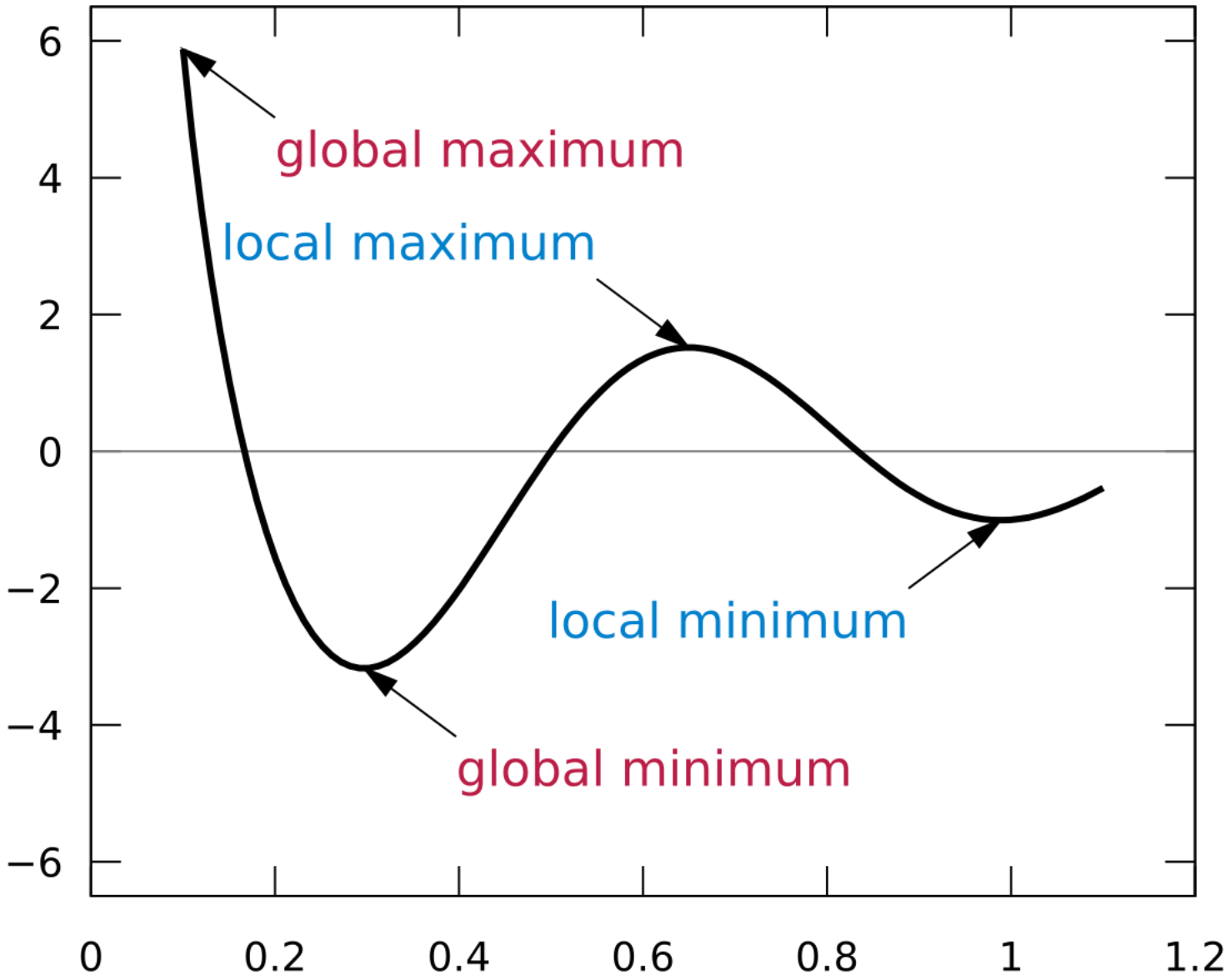
Stochastic Gradient Descent Algorithm

Basic Gradient Descent Algorithm

- The gradient descent algorithm is an approximate and iterative method for mathematical optimization. You can use it to approach the minimum of any differentiable function.
- Although gradient descent sometimes gets stuck in a local minimum or a saddle point instead of finding the global minimum, it's widely used in practice.
- machine learning methods often apply it internally to optimize model parameters. For example, neural networks find weights and biases with gradient descent.

local and global minima



Cost Function: The Goal of Optimization

- The **cost function**, or loss function, is the function to be minimized (or maximized) by varying the decision variables.
- Many machine learning methods solve optimization problems under the surface.
- ML algorithms tend to minimize the difference between actual and predicted outputs by adjusting the model parameters (like weights and biases for neural networks, decision rules for random forest or gradient boosting, and so on).

Minimize errors in regression problems

- In a regression problem, you typically have the vectors of input variables $\mathbf{x} = (x_1, \ldots, x_r)$ and the actual outputs y.
- You want to find a model that maps $\mathbf{x}$ to a predicted response $f(\mathbf{x})$ so that $f(\mathbf{x})$ is as close as possible to y.
- For example, you might want to predict an output such as a person's salary given inputs like the person's number of years at the company or level of education.
- Your goal is to minimize the difference between the prediction $f(\mathbf{x})$ and the actual data y. This difference is called the **residual**.

Minimize errors in regression problems

- In this type of problem, you want to minimize the sum of squared residuals (SSR), where SSR = $\Sigma_i(y_i - f(\mathbf{x}_i))^2$ for all observations $i = 1, \ldots, n$, where n is the total number of observations. Alternatively, you could use the mean squared error (MSE = SSR / n) instead of SSR.
- Both SSR and MSE use the square of the difference between the actual and predicted outputs. The lower the difference, the more accurate the prediction. A difference of zero indicates that the prediction is equal to the actual data.
- SSR or MSE is minimized by adjusting the model parameters. For example, in linear regression, you want to find the function $f(\mathbf{x}) = b_0 + b_1x_1 + \cdots + b_rx_r$, so you need to determine the weights $b_0, b_1, \ldots, b_r$ that minimize SSR or MSE.

