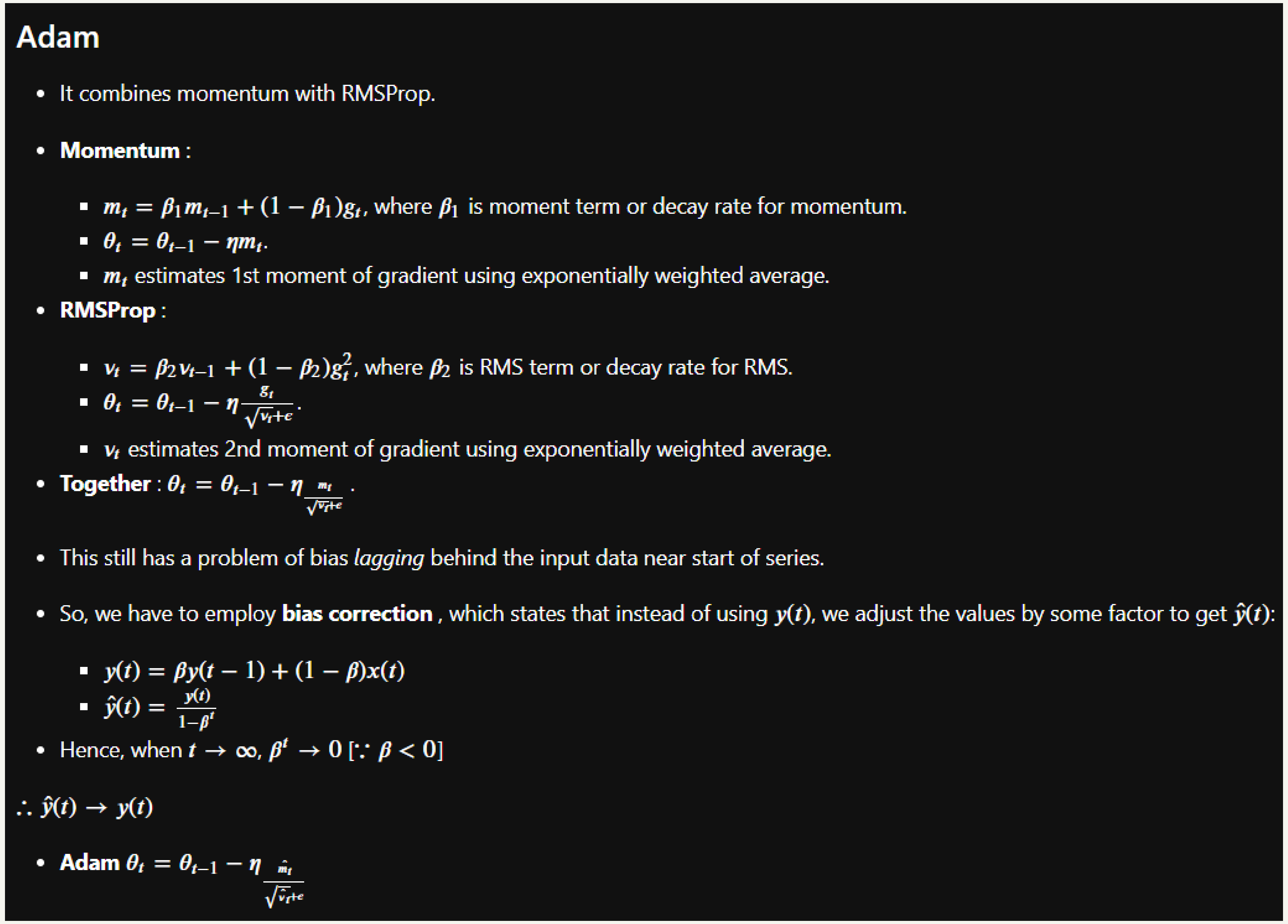


J(cost)


AdaDelta
10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
J or z
θ2 or y
θ1 or x
−4
−3
−2
−1
0
1
2
3
4



