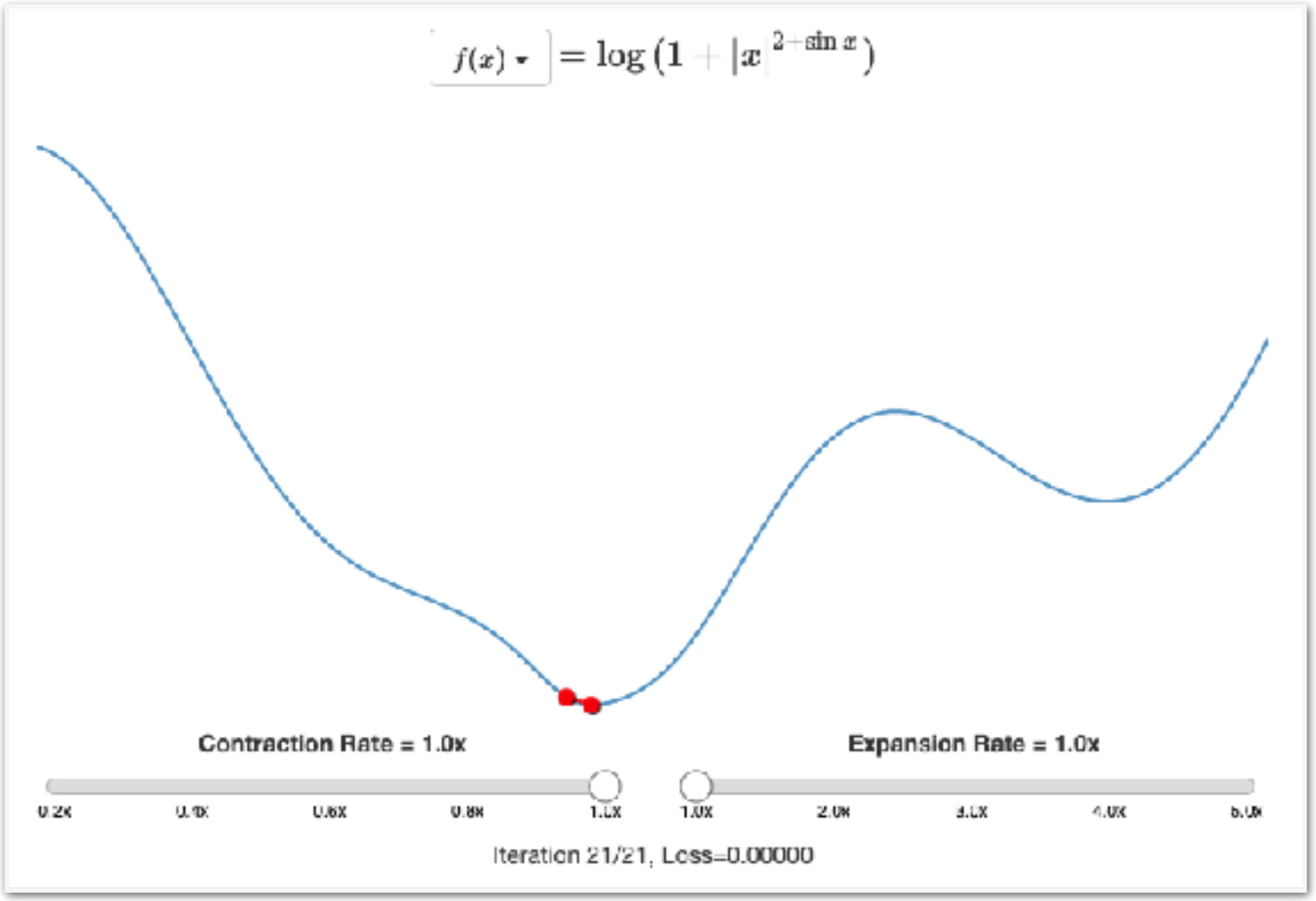
Jordi Vitrià

Deep Learning
Optimization

Nelder-Mead

The most simple thing we can try to minimize a function would be to sample two points relatively near each other, and and just repeatedly take a step down away from the largest value.