Classification problems

- In a classification problem, the outputs y are categorical, often either 0 or 1. For example, you might try to predict whether an email is spam or not. In the case of binary outputs, it's convenient to minimize the cross-entropy function that also depends on the actual outputs y_i and the corresponding predictions $p(\mathbf{x}_i)$:

$$H = -\sum_i \left(y_i \log(p(\mathbf{x}_i)) + (1 - y_i) \log(1 - p(\mathbf{x}_i))\right)$$

Logistic Regression

- In logistic regression, which is often used to solve classification problems, the functions $p(\mathbf{x})$ and $f(\mathbf{x})$ are defined as the following:

$$p(\mathbf{x}) = \frac{1}{1 + \exp(-f(\mathbf{x}))}$$

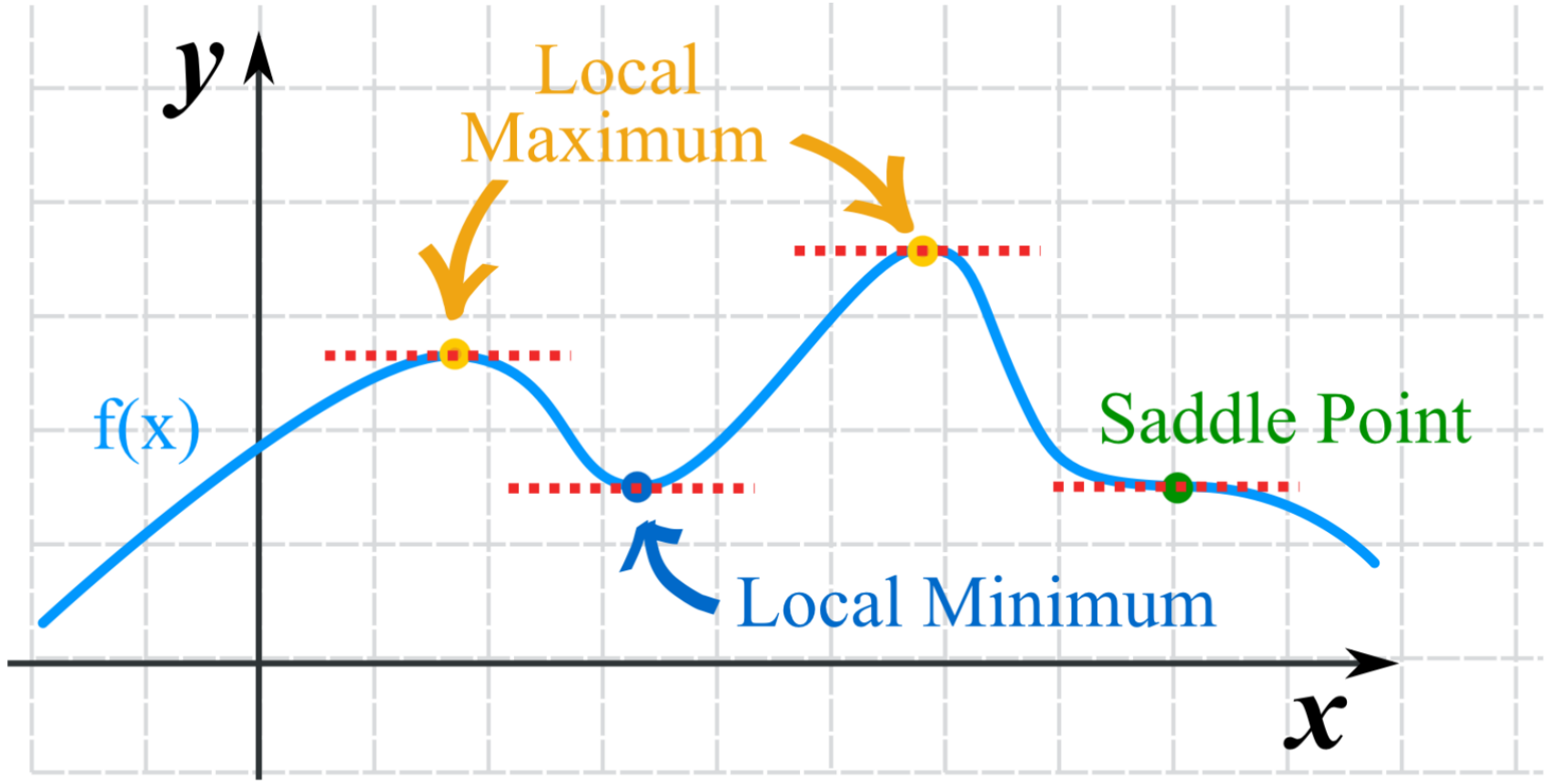
$$f(\mathbf{x}) = b_0 + b_1 x_1 + \cdots + b_r x_r$$

- Again, you need to find the weights b_0, b_1, …, b_r, but this time they should minimize the cross-entropy function.

Gradient of a Function: Calculus Refresher

- In calculus, the derivative of a function shows you how much a value changes when you modify its argument (or arguments).
- Derivatives are important for optimization because the zero derivatives might indicate a minimum, maximum, or saddle point.
- The gradient of a function C of several independent variables $v_1, \ldots, v_r$ is denoted with $\nabla C(v_1, \ldots, v_r)$ and defined as the vector function of the partial derivatives of C with respect to each independent variable: $\nabla C = (\partial C/\partial v_1, \ldots, \partial C/v_r)$. The symbol ∇ is called nabla.

Minimum, maximum, and saddle point



Gradient of a Function: Calculus Refresher

- The nonzero value of the gradient of a function C at a given point defines the direction and rate of the fastest increase of C.
- When working with gradient descent, you're interested in the direction of the fastest *decrease* in the cost function.
- This direction is determined by the negative gradient, $-\nabla C$.

Intuition Behind Gradient Descent

- To understand the gradient descent algorithm, imagine a drop of water sliding down the side of a bowl or a ball rolling down a hill.
- The drop and the ball tend to move in the direction of the fastest decrease until they reach the bottom. With time, they’ll gain momentum and accelerate.

Intuition Behind Gradient Descent

- The idea behind gradient descent is similar: you start with an arbitrarily chosen position of the point or vector $\mathbf{v} = (v_1, \ldots, v_r)$ and move it iteratively in the direction of the fastest decrease of the cost function. As mentioned, this is the direction of the negative gradient vector, $-\nabla C$.
- Once you have a random starting point $\mathbf{v} = (v_1, \ldots, v_r)$, you **update** it, or move it to a new position in the direction of the negative gradient: $\mathbf{v} \rightarrow \mathbf{v} - \eta\nabla C$, where η (pronounced “ee-tah”) is a small positive value called the **learning rate**.

Learning Rate

- The learning rate determines how large the update or moving step is. It's a very important parameter.
- If η is too small, then the algorithm might converge very slowly.
- Large η values can also cause issues with convergence or make the algorithm divergent.

Implementation of Basic Gradient Descent

- This is a basic implementation of the algorithm that starts with an arbitrary point, start, iteratively moves it toward the minimum, and returns a point that is hopefully at or near the minimum:

```
def gradient_descent(gradient, start, learn_rate, n_iter):
    vector = start
    for _ in range(n_iter):
        diff = -learn_rate * gradient(vector)
        vector += diff
    return vector
```

The function takes a starting point (line 2), iteratively updates it according to the learning rate and the value of the gradient (lines 3 to 5), and finally returns the last position found.

gradient_descent() arguments

1. gradient is the function or any Python callable object that takes a vector and returns the gradient of the function you're trying to minimize.
2. start is the point where the algorithm starts its search, given as a sequence (tuple, list, NumPy array, and so on) or scalar (in the case of a one-dimensional problem).
3. learn_rate is the learning rate that controls the magnitude of the vector update.
4. n_iter is the number of iterations.

Add another termination criterion

```
import numpy as np

def gradient_descent(
    gradient, start, learn_rate, n_iter=50, tolerance=1e-
06):
    vector = start
    for _ in range(n_iter):
        diff = -learn_rate * gradient(vector)
        if np.all(np.abs(diff) <= tolerance):
            break
        vector += diff
    return vector
```

You now have the additional parameter tolerance (line 4), which specifies the minimal allowed movement in each iteration. You've also defined the default values for tolerance and n_iter, so you don't have to specify them each time you call gradient_descent().

