3
Optimization via Gradient Descent

This chapter discusses how to train model parameters through *optimization* techniques that help find the best (or fairly good) model that has low training loss. We assume that you have seen simple root-finding techniques in high school or in calculus. Optimization in machine learning often uses a procedure called *gradient descent*. This chapter assumes your knowledge of basic multivariable calculus. If you have not taken a course in multivariable calculus, read Chapter 19 to familiarize yourself with the basic definitions.

3.1 Gradient Descent

In general, an ML model has an associated loss function. The "best" model is the one that minimizes the training loss. In most cases, it is impossible or difficult to find the minimum analytically; instead, we use a numerical method called the *gradient descent algorithm* to find the (approximate) optimum.

3.1.1 Univariate Example

Let's start with an univariate example to motivate the topic. Let $f(w) = 4w^2 - 6w - 9$ be a quadratic function. Figure 3.1 shows the graph of this function.

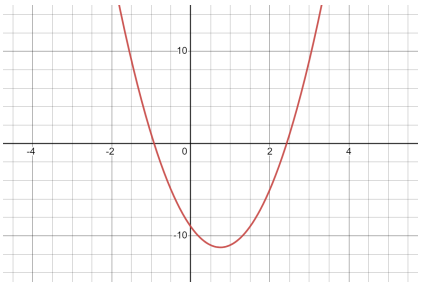

Figure 3.1: The graph of $f(w) = 4w^2 - 6w - 9$

Let's say that f attains its minimum at some point $w = w^*$. How

should we find the value of w^*? Here is an idea. Let's start from some random point on the curve and "walk down" the curve.

Notice from the graph that $f'(w^*) = 0$. Also, f is decreasing (*i.e.*, $f'(w) < 0$) when $w < w^*$ and increasing (*i.e.*, $f'(w) > 0$) when $w > w^*$. So if we examine a point w and find that $f'(w) = 0$, then we have arrived at our minimum. If $f'(w) > 0$, then we are currently on the right side of the minimum, so we need to *decrease* w. On the other hand, if $f'(w) < 0$, then we need to *increase* w.

For example, we start with the point $w = 0$. Since $f'(w) = -6 < 0$, we know that we are on the left side of the minimum, so we update $w \leftarrow 1$. Since $f'(w) = 2 > 0$, we are now on the right side of the minimum, so we update $w \leftarrow \frac{1}{2}$. When we iterate this process, we hope that we eventually slide down to the bottom of the curve. Observe that the change of the value of w has the opposite sign from $f'(w)$ at that point. That is, for each step of this iteration, we can always find a $\eta > 0$ which decreases the value of w when

$$w \leftarrow w - \eta f'(w)$$

This is not a mere coincidence — a similar result holds for a multivariate function.

3.1.2 *Gradient Descent (GD)*

Let $f : \mathbb{R}^d \to \mathbb{R}$ be a multivariate function. If we want to "walk down" the curve of f as in the univariate case, we need to find a direction from the current point $\vec{\mathbf{w}}$ that *decreases* f.

A generalization of the Taylor expansion in the multivariable setting shows that the value of f in a small neighborhood around $\vec{\mathbf{x}} = (x_1, x_2, \ldots, x_d)$ can be approximated as a linear function in terms of the gradient.

$$f(\vec{\mathbf{w}} + \vec{\mathbf{h}}) \approx f(\vec{\mathbf{w}}) + \nabla f(\vec{\mathbf{w}}) \cdot \vec{\mathbf{h}}$$

where $\vec{\mathbf{h}} \in \mathbb{R}^d$ is small enough (*i.e.*, $\left\|\vec{\mathbf{h}}\right\| \approx 0$).

If ∇f is nonzero and we choose $\vec{\mathbf{h}} = -\eta \nabla f$ where η is a sufficiently small positive number, then

$$f(\vec{\mathbf{w}} - \eta \nabla f) \approx f(\vec{\mathbf{w}}) - \eta \|\nabla f\|_2^2$$

Since $\|\nabla f\|_2^2$ is positive, being the squared length of the vector ∇f, we conclude that the update $\vec{\mathbf{w}} \leftarrow \vec{\mathbf{w}} - \eta \nabla f$ causes a decrease in value of f. [1] This discussion motivates the *gradient descent algorithm*, which iteratively decreases the value of f until $\nabla f \approx 0$.