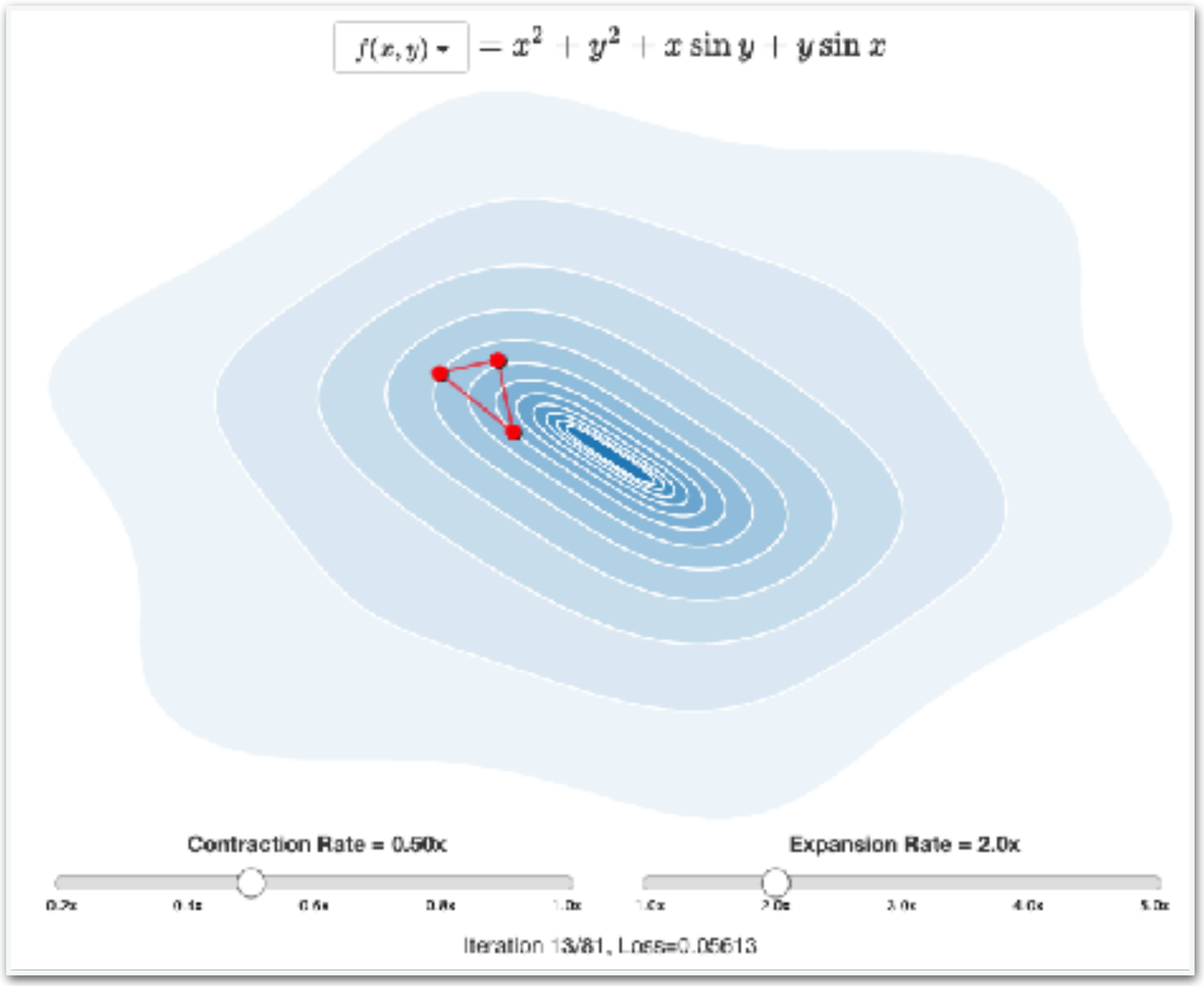


Iteration 21/21, Loss=0.00000

http://www.benfrederickson.com/numerical-optimization/

Nelder-Mead

The most simple thing we can try to minimize a function would be to sample two points relatively near each other, and and just repeatedly take a step down away from the largest value.



$f(x,y) = x^2 + y^2 + x \sin y + y \sin x$

Contraction Rate = 0.50x

Expansion Rate = 2.0x

Iteration 13/81, Loss=0.05613

http://www.benfrederickson.com/numerical-optimization/

Gradient Descend

Nelder-Mead method can be easily extended into higher dimensional examples, all that's required is taking one more point than there are dimensions.

In the case of deep learning and other high dimensional problems the main limitation of this method is **the number of function evaluations** wee need.

Let’s see an alternative: the use of **function derivatives**.

Let's suppose that we have a function we are considering the minimization of a function:

$$f(x) = x^2$$

Our objective is to find the argument x that **minimizes** this function.

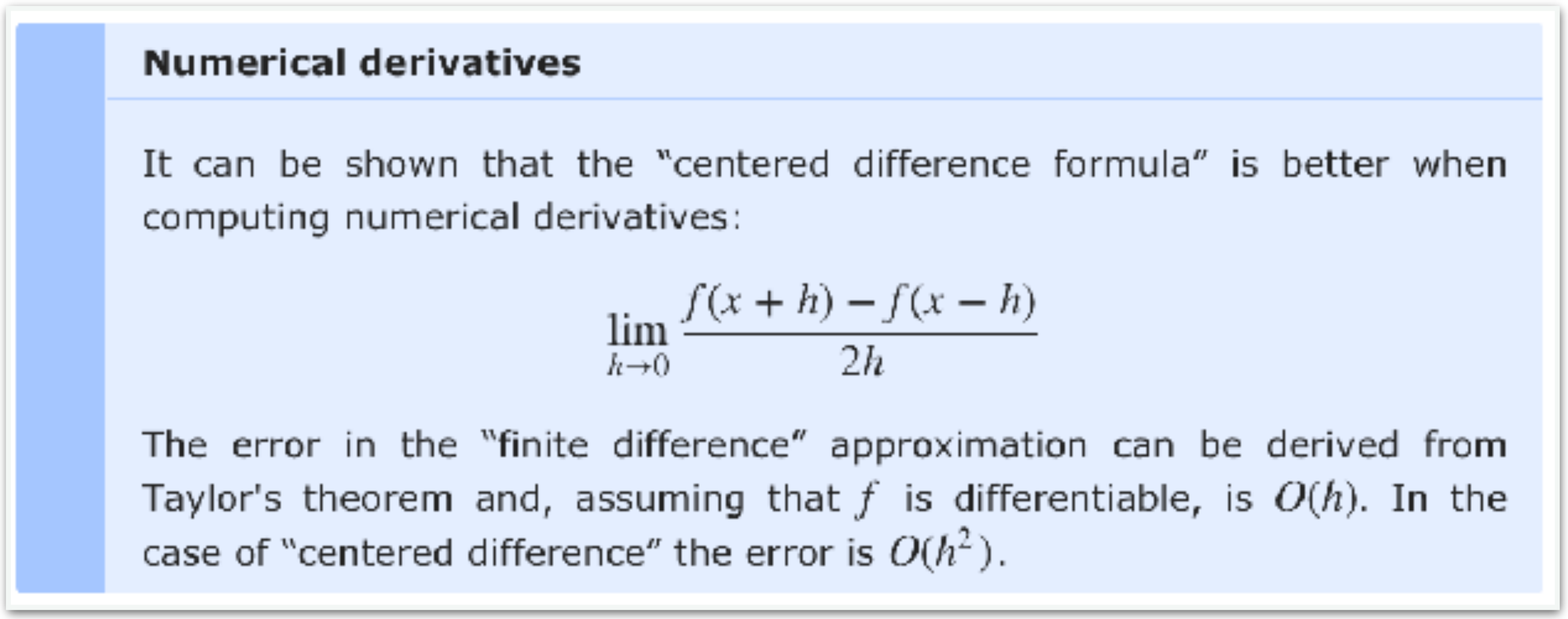
Gradient Descend

Derivative definition:

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

The derivative specifies how to scale a small change in the input in order to obtain the corresponding change in the output:

$$f(x+h) \approx f(x) + hf'(x)$$



Numerical derivatives

It can be shown that the "centered difference formula" is better when computing numerical derivatives:

$$\lim_{h \to 0} \frac{f(x+h) - f(x-h)}{2h}$$

The error in the "finite difference" approximation can be derived from Taylor's theorem and, assuming that f is differentiable, is $O(h)$. In the case of "centered difference" the error is $O(h^2)$.