Adam



Adam

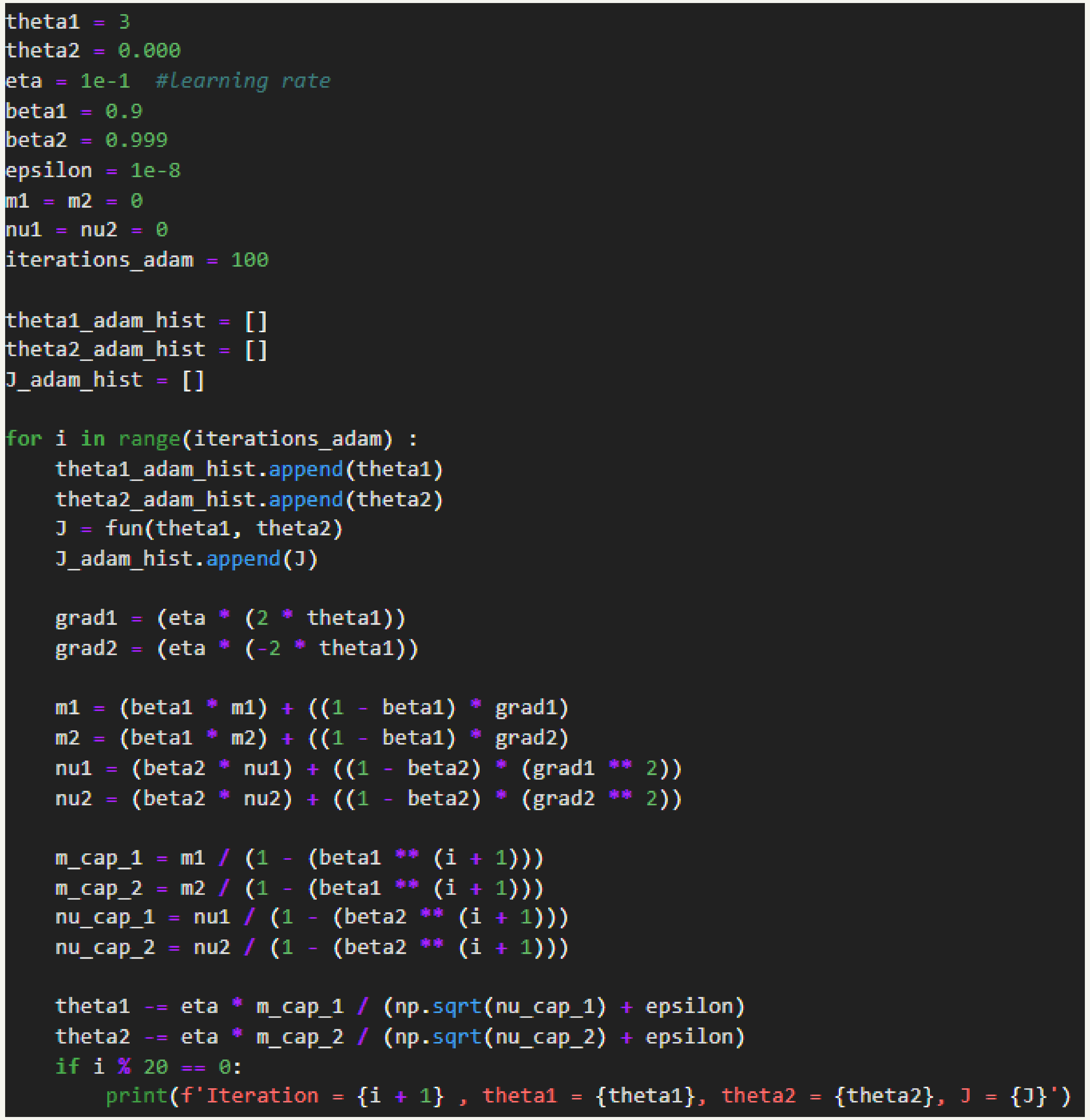
- It combines momentum with RMSProp.
- **Momentum** :
  - $m_t = \beta_1 m_{t-1} + (1-\beta_1)g_t$, where β_1 is moment term or decay rate for momentum.
  - $\theta_t = \theta_{t-1} - \eta m_t$.
  - m_t estimates 1st moment of gradient using exponentially weighted average.
- **RMSProp** :
  - $v_t = \beta_2 v_{t-1} + (1-\beta_2)g_t^2$, where β_2 is RMS term or decay rate for RMS.
  - $\theta_t = \theta_{t-1} - \eta \frac{g_t}{\sqrt{v_t + \epsilon}}$.
  - v_t estimates 2nd moment of gradient using exponentially weighted average.
- **Together** : $\theta_t = \theta_{t-1} - \eta \frac{m_t}{\sqrt{v_t + \epsilon}}$.
- This still has a problem of bias *lagging* behind the input data near start of series.
- So, we have to employ **bias correction** , which states that instead of using $y(t)$, we adjust the values by some factor to get $\hat{y}(t)$:
  - $y(t) = \beta y(t-1) + (1-\beta)x(t)$
  - $\hat{y}(t) = \frac{y(t)}{1-\beta^t}$
- Hence, when $t \to \infty$, $\beta^t \to 0$ [$\because \beta < 0$]