[1] In fact, the gradient ∇f is known as the direction of steepest increase of f. Hence, the opposite direction $-\nabla f$ is the direction of steepest decrease of f.

Definition 3.1.1 (Gradient Descent)**.** *Gradient descent is an iterative algorithm that updates the weight vector* $\vec{\mathbf{w}}$ *with the following rule:*

$$\vec{\mathbf{w}} \leftarrow \vec{\mathbf{w}} - \eta \nabla f(\vec{\mathbf{w}}) \tag{3.1}$$

where $\eta > 0$ is a sufficiently small positive constant, called the ***learning rate*** *or* ***step size****.*

We illustrate with an example.

Example 3.1.2. *Let $f(w_1, w_2) = (w_1^2 + w_2^2)^4 - 7(w_1^2 + w_2^2)^3 + 13(w_1^2 + w_2^2)^2$. From Figure 3.2, we see that it attains a global minimum at $(0, 0)$. The partial derivatives of f can be calculated as:*

$$\frac{\partial f}{\partial w_1} = 2w_1(w_1^2 + w_2^2)(4(w_1^2 + w_2^2)^2 - 21(w_1^2 + w_2^2) + 26)$$
$$\frac{\partial f}{\partial w_2} = 2w_2(w_1^2 + w_2^2)(4(w_1^2 + w_2^2)^2 - 21(w_1^2 + w_2^2) + 26)$$

Now imagine initiating the gradient descent algorithm from the point $(0.5, 1)$ where the gradient vector is $(7.5, 15)$. One iteration of gradient descent with $\eta = 0.01$ would move from $(0.5, 1)$ to $(0.425, 0.85)$. The gradient vector at $(0.425, 0.85)$ is $(7.90, 15.81)$ and the next iteration of GD will move the point from $(0.425, 0.85)$ to $(0.35, 0.69)$. After 200 iterations, the algorithm moves the point to $(0.03, 0.06)$, which is very close to the global minimum of f.

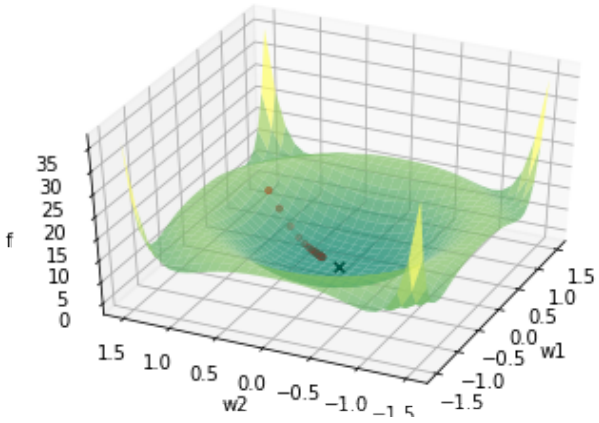

Figure 3.2: The graph of $f(w_1, w_2) = (w_1^2 + w_2^2)^4 - 7(w_1^2 + w_2^2)^3 + 13(w_1^2 + w_2^2)^2$. The function attains a global minimum at $(0, 0)$.

3.1.3 *Learning Rate (LR)*

Choosing an appropriate learning rate is crucial for GD. Figure 3.3 shows the result of two iterations of gradient descent with a different learning rate. On the left, we see the result when η is too small. The change of w is too small, and the loss function converges to the minimum very slowly. On the right, we see the result when η is too big. The change of w is too large, and the algorithm "shoots past" the minimum. If η is even larger, the algorithm may even fail to converge to the minimum.

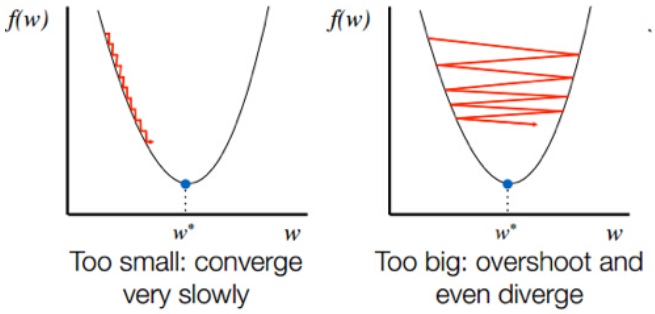

Figure 3.3: Two iterations of gradient descent with different learning rates.