Add another termination criterion

- Lines 9 and 10 enable gradient_descent() to stop iterating and return the result before n_iter is reached if the vector update in the current iteration is less than or equal to tolerance. This often happens near the minimum, where gradients are usually very small. Unfortunately, it can also happen near a local minimum or a saddle point.
- Line 9 uses the convenient NumPy functions numpy.all() and numpy.abs() to compare the absolute values of diff and tolerance in a single statement. That’s why you import numpy on line 1.

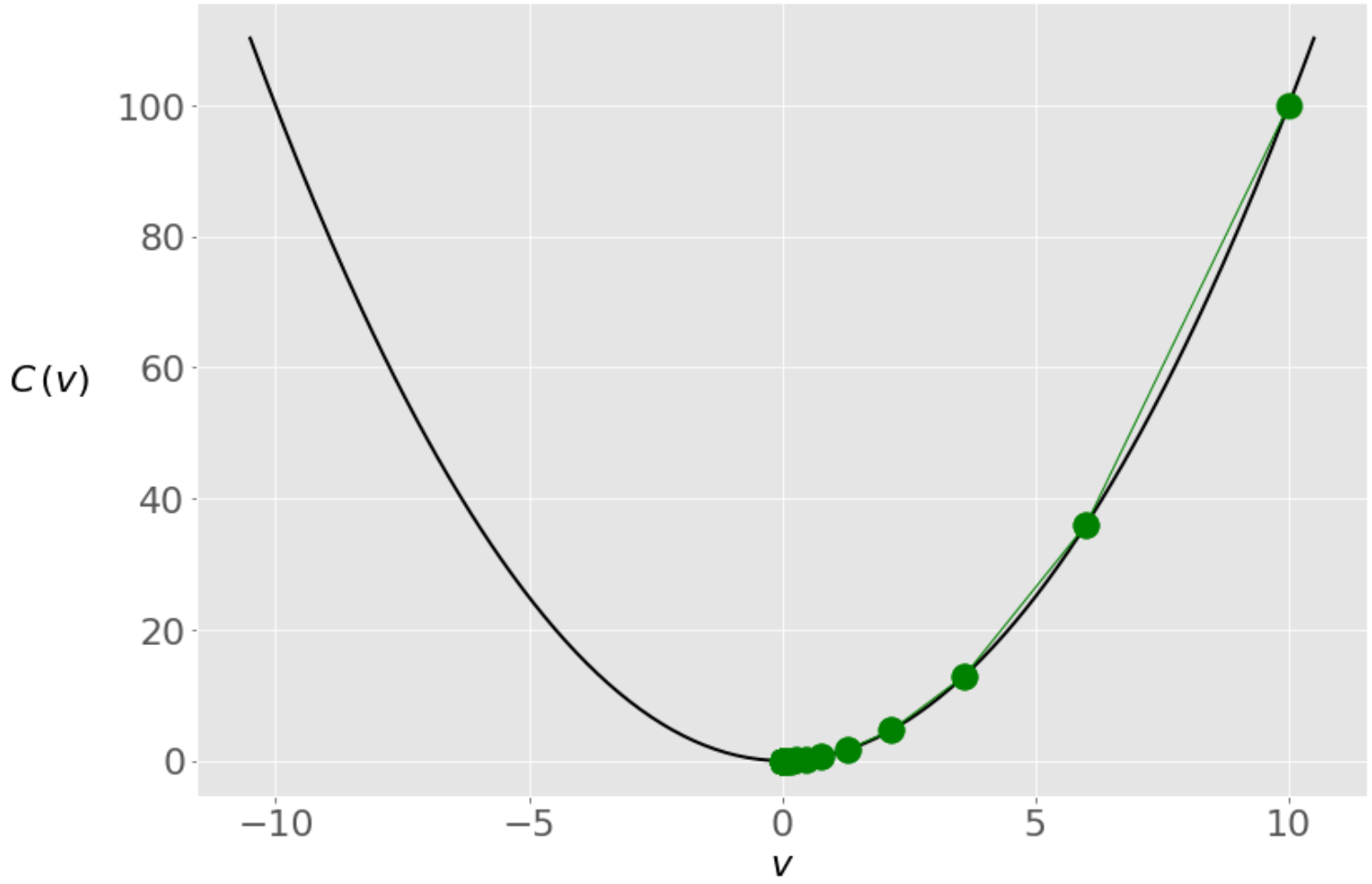
Test the function

- start with a small example and find the minimum of the function $C = v^2$.
- This function has only one independent variable (v), and its gradient is the derivative $2v$. It's a differentiable convex function, and the analytical way to find its minimum is straightforward.
- However, in practice, analytical differentiation can be difficult or even impossible and is often approximated with numerical methods.

```
>>> gradient_descent(
...     gradient=lambda v: 2 * v, start=10.0, learn_rate=0.2)
2.210739197207331e-06
```

Test the function

- You use the lambda function lambda v: 2 * v to provide the gradient of v^2. You start from the value 10.0 and set the learning rate to 0.2.
- You get a result that's very close to zero, which is the correct minimum.
- You start from the rightmost green dot ($v = 10$) and move toward the minimum ($v = 0$). The updates are larger at first because the value of the gradient (and slope) is higher. As you approach the minimum, they become lower.