Gradient Descend

The derivative tells how to change x in order to make a small improvement in order to make a small improvement in f. Then, we can follow these steps to decrease the value of the function:

- Start from a random x
- Compute the derivative $f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x-h)}{2h}$
- Walk a small step (possibly weighted by the derivative module) in the **opposite** direction of the derivative, because we know that $f(x - h\ \mathrm{sign}(f'(x))$ is less than $f(x)$ for a small step.

The search for the minima ends when the derivative is zero because we have no more information about which direction to move.

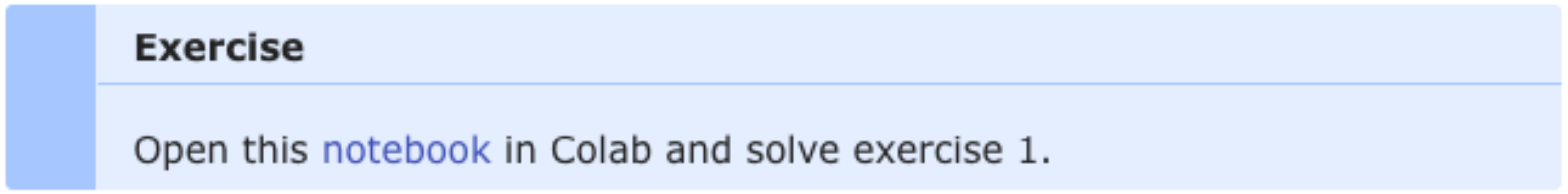
Gradient Descend

There are two problems with numerical derivatives:

- They are approximate.
- They are slower than necessary to evaluate (we need **two function evaluations**).

What about using calculus?

- Start from a random x
- Compute the derivative $f'(x)$ analitically
- Walk a small step in the **opposite** direction of the derivative.



Exercise

Open this notebook in Colab and solve exercise 1.

Gradient Descend

Let's consider a n-dimensional function $f : \mathbb{R}^n \to \mathbb{R}$.

$$f(\mathbf{x}) = \sum_n x_n^2$$

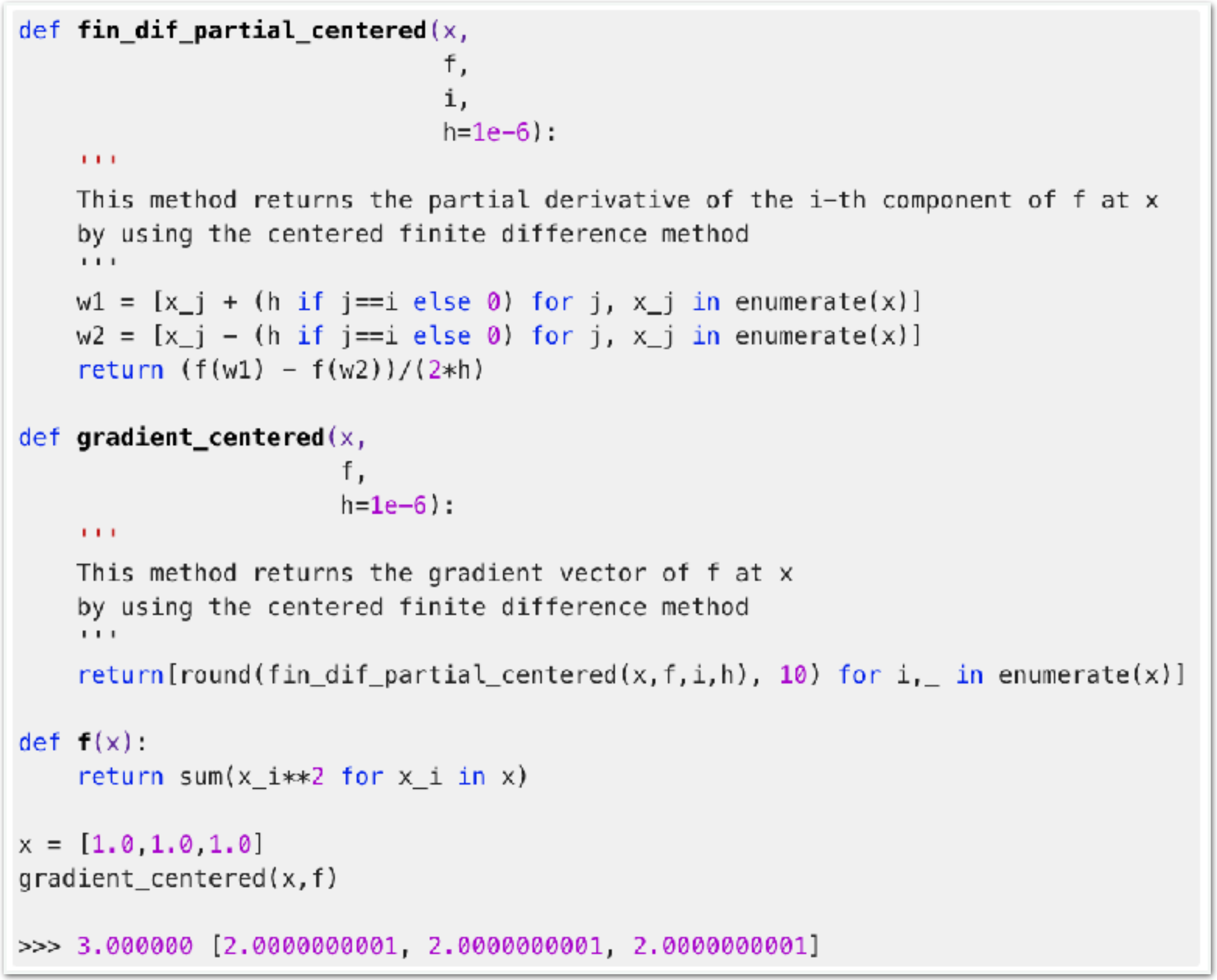
Our objective is to find the argument that minimizes this function. The gradient of ∇f, is the vector whose components are the n partial derivatives of f. It is thus a vector-valued function:

$$\nabla f = (\frac{\partial f}{\partial x_1}, \dots, \frac{\partial f}{\partial x_n})$$

The gradient points in the direction of the greatest rate of **increase** of the function.

It is important to be aware that this gradient computation is very expensive: if $\mathbf{x}$ has dimension n, we have to evaluate f at $2 * n$ points (2 for every dimension).

Gradient Descend



```
def fin_dif_partial_centered(x,
                             f,
                             i,
                             h=1e-6):
    '''
    This method returns the partial derivative of the i-th component of f at x
    by using the centered finite difference method
    '''
    w1 = [x_j + (h if j==i else 0) for j, x_j in enumerate(x)]
    w2 = [x_j - (h if j==i else 0) for j, x_j in enumerate(x)]
    return (f(w1) - f(w2))/(2*h)

def gradient_centered(x,
                      f,
                      h=1e-6):
    '''
    This method returns the gradient vector of f at x
    by using the centered finite difference method
    '''
    return[round(fin_dif_partial_centered(x,f,i,h), 10) for i,_ in enumerate(x)]

def f(x):
    return sum(x_i**2 for x_i in x)

x = [1.0,1.0,1.0]
gradient_centered(x,f)

>>> 3.000000 [2.0000000001, 2.0000000001, 2.0000000001]
```

Gradient Descend

The step size, **alpha** or h, is a slippy concept: if it is too small we will slowly converge to the solution, if it is too large we can diverge from the solution.

There are several policies to follow when selecting the step size:

- Constant size steps. In this case, the size step determines the precision of the solution.
- Decreasing step sizes.
- At each step, select the optimal step.

Magical step size: 0.01

Gradient Descend and Machine Learning

In general, we have:

- A dataset $(\mathbf{x}, y)$ of N examples.
- A target function $f_{\mathbf{w}}$, that we want to minimize, representing the **discrepancy between our data and the model** we want to fit. The model is indexed by a set of parameters $\mathbf{w}$.
- The gradient of the target function, g_f.

To fit the model is to find the optimal parameters $\mathbf{w}$ that minimize the following expression:

$$\frac{1}{N}\sum_i (y_i - f(\mathbf{x}_i, \mathbf{w}))^2$$

Example

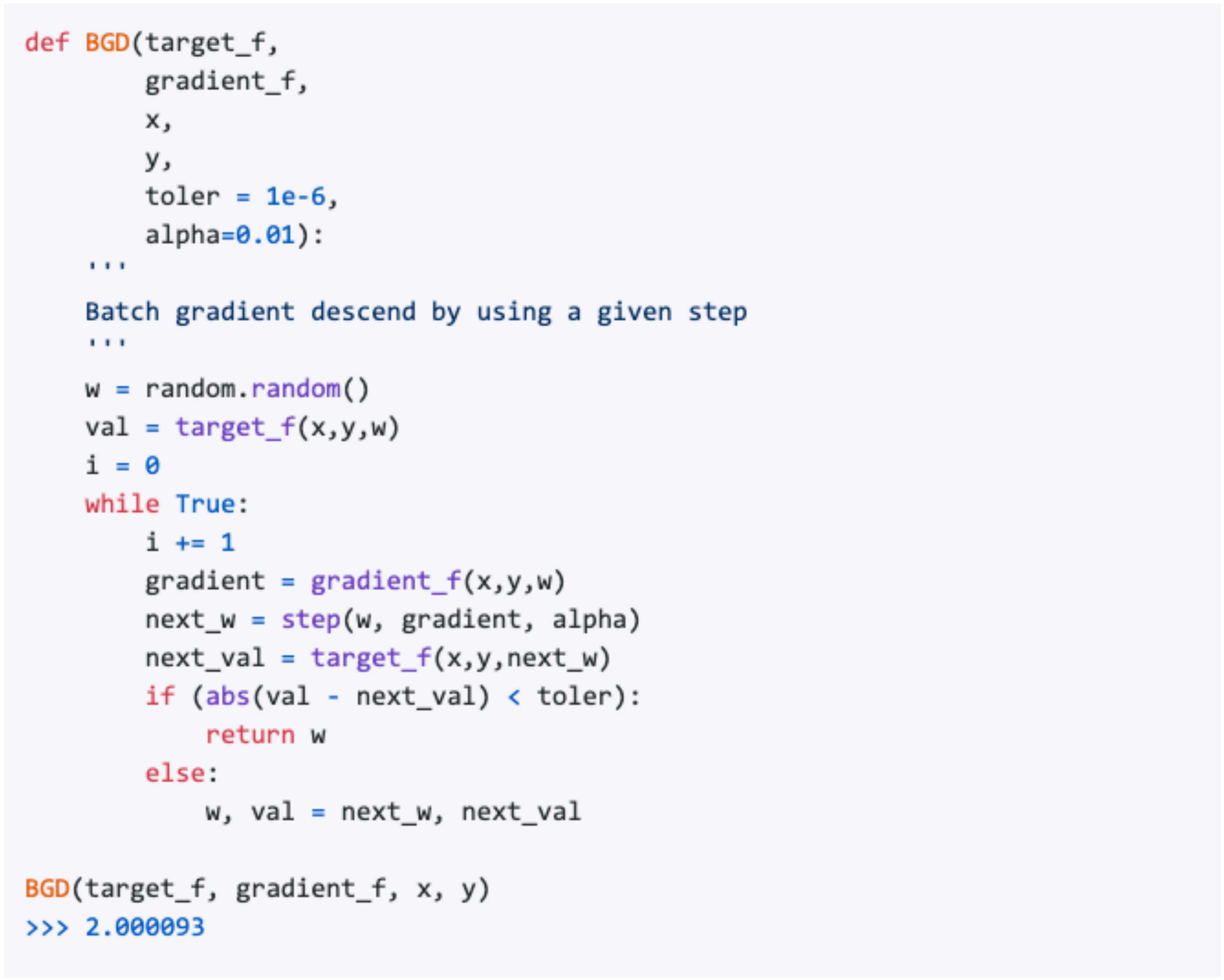
Let’s suppose that our model is a one-dimensional linear model:

$$f(\mathbf{x}, \mathbf{w}) = w \cdot x$$

We can implement **gradient descend** in the following way:

```python
import numpy as np
import random

# f = 2x
x = np.arange(10)
y = np.array([2*i for i in x])

# f_target = 1/n Sum (y - wx)**2
def target_f(x,y,w):
    return np.sum((y - x * w)**2.0) / x.size

# gradient_f = 2/n Sum 2wx**2 - 2xy
def gradient_f(x,y,w):
    return 2 * np.sum(2*w*(x**2) - 2*x*y) / x.size

def step(w,grad,alpha):
    return w - alpha * grad
```

Example



```python
def BGD(target_f,
        gradient_f,
        x,
        y,
        toler = 1e-6,
        alpha=0.01):
    '''
    Batch gradient descend by using a given step
    '''
    w = random.random()
    val = target_f(x,y,w)
    i = 0
    while True:
        i += 1
        gradient = gradient_f(x,y,w)
        next_w = step(w, gradient, alpha)
        next_val = target_f(x,y,next_w)
        if (abs(val - next_val) < toler):
            return w
        else:
            w, val = next_w, next_val

BGD(target_f, gradient_f, x, y)
>>> 2.000093
```

It is called (batch) gradient descend because at every step, when computing

next_val = target_f(x,y,next_w)

we are using the whole dataset!

Stochastic Gradient Descend

The last function evals the whole dataset $(\mathbf{x}_i, y_i)$ at every step. If the dataset is large, this strategy is too costly.

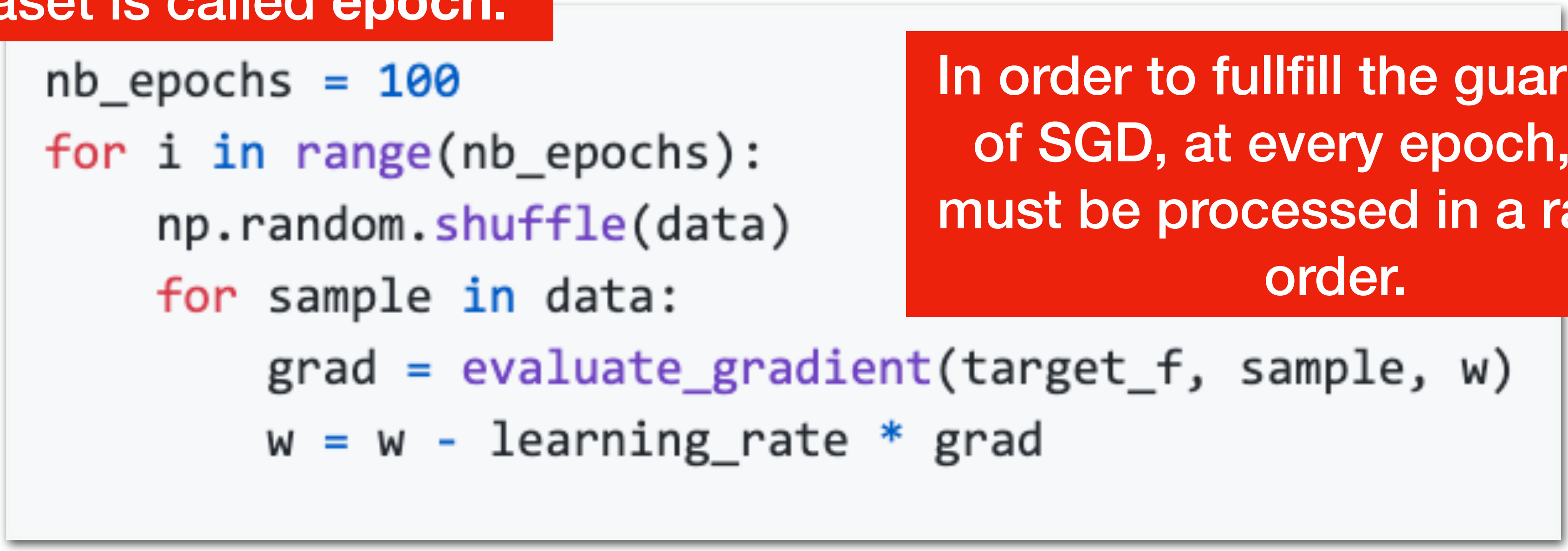
In this case we will use a strategy called **SGD** (*Stochastic Gradient Descend*), that is based on the following fact:

When learning from data, the cost function is additive: it is computed by adding sample reconstruction errors.
In this case, it can be shown that we can compute a good estimate of the gradient (and move towards the minimum) by using only one data sample (or a small data sample) at each iteration.

If we apply this method we have some **theoretical guarantees** to find a good minimum.