$\therefore \hat{y}(t) \to y(t)$

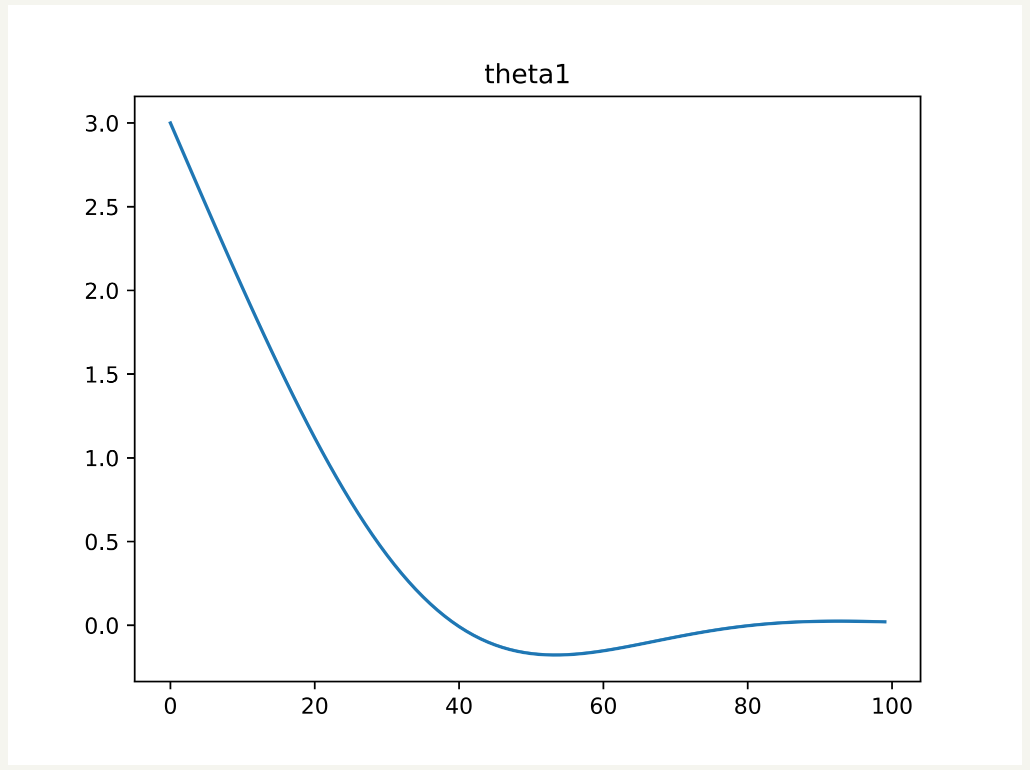
- **Adam** $\theta_t = \theta_{t-1} - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t + \epsilon}}$