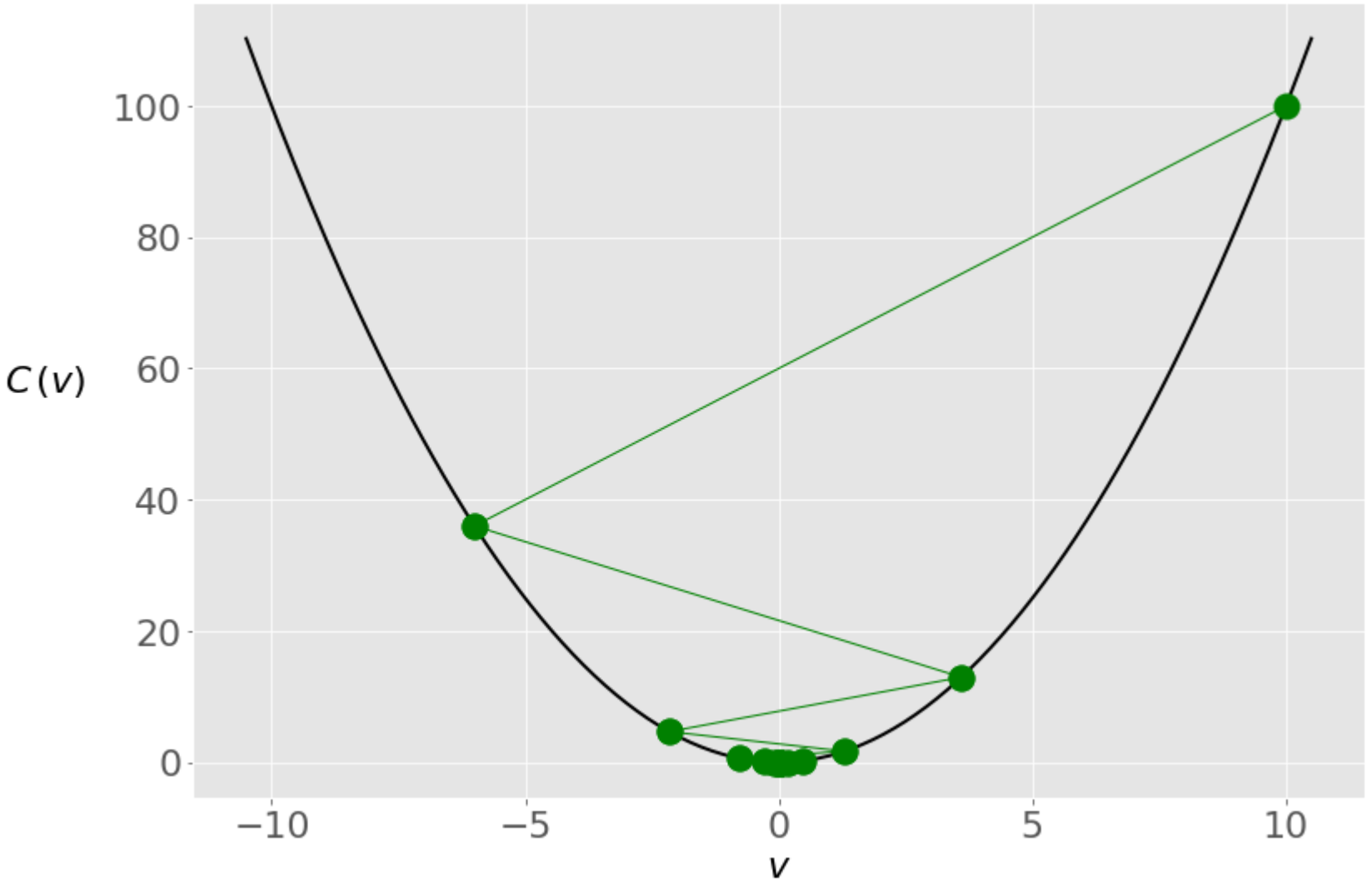
Learning Rate Impact

- The learning rate is a very important parameter of the algorithm. Different learning rate values can significantly affect the behavior of gradient descent. Consider the previous example, but with a learning rate of 0.8 instead of 0.2

```
>>> gradient_descent(
...     gradient=lambda v: 2 * v, start=10.0, learn_rate=0.8)
-4.7751966659678 6e-07
```

Learning Rate Impact

- You get another solution that's very close to zero, but the internal behavior of the algorithm is different.
- This is what happens with the value of v through the iterations
- you again start with v = 10, but because of the high learning rate, you get a large change in v that passes to the other side of the optimum and becomes −6. It crosses zero a few more times before settling near it.

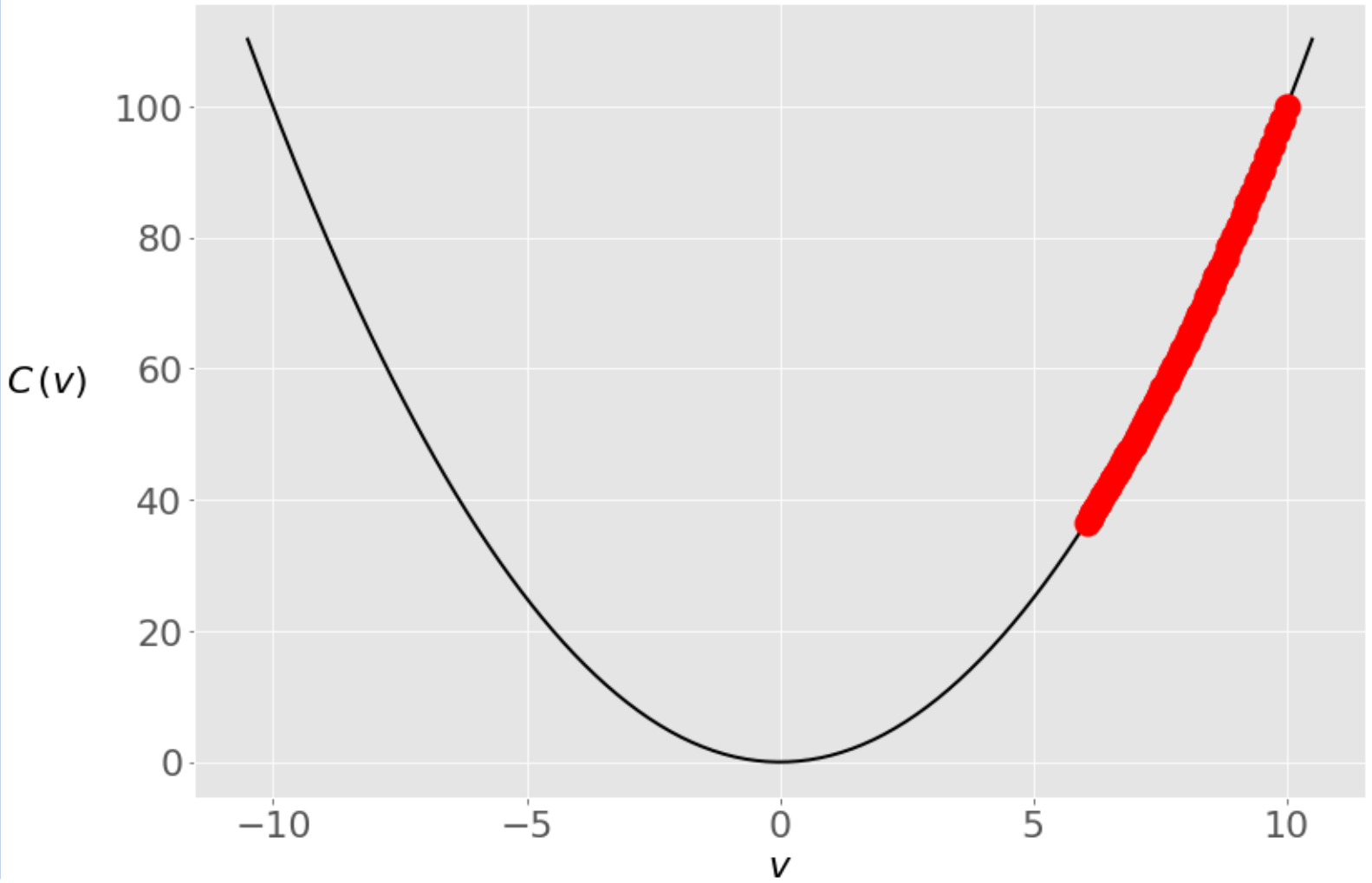


Learning Rate Impact

- Small learning rates can result in very slow convergence.
- If the number of iterations is limited, then the algorithm may return before the minimum is found. Otherwise, the whole process might take an unacceptably large amount of time. To illustrate this, run gradient_descent() again, this time with a much smaller learning rate of 0.005

```
>>> gradient_descent(
...     gradient=lambda v: 2 * v, start=10.0, learn_rate=0.005)
6.050060671375367
```