Stochastic Gradient Descend

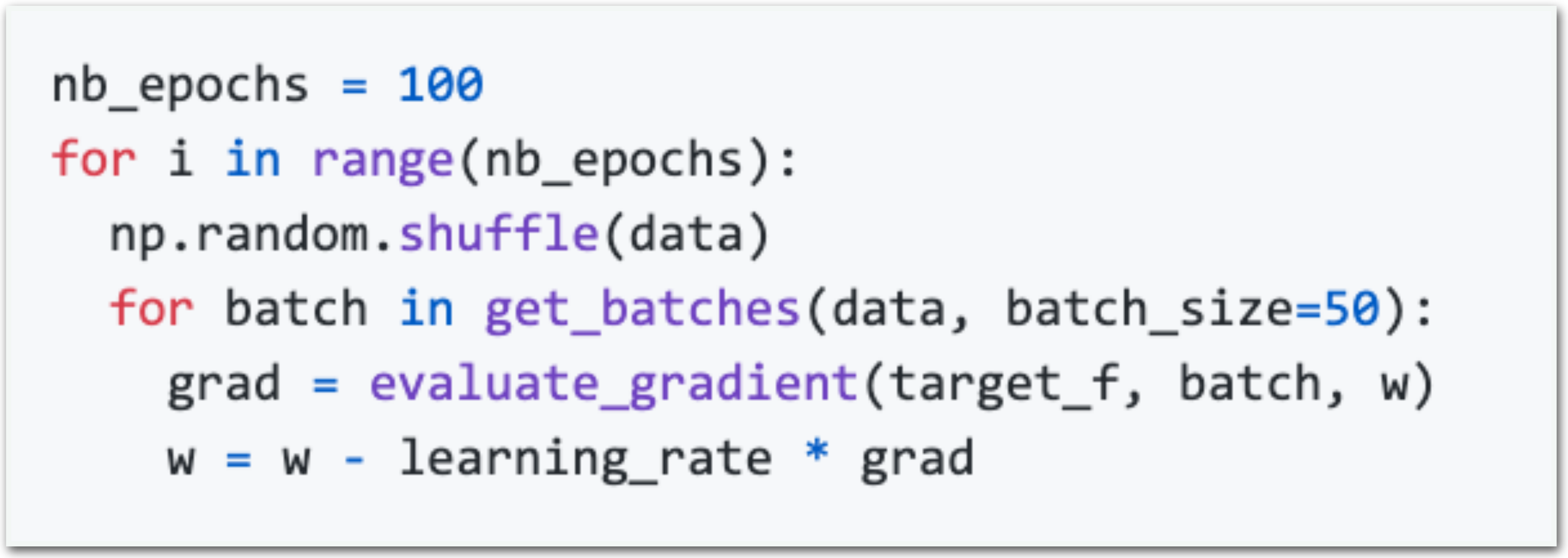
A full iteration over the dataset is called **epoch.**



```
nb_epochs = 100
for i in range(nb_epochs):
    np.random.shuffle(data)
    for sample in data:
        grad = evaluate_gradient(target_f, sample, w)
        w = w - learning_rate * grad
```

In order to fullfill the guarantees of SGD, at every epoch, data must be processed in a random order.

Minibatch Gradient Descend



```
nb_epochs = 100
for i in range(nb_epochs):
  np.random.shuffle(data)
  for batch in get_batches(data, batch_size=50):
    grad = evaluate_gradient(target_f, batch, w)
    w = w - learning_rate * grad
```

Loss Functions

$$L(y, f(\mathbf{x})) = \frac{1}{n}\sum_i \ell(y_i, f(\mathbf{x_i}))$$

Loss functions represent the price paid for inaccuracy of predictions in classification/regression problems.

In regression problems, the most common loss function is the **square loss function**:

$$L(y, f(\mathbf{x})) = \frac{1}{n}\sum_i (y_i - f(\mathbf{x}_i))^2$$

In classification this function could be the **zero-one loss,** that is

$$\ell(y_i, f(\mathbf{x_i}))$$

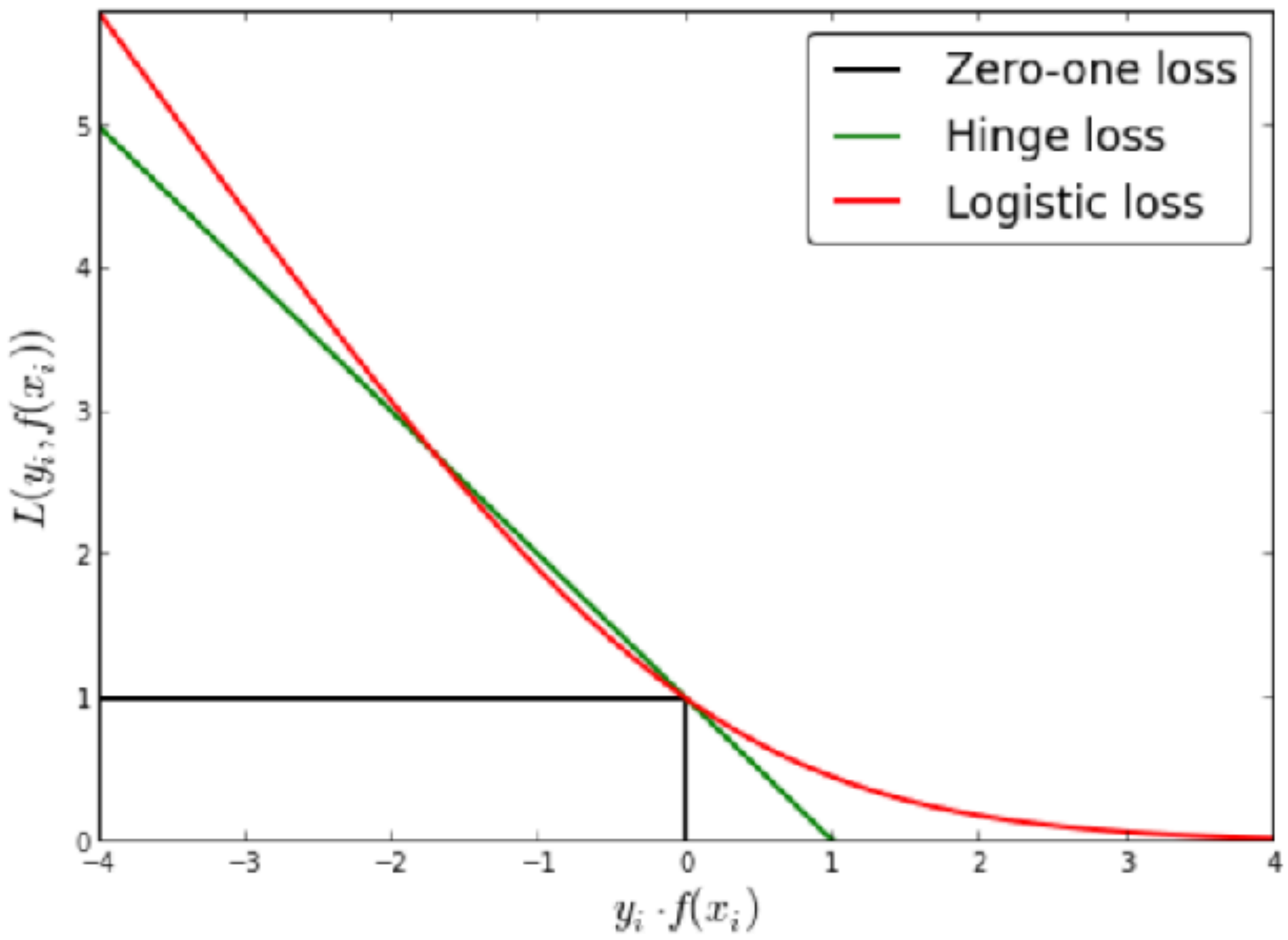
is 0 when $y_i = f(\mathbf{x}_i)$ and 1 otherwise.