- Adam is a popular variant of gradient descent that combines the benefits of both Adagrad and RMSProp. Adam is an adaptive learning rate method that uses the first and second moments of the gradients to adjust the learning rate.
- Adam works by maintaining two moving averages of the gradient - one for the mean and one for the variance - and using these averages to compute the learning rate for each parameter in the model. This helps Adam to adapt to the data more effectively and handle noisy gradients.
- **Advantages**: Adam is able to converge more quickly and efficiently than other optimization techniques, even for large-scale datasets and complex models.
- **Limitations**: Its sensitivity to the hyperparameter choices and the possible overfitting to small minibatches.

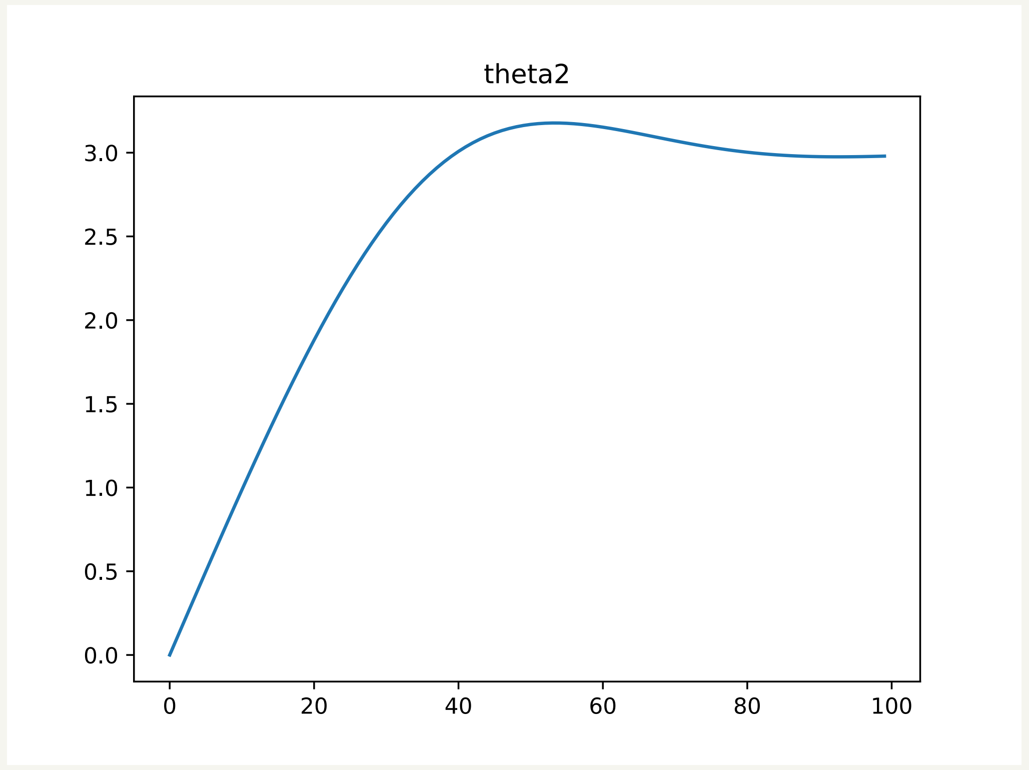
Code



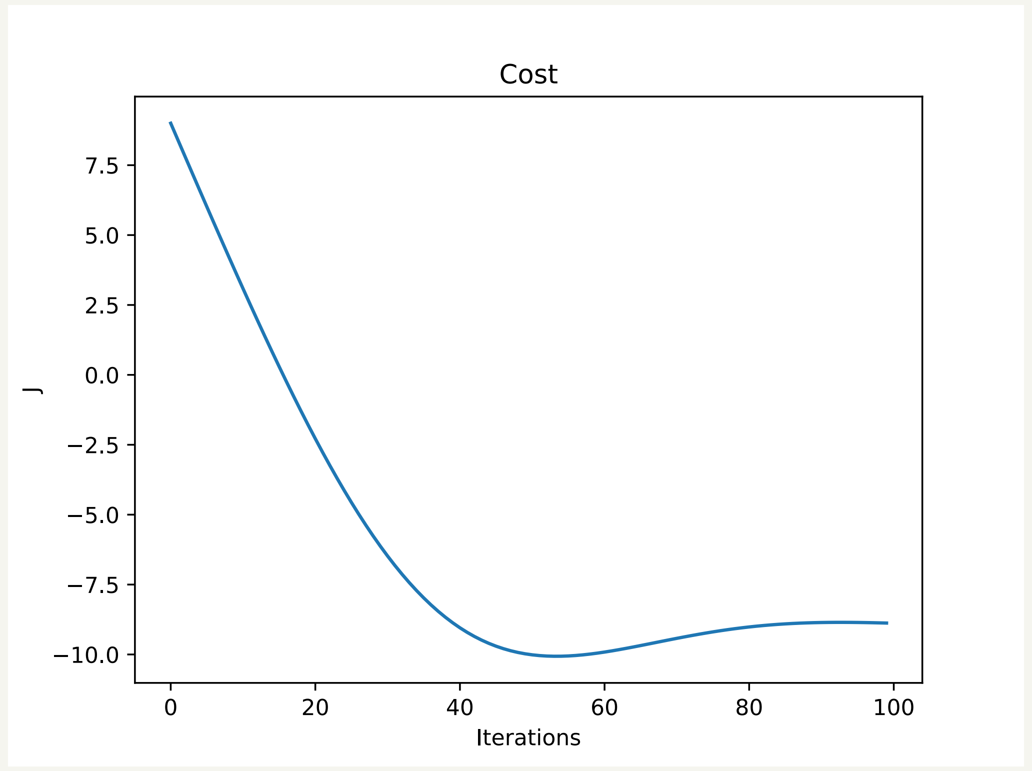
```python
theta1 = 3
theta2 = 0.000
eta = 1e-1  #learning rate
beta1 = 0.9
beta2 = 0.999