- The result is now 6.05, which is nowhere near the true minimum of zero.
- This is because the changes in the vector are very small due to the small learning rate



Learning Rate Impact

- The search process starts at v = 10 as before, but it can't reach zero in fifty iterations. However, with a hundred iterations, the error will be much smaller, and with a thousand iterations, you'll be very close to zero

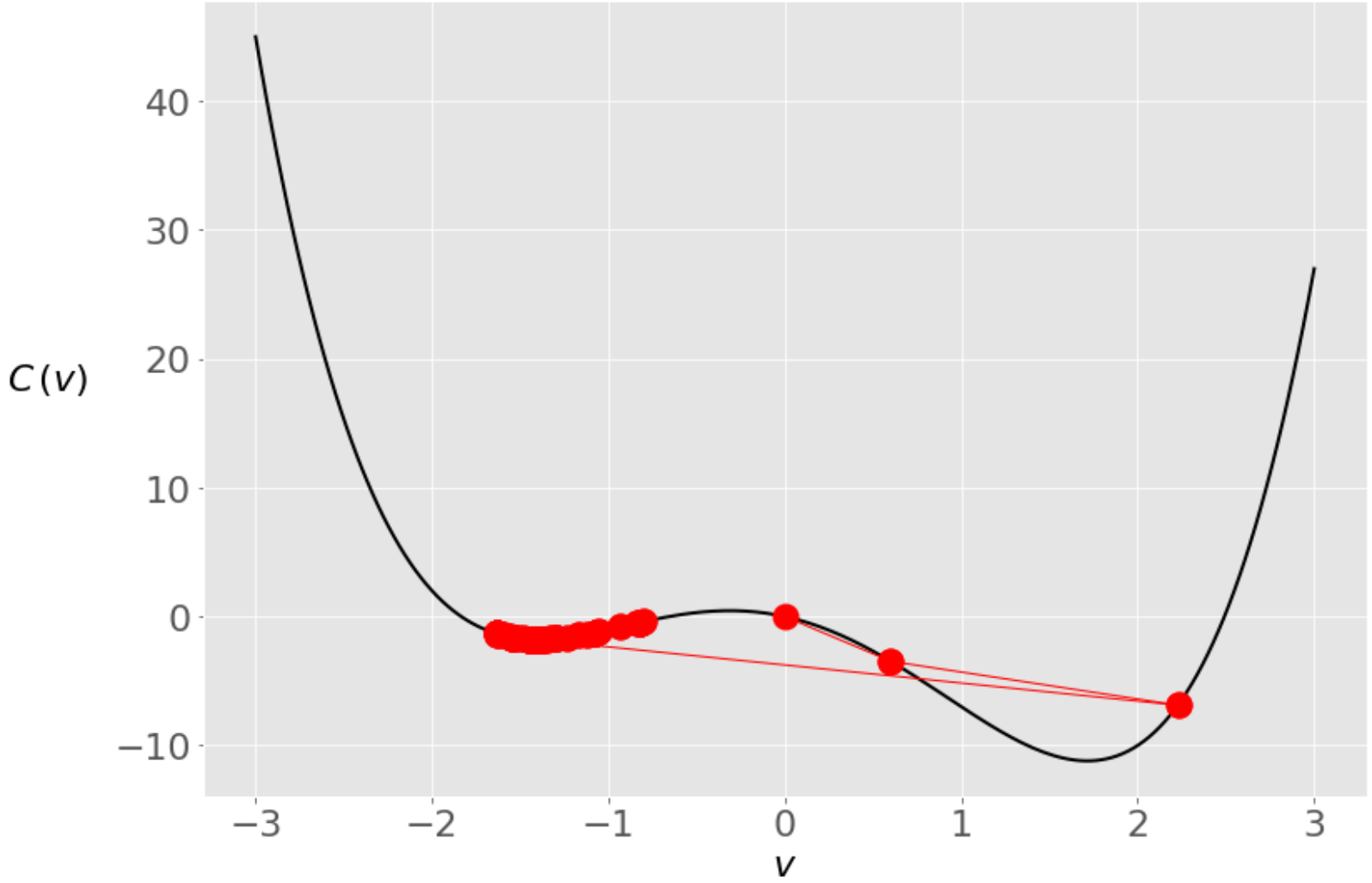
```
>>> gradient_descent(
...     gradient=lambda v: 2 * v, start=10.0, learn_rate=0.005, n_iter=100)
3.660323412732294
>>> gradient_descent(
...     gradient=lambda v: 2 * v, start=10.0, learn_rate=0.005, n_iter=1000)
0.00043171241741065828
>>> gradient_descent(
...     gradient=lambda v: 2 * v, start=10.0, learn_rate=0.005, n_iter=2000)
9.952518849647663e-05
```

Local minima / saddle points

- Nonconvex functions might have local minima or saddle points where the algorithm can get trapped. In such situations, your choice of learning rate or starting point can make the difference between finding a local minimum and finding the global minimum.
- Consider the function v^4 - $5v^2$ - $3v$. It has a global minimum in $v \approx 1.7$ and a local minimum in $v \approx -1.42$. The gradient of this function is $4v^3 - 10v - 3$.