This function is discontinuous with flat regions and is thus extremely hard to optimize using gradient-based methods. For this reason it is usual to consider a proxy to the loss called a *surrogate loss function*. For computational reasons this is usually convex function.

Surrogate Loss Functions

$$L(y, f(\mathbf{x})) = \frac{1}{N} \sum_{i} \max(0, 1 - y_i f(\mathbf{x}_i))$$

Hinge / Margin Loss (i.e. Suport Vector Machines)



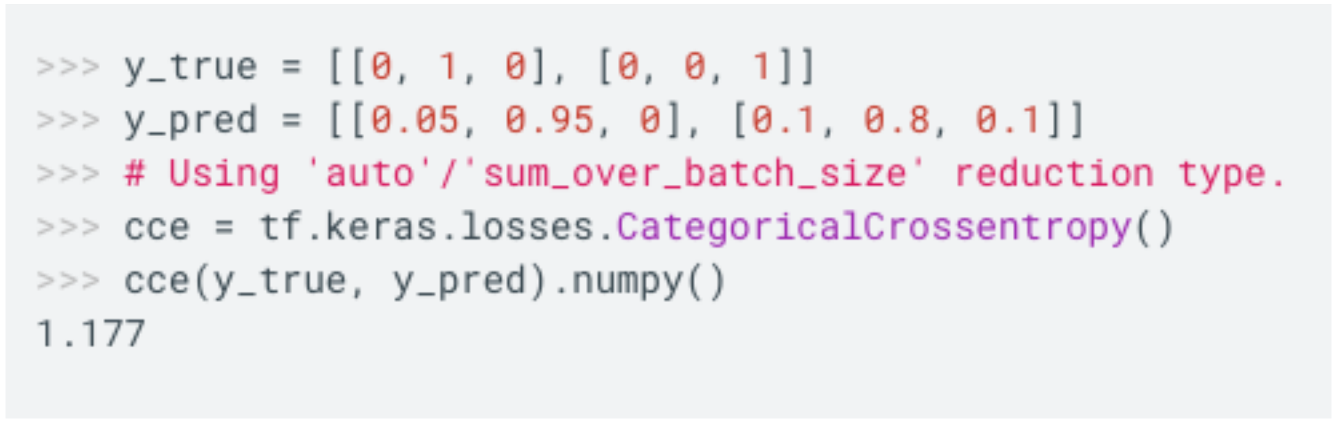
Surrogate Loss Functions

$$L(y, \hat{y}) = \frac{1}{N} \sum_{i} log(1 + exp(-y_i \hat{y}_i))$$

Binary Logistic Regression

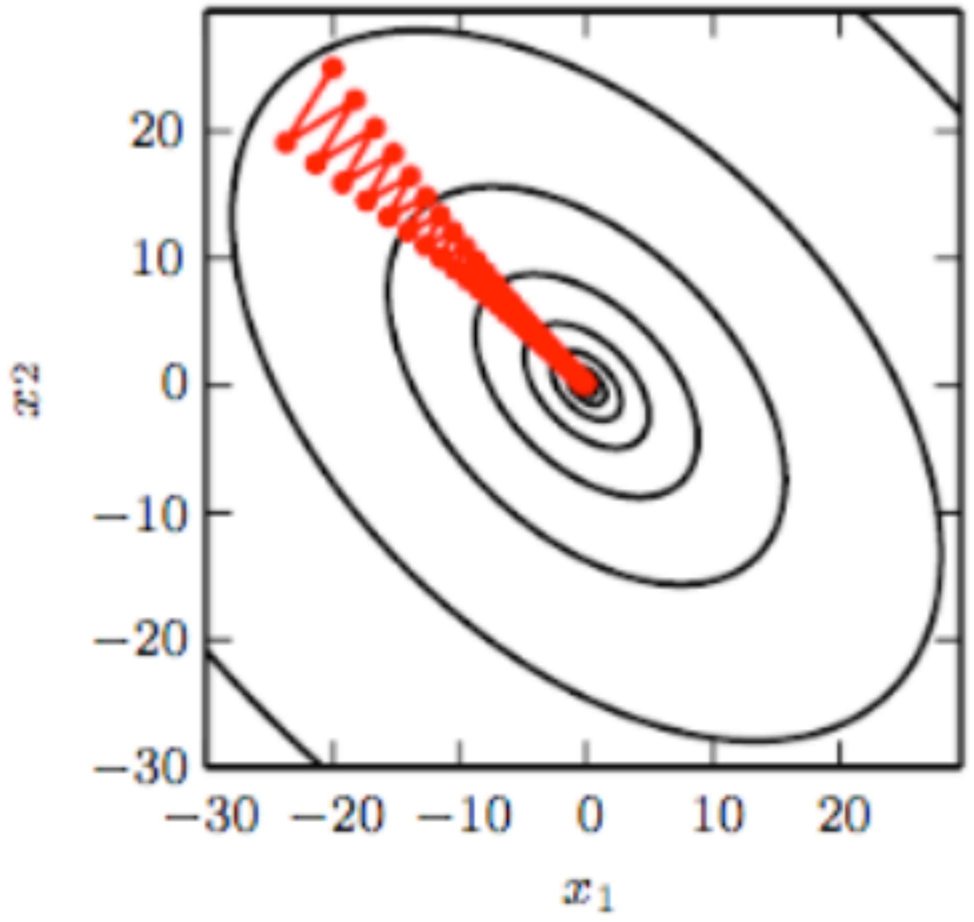
$$L(y, \hat{y}) = -\frac{1}{N} \sum_{i} \sum_{j \in C} y_{i,j} \log \hat{y}_{i,j}$$