epsilon = 1e-8
m1 = m2 = 0
nu1 = nu2 = 0
iterations_adam = 100

theta1_adam_hist = []
theta2_adam_hist = []
J_adam_hist = []

for i in range(iterations_adam) :
    theta1_adam_hist.append(theta1)
    theta2_adam_hist.append(theta2)
    J = fun(theta1, theta2)
    J_adam_hist.append(J)

    grad1 = (eta * (2 * theta1))
    grad2 = (eta * (-2 * theta1))

    m1 = (beta1 * m1) + ((1 - beta1) * grad1)
    m2 = (beta1 * m2) + ((1 - beta1) * grad2)
    nu1 = (beta2 * nu1) + ((1 - beta2) * (grad1 ** 2))
    nu2 = (beta2 * nu2) + ((1 - beta2) * (grad2 ** 2))

    m_cap_1 = m1 / (1 - (beta1 ** (i + 1)))
    m_cap_2 = m2 / (1 - (beta1 ** (i + 1)))
    nu_cap_1 = nu1 / (1 - (beta2 ** (i + 1)))
    nu_cap_2 = nu2 / (1 - (beta2 ** (i + 1)))

    theta1 -= eta * m_cap_1 / (np.sqrt(nu_cap_1) + epsilon)
    theta2 -= eta * m_cap_2 / (np.sqrt(nu_cap_2) + epsilon)
    if i % 20 == 0:
        print(f'Iteration = {i + 1} , theta1 = {theta1}, theta2 = {theta2}, J = {J}')
```