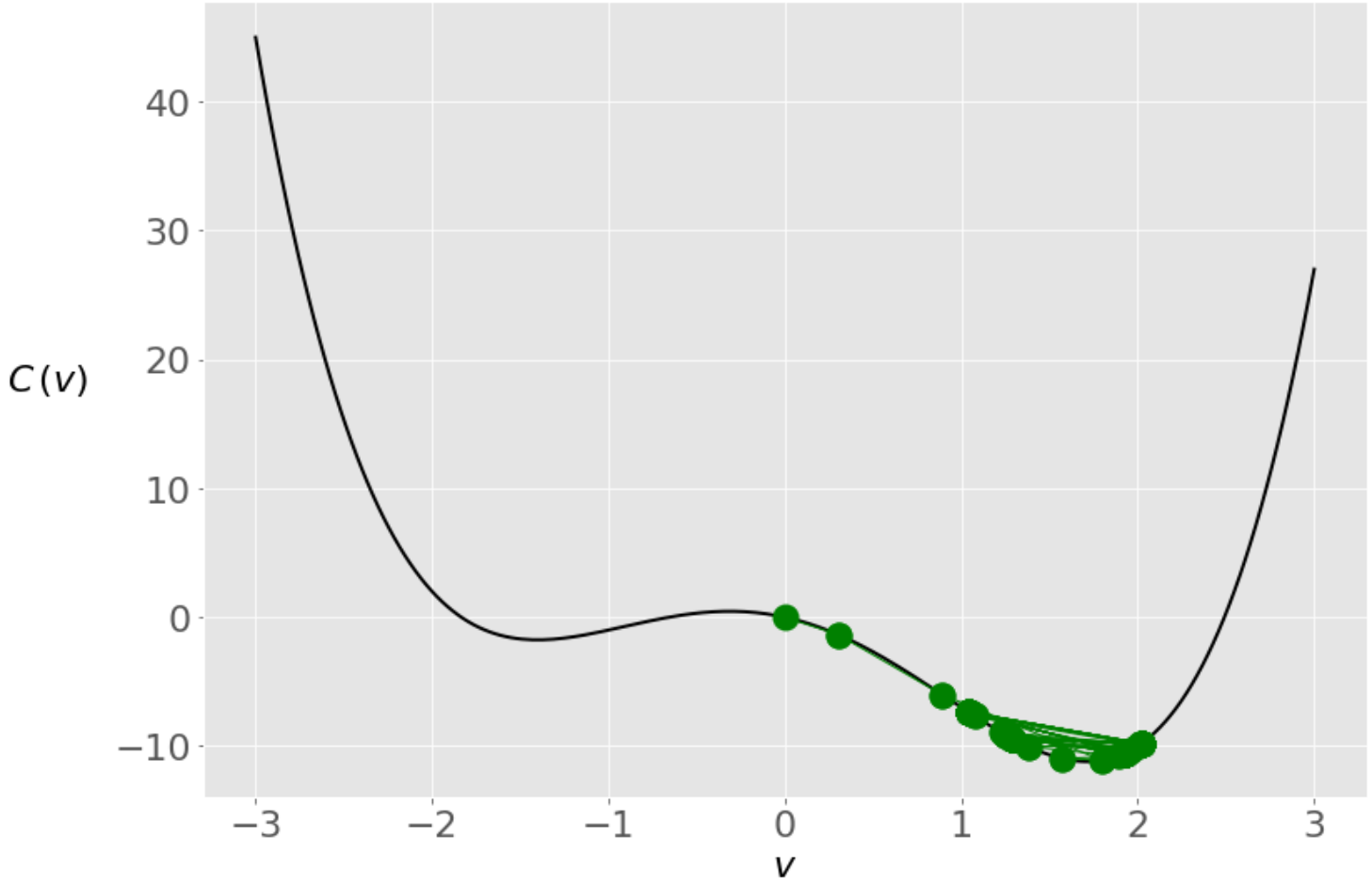
```
>>> gradient_descent(
...     gradient=lambda v: 4 * v**3 - 10 * v - 3, start=0,  learn_rate=0.2)
-1.4207567437458342
```

- You started at zero this time, and the algorithm ended near the local minimum.
- During the first two iterations, your vector was moving toward the global minimum, but then it crossed to the opposite side and stayed trapped in the local minimum.
- You can prevent this with a smaller learning rate



```
>>> gradient_descent(
...     gradient=lambda v: 4 * v**3 - 10 * v - 3, start=0,  learn_rate=0.1)
-1.4207567437458342
```

- A lower learning rate prevents the vector from making large jumps, and in this case, the vector remains closer to the global optimum.
- Adjusting the learning rate is tricky. You can't know the best value in advance. There are many techniques and heuristics that try to help with this. In addition, machine learning practitioners often tune the learning rate during model selection and evaluation.
- Besides the learning rate, the starting point can affect the solution significantly, especially with nonconvex functions.



Application of the Gradient Descent Algorithm

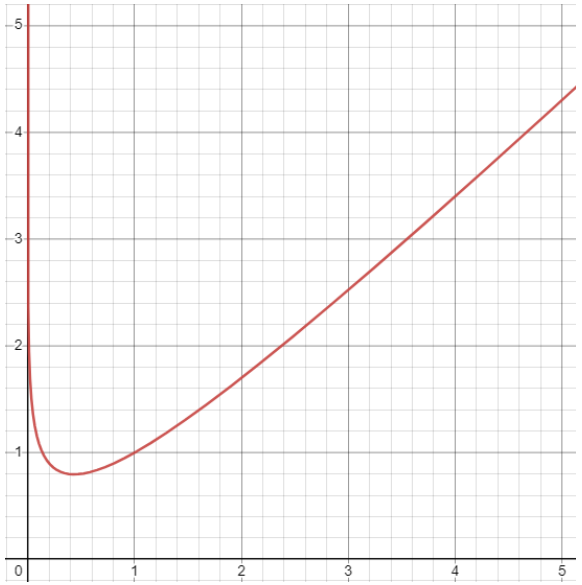
- Gradient Decent can be used in real-life machine learning problems like linear regression.
- We'll modify the code of gradient_descent() because we need the data from the observations to calculate the gradient.

Short Examples

- First, you'll apply gradient_descent() to another one-dimensional problem.
- Take the function $v - \log(v)$. The gradient of this function is $1 - 1/v$. With this information, you can find its minimum

  https://www.mathsisfun.com/calculus/derivatives-introduction.html

```
>>> gradient_descent(gradient=lambda v: 1 - 1 / v, start=2.5, learn_rate=0.5)
1.0000011077232125
```



$v_1^2 + v_2^4$

- $v_1^2 + v_2^4$ that has the gradient vector $(2v_1, 4v_2^3)$

```
>>> gradient_descent(
...     gradient=lambda v: np.array([2 * v[0], 4 * v[1]**3]),
...     start=np.array([1.0, 1.0]), learn_rate=0.2, tolerance=1e-08)
array([8.08281277e-12, 9.75207120e-02])
```

in this case, your gradient function returns an array, and the start value is an array, so you get an array as the result. The resulting values are almost equal to zero, so you can say that gradient_descent() correctly found that the minimum of this function is at $v_1 = v_2 = 0$.

Ordinary Least Squares

- As you've already learned, linear regression and the ordinary least squares method start with the observed values of the inputs $\mathbf{x} = (x_1, \ldots, x_r)$ and outputs y. They define a linear function $f(\mathbf{x}) = b_0 + b_1 x_1 + \cdots + b_r x_r$, which is as close as possible to y.
- This is an optimization problem. It finds the values of weights $b_0, b_1, \ldots, b_r$ that minimize the sum of squared residuals SSR = $\Sigma_i (y_i - f(\mathbf{x}_i))^2$ or the mean squared error MSE = SSR / n. Here, n is the total number of observations and $i = 1, \ldots, n$.
- You can also use the cost function C = SSR / $(2n)$, which is mathematically more convenient than SSR or MSE.