The natural question to ask is: what is the appropriate learning rate? There is some theory, and the best setting is known in some cases. But in general, it has to be set by trial and error, especially for non-convex loss functions. For instance, we start with some learning rate, say 0.5 and decrease η by $\frac{1}{2}$ if we do not observe a steady decrease in the training loss. Such heuristics are called *training schedules* and they are derived via trial and error on that *particular* dataset and model. [2]

3.1.4 *Non-convex Functions*

For convex functions that are "bowl shaped," gradient descent with a small enough learning rate provably converges to the minimum solution. But for non-convex functions, the best we can hope for is converging to a point where $\nabla f = 0$. [3] Finding the global minimum of a non-convex function is NP-hard in the worst case.

In practice, loss functions are non-convex and have multiple local minima. Then, the gradient descent algorithm may converge to a different local minimum based on the initialization of the parameter vector $\vec{\mathbf{w}}$.

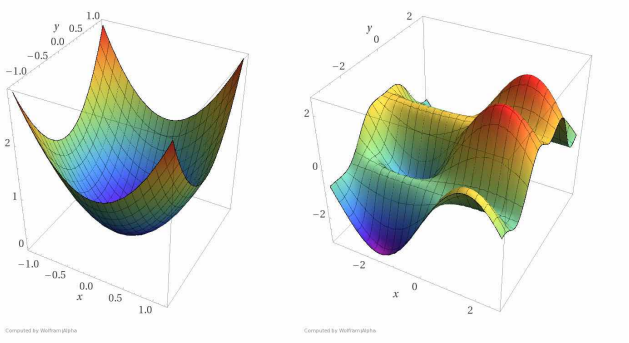

Figure 3.4: An example of a convex and a non-convex function in two variables. For non-convex functions, GD will reach a stationary point, where the gradient is zero. Figure from https://www.kdnuggets.com/2016/12/hard-thing-about-deep-learning.html.

Example 3.1.3. *Consider the function* $f(w) = \frac{1}{3}w^4 - \frac{1}{2}w^3 - w^2 + w$*, which has two local minima at* $(-1, -1)$ *and* $(2, -1)$*. As seen in Figure 3.5, the*

[2] Constants whose values are decided by trial and error based on dataset and model are called *hyperparameters*. Modern ML models have several hyperparameters. Often optimization packages will suggest a default value and a fine-tuning method.

[3] Points where the gradient is zero are called *stationary points*, which include local minima, local maxima, and saddle points. It is possible for a GD algorithm to terminate at a saddle point, instead of the intended local minimum. There is advanced theory on how to escape saddle points, which will not be covered in this course.

local minimum that the gradient descent algorithm outputs depends on the initial point.

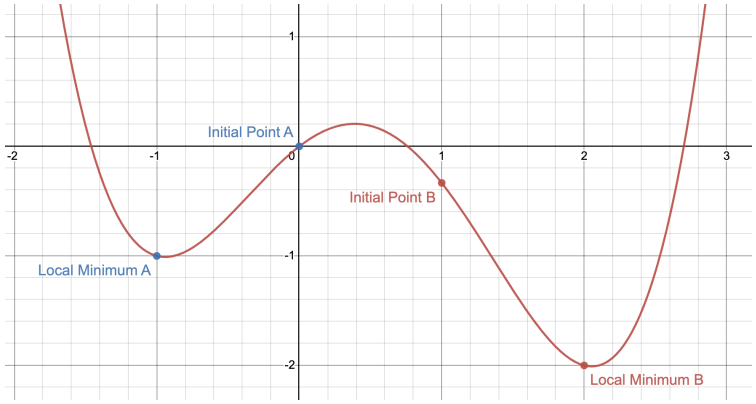

Figure 3.5: The graph of $f(w) = \frac{1}{3}w^4 - \frac{1}{2}w^3 - w^2 + w$ with two local minima.

3.2 *Implications of the Linearity of a Gradient*

The fact that gradient is a linear operator (*i.e.*, $\nabla(f_1 + f_2) = \nabla f_1 + \nabla f_2$) has great practical importance in machine learning.

Just like in (1.4), the training loss of a machine learning model is usually defined as the average (or the sum) of the loss on individual training data points. By the linearity of gradient, the gradient of the entire loss can be found by taking the sum of the gradient of the loss on individual data points.

3.2.1 *Stochastic Gradient Descent*

Since computing the gradient of the loss involves some computation on each of the data points, the computation can be quite slow for today's large data sets, which can contain millions of data points. A simple workaround is to estimate the gradient at each