Multiclass Cross-Entropy Loss

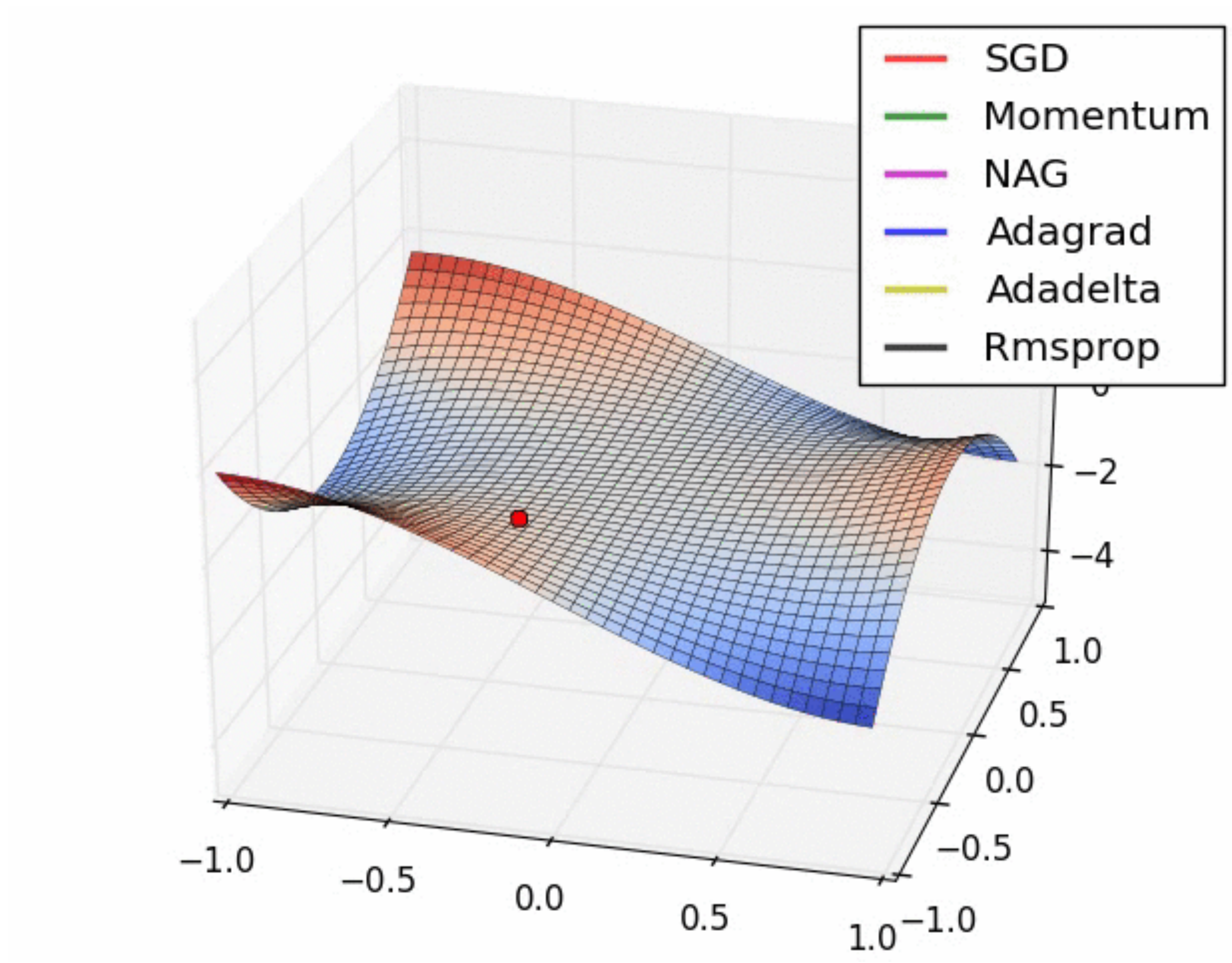

```
>>> y_true = [[0, 1, 0], [0, 0, 1]]
>>> y_pred = [[0.05, 0.95, 0], [0.1, 0.8, 0.1]]
>>> # Using 'auto'/'sum_over_batch_size' reduction type.
>>> cce = tf.keras.losses.CategoricalCrossentropy()
>>> cce(y_true, y_pred).numpy()
1.177
```

Advanced Gradient Descend

SGD has trouble navigating ravines, i.e. areas where the surface curves much more steeply in one dimension than in another, which are common around local optima. In these scenarios, SGD oscillates across the slopes of the ravine while only making hesitant progress along the bottom towards the local optimum.



Advanced Gradient Descend


SGD
Momentum
NAG
Adagrad
Adadelta
Rmsprop