Ordinary Least Squares

- First, you need calculus to find the gradient of the cost function $C = \Sigma_i(y_i - b_0 - b_1 x_i)^2 / (2n)$. Since you have two decision variables, b_0 and b_1, the gradient ∇C is a vector with two components:
  1. $\partial C/\partial b_0 = (1/n)\ \Sigma_i(b_0 + b_1 x_i - y_i) = \text{mean}(b_0 + b_1 x_i - y_i)$
  2. $\partial C/\partial b_1 = (1/n)\ \Sigma_i(b_0 + b_1 x_i - y_i)\ x_i = \text{mean}((b_0 + b_1 x_i - y_i)\ x_i)$
- You need the values of x and y to calculate the gradient of this cost function. Your gradient function will have as inputs not only b_0 and b_1 but also x and y.

```
def ssr_gradient(x, y, b):
    res = b[0] + b[1] * x - y
    return res.mean(), (res * x).mean()  # .mean() is a method of np.ndarray
```

ssr_gradient() takes the arrays x and y, which contain the observation inputs and outputs, and the array b that holds the current values of the decision variables b_0 and b_1 , and returns the pair of values of $\partial C/\partial b_0$ and $\partial C/\partial b_1$

Small adjustments

- Add x and y as the parameters of gradient_descent() on line 4.
- Provide x and y to the gradient function and make sure you convert your gradient tuple to a NumPy array on line 8.

```
import numpy as np

def gradient_descent(
    gradient, x, y, start, learn_rate=0.1, n_iter=50, tolerance=1e-06
):
    vector = start
    for _ in range(n_iter):
        diff = -learn_rate * np.array(gradient(x, y, vector))
        if np.all(np.abs(diff) <= tolerance):
            break
        vector += diff
    return vector
```

Call the gradient function

```
>>> x = np.array([5, 15, 25, 35, 45, 55])
>>> y = np.array([5, 20, 14, 32, 22, 38])

>>> gradient_descent(
...     ssr_gradient, x, y, start=[0.5, 0.5], learn_rate=0.0008,
...     n_iter=100_000
... )
array([5.62822349, 0.54012867])
```

Results

- The result is an array with two values that correspond to the decision variables: $b_0 = 5.63$ and $b_1 = 0.54$.
- The best regression line is $f(x) = 5.63 + 0.54x$.
- As in the previous examples, this result heavily depends on the learning rate. You might not get such a good result with too low or too high of a learning rate.

Improvement of the Code

- You can make gradient_descent() more robust, comprehensive, and better-looking without modifying its core functionality

```
import numpy as np

def gradient_descent(
    gradient, x, y, start, learn_rate=0.1, n_iter=50,
tolerance=1e-06,
    dtype="float64"
):
    # Checking if the gradient is callable
    if not callable(gradient):
        raise TypeError("'gradient' must be callable")

    # Setting up the data type for NumPy arrays
    dtype_ = np.dtype(dtype)

    # Converting x and y to NumPy arrays
    x, y = np.array(x, dtype=dtype_), np.array(y,
dtype=dtype_)
    if x.shape[0] != y.shape[0]:
        raise ValueError("'x' and 'y' lengths do not match")
```

```
    # Initializing the values of the variables
    vector = np.array(start, dtype=dtype_)

    # Setting up and checking the learning rate
    learn_rate = np.array(learn_rate, dtype=dtype_)
    if np.any(learn_rate <= 0):
        raise ValueError("'learn_rate' must be greater than
zero")

    # Setting up and checking the maximal number of
iterations
    n_iter = int(n_iter)
    if n_iter <= 0:
        raise ValueError("'n_iter' must be greater than zero")
```

```
    # Setting up and checking the tolerance
    tolerance = np.array(tolerance, dtype=dtype_)
    if np.any(tolerance <= 0):
        raise ValueError("'tolerance' must be greater than
zero")

    # Performing the gradient descent loop
    for _ in range(n_iter):
        # Recalculating the difference
        diff = -learn_rate * np.array(gradient(x, y, vector),
dtype_)

        # Checking if the absolute difference is small enough
        if np.all(np.abs(diff) <= tolerance):
            break

        # Updating the values of the variables
        vector += diff

    return vector if vector.shape else vector.item()
```

Changes

- check if gradient is a Python callable object and whether it can be used as a function. If not, then the function will raise a TypeError.
- sets an instance of numpy.dtype, which will be used as the data type for all arrays throughout the function.
- takes the arguments x and y and produces NumPy arrays with the desired data type. The arguments x and y can be lists, tuples, arrays, or other sequences.
- compare the sizes of x and y. This is useful because you want to be sure that both arrays have the same number of observations. If they don't, then the function will raise a ValueError.
- converts the argument start to a NumPy array. This is an interesting trick: if start is a Python scalar, then it'll be transformed into a corresponding NumPy object (an array with one item and zero dimensions). If you pass a sequence, then it'll become a regular NumPy array with the same number of elements.
- does the same thing with the learning rate. This can be very useful because it enables you to specify different learning rates for each decision variable by passing a list, tuple, or NumPy array to gradient_descent().
- check if the learning rate value (or values for all variables) is greater than zero.
- similarly set n_iter and tolerance and check that they are greater than zero.
- conveniently returns the resulting array if you have several decision variables or a Python scalar if you have a single variable.

Stochastic Gradient Descent Algorithms

- **Stochastic gradient descent algorithms** are a modification of gradient descent. In stochastic gradient descent, you calculate the gradient using just a random small part of the observations instead of all of them. In some cases, this approach can reduce computation time.
- **Online stochastic gradient descent** is a variant of stochastic gradient descent in which you estimate the gradient of the cost function for each observation and update the decision variables accordingly. This can help you find the global minimum, especially if the objective function is convex.
- **Batch stochastic gradient descent** is somewhere between ordinary gradient descent and the online method. The gradients are calculated, and the decision variables are updated iteratively with subsets of all observations, called **minibatches**. This variant is very popular for training neural networks.

Minibatches in Stochastic Gradient Descent

- Stochastic gradient descent starts with an initial vector of decision variables and updates it through several iterations. The difference between the two is in what happens inside the iterations:
  - Stochastic gradient descent randomly divides the set of observations into minibatches.
  - For each minibatch, the gradient is computed and the vector is moved.
  - Once all minibatches are used, you say that the iteration, or **epoch**, is finished and start the next one.
- This algorithm randomly selects observations for minibatches, so you need to simulate this random (or pseudorandom) behavior. You can do that with random number generation. Python has the built-in random module, and NumPy has its own random generator. The latter is more convenient when you work with arrays.
- You'll create a new function called sgd() that is very similar to gradient_descent() but uses randomly selected minibatches to move along the search space:

Python

```python
import numpy as np

def sgd(
    gradient, x, y, start, learn_rate=0.1, batch_size=1, n_iter=50
    tolerance=1e-06, dtype="float64", random_state=None
):
    # Checking if the gradient is callable
    if not callable(gradient):
        raise TypeError("'gradient' must be callable")

    # Setting up the data type for NumPy arrays
    dtype_ = np.dtype(dtype)

    # Converting x and y to NumPy arrays
    x, y = np.array(x, dtype=dtype_), np.array(y, dtype=dtype_)
    n_obs = x.shape[0]
    if n_obs != y.shape[0]:
        raise ValueError("'x' and 'y' lengths do not match")
    xy = np.c_[x.reshape(n_obs, -1), y.reshape(n_obs, 1)]

    # Initializing the random number generator
    seed = None if random_state is None else int(random_state)
    rng = np.random.default_rng(seed=seed)

    # Initializing the values of the variables
    vector = np.array(start, dtype=dtype_)
    vector = np.array(start, dtype=dtype_)

    # Setting up and checking the learning rate
    learn_rate = np.array(learn_rate, dtype=dtype_)
    if np.any(learn_rate <= 0):
        raise ValueError("'learn_rate' must be greater than zero")

    # Setting up and checking the size of minibatches
    batch_size = int(batch_size)
    if not 0 < batch_size <= n_obs:
        raise ValueError(
            "'batch_size' must be greater than zero and less than "
            "or equal to the number of observations"
        )

    # Setting up and checking the maximal number of iterations
    n_iter = int(n_iter)
    if n_iter <= 0:
        raise ValueError("'n_iter' must be greater than zero")

    # Setting up and checking the tolerance
    tolerance = np.array(tolerance, dtype=dtype_)
    if np.any(tolerance <= 0):
        raise ValueError("'tolerance' must be greater than zero")