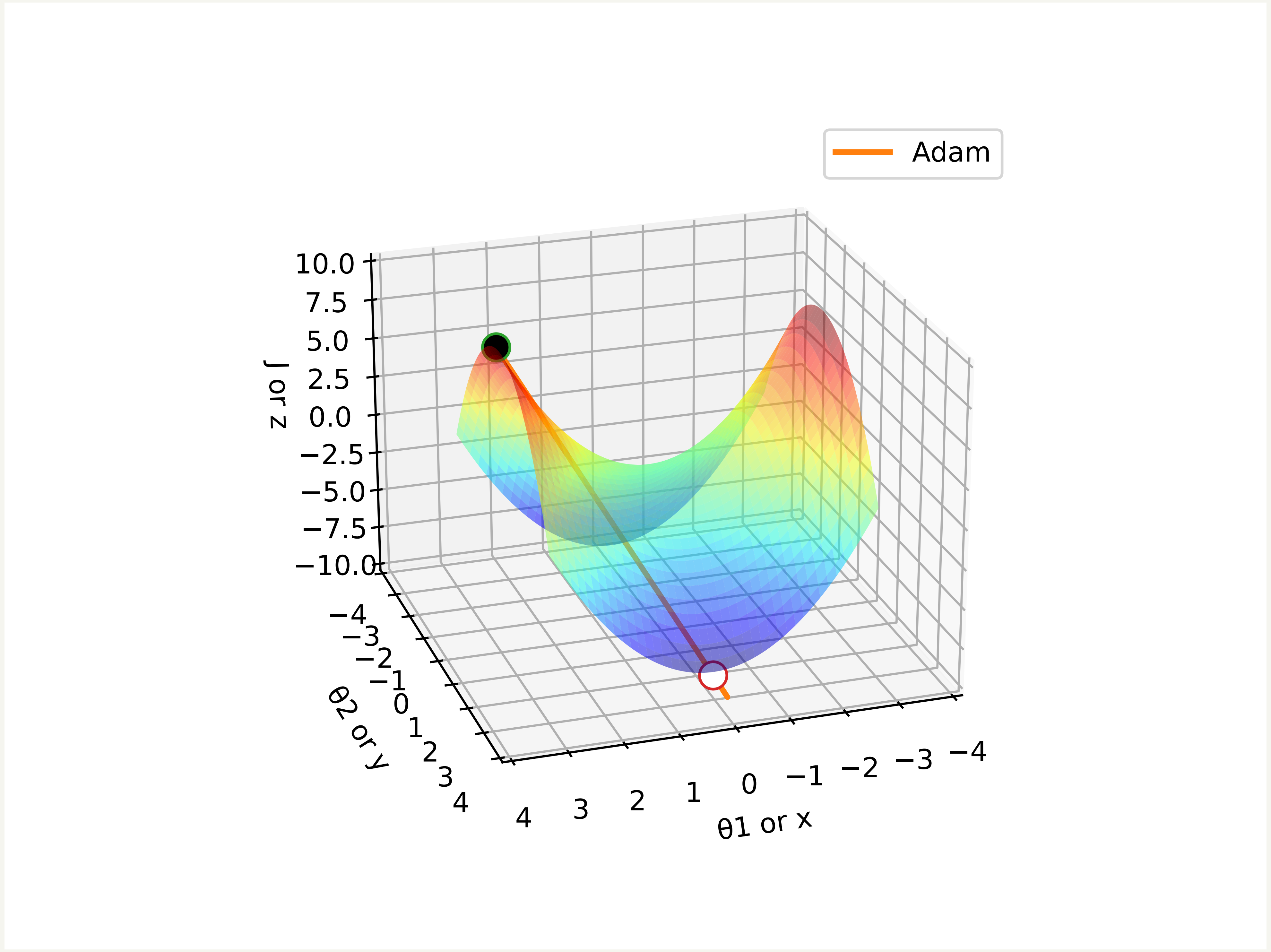
theta1



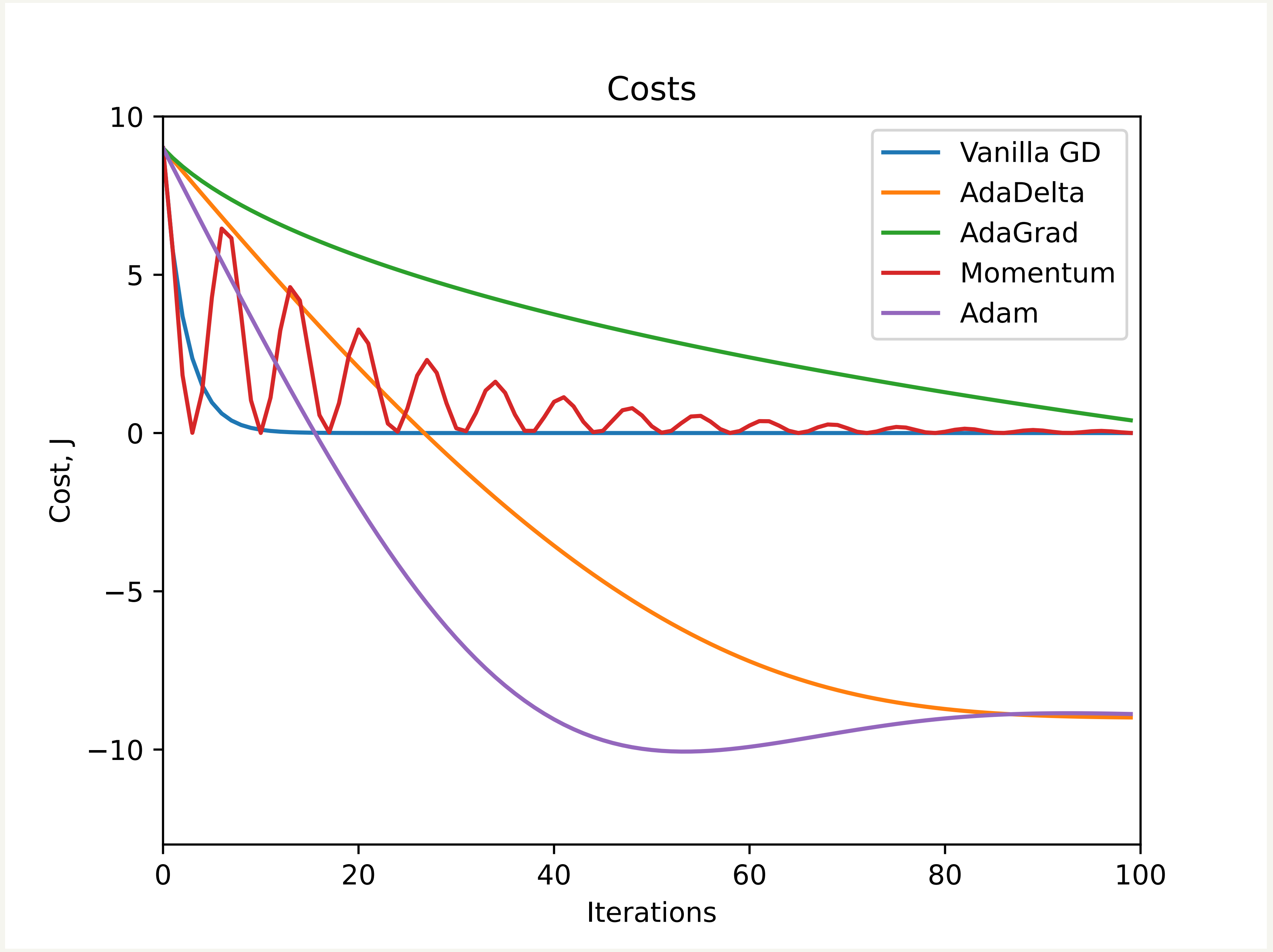
theta2



J(cost)


Adam
10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
J or z
−4
−3
−2
−1
0
1
2
3
4
θ2 or y
4
3
2
1
0
−1
−2
−3
−4
θ1 or x

Comparison


Costs
Vanilla GD
AdaDelta
AdaGrad
Momentum
Adam
Cost, J
Iterations
10
5
0
−5
−10
0
20
40
60
80
100