```

```
    # Performing the gradient descent loop
    for _ in range(n_iter):
        # Shuffle x and y
        rng.shuffle(xy)

        # Performing minibatch moves
        for start in range(0, n_obs, batch_size):
            stop = start + batch_size
            x_batch, y_batch = xy[start:stop, :-1], xy[start:stop, -1:]

            # Recalculating the difference
            grad = np.array(gradient(x_batch, y_batch, vector), dtype_)
            diff = -learn_rate * grad

            # Checking if the absolute difference is small enough
            if np.all(np.abs(diff) <= tolerance):
                break

            # Updating the values of the variables
            vector += diff

    return vector if vector.shape else vector.item()
```

The inner for loop is repeated for each minibatch. The main difference from the ordinary gradient descent is that the gradient is calculated for the observations from a minibatch (x_batch and y_batch) instead of for all observations (x and y).

test the implementation of stochastic gradient descent

```
>>> sgd(
...     ssr_gradient, x, y, start=[0.5, 0.5], learn_rate=0.0008,
...     batch_size=3, n_iter=100_000, random_state=0
... )
array([5.63093736, 0.53982921])
```

Momentum in Stochastic Gradient Descent

- You can use momentum to correct the effect of the learning rate. The idea is to remember the previous update of the vector and apply it when calculating the next one. You don’t move the vector exactly in the direction of the negative gradient, but you also tend to keep the direction and magnitude from the previous move.
- The parameter called the decay rate or decay factor defines how strong the contribution of the previous update is. To include the momentum and the decay rate, you can modify sgd() by adding the parameter decay_rate and use it to calculate the direction and magnitude of the vector update (diff)

```python
import numpy as np

def sgd(
    gradient, x, y, start, learn_rate=0.1, decay_rate=0.0, batch_size=1,
    n_iter=50, tolerance=1e-06, dtype="float64", random_state=None
):
    # Checking if the gradient is callable
    if not callable(gradient):
        raise TypeError("'gradient' must be callable")

    # Setting up the data type for NumPy arrays
    dtype_ = np.dtype(dtype)

    # Converting x and y to NumPy arrays
    x, y = np.array(x, dtype=dtype_), np.array(y, dtype=dtype_)
    n_obs = x.shape[0]
    if n_obs != y.shape[0]:
        raise ValueError("'x' and 'y' lengths do not match")
    xy = np.c_[x.reshape(n_obs, -1), y.reshape(n_obs, 1)]

    # Initializing the random number generator
    seed = None if random_state is None else int(random_state)
    rng = np.random.default_rng(seed=seed)

    # Initializing the values of the variables
    vector = np.array(start, dtype=dtype_)

    # Setting up and checking the learning rate
    learn_rate = np.array(learn_rate, dtype=dtype_)
    if np.any(learn_rate <= 0):
        raise ValueError("'learn_rate' must be greater than zero")

    # Setting up and checking the decay rate
    decay_rate = np.array(decay_rate, dtype=dtype_)
    if np.any(decay_rate < 0) or np.any(decay_rate > 1):
        raise ValueError("'decay_rate' must be between zero and one")

    # Setting up and checking the size of minibatches
    batch_size = int(batch_size)
    if not 0 < batch_size <= n_obs:
        raise ValueError(
            "'batch_size' must be greater than zero and less than "
            "or equal to the number of observations"
        )

    # Setting up and checking the maximal number of iterations
    n_iter = int(n_iter)
    if n_iter <= 0:
        raise ValueError("'n_iter' must be greater than zero")

    # Setting up and checking the tolerance
    tolerance = np.array(tolerance, dtype=dtype_)
    if np.any(tolerance <= 0):
        raise ValueError("'tolerance' must be greater than zero")
```

```
55
56      # Setting the difference to zero for the first iteration
57      diff = 0
58
59      # Performing the gradient descent loop
60      for _ in range(n_iter):
61          # Shuffle x and y
62          rng.shuffle(xy)
63
64          # Performing minibatch moves
65          for start in range(0, n_obs, batch_size):
66              stop = start + batch_size
67              x_batch, y_batch = xy[start:stop, :-1], xy[start:stop, -1:]
68
69              # Recalculating the difference
70              grad = np.array(gradient(x_batch, y_batch, vector), dtype_)
71              diff = decay_rate * diff - learn_rate * grad
72
73              # Checking if the absolute difference is small enough
74              if np.all(np.abs(diff) <= tolerance):
75                  break
76
77              # Updating the values of the variables
78              vector += diff
79
80      return vector if vector.shape else vector.item()
```

You recalculate diff with the learning rate and gradient but also add the product of the decay rate and the old value of diff. Now diff has two components:

1. decay_rate * diff is the momentum, or impact of the previous move.
2. -learn_rate * grad is the impact of the current gradient.
3. The decay and learning rates serve as the weights that define the contributions of the two.

Gradient Descent in Keras and TensorFlow

- Stochastic gradient descent is widely used to train neural networks. The libraries for neural networks often have different variants of optimization algorithms based on stochastic gradient descent, such as:
  - Adam
  - Adagrad
  - Adadelta
  - RMSProp
- These optimization libraries are usually called internally when neural network software is trained. However, you can use them independently as well

```
>>> import tensorflow as tf

>>> # Create needed objects
>>> sgd = tf.keras.optimizers.SGD(learning_rate=0.1, momentum=0.9)
>>> var = tf.Variable(2.5)
>>> cost = lambda: 2 + var ** 2

>>> # Perform optimization
>>> for _ in range(100):
...     sgd.minimize(cost, var_list=[var])

>>> # Extract results
>>> var.numpy()
-0.007128528
>>> cost().numpy()
2.0000508
```

In this example, you first import tensorflow and then create the object needed for optimization:

1. sgd is an instance of the stochastic gradient descent optimizer with a learning rate of 0.1 and a momentum of 0.9.
2. var is an instance of the decision variable with an initial value of 2.5.
3. cost is the cost function, which is a square function in this case.

Gradient Descent in Keras and TensorFlow

- The main part of the code is a for loop that iteratively calls .minimize() and modifies var and cost. Once the loop is exhausted, you can get the values of the decision variable and the cost function with .numpy().
- You can find more information on these algorithms in the Keras and TensorFlow documentation. The article An overview of gradient descent optimization algorithms offers a comprehensive list with explanations of gradient descent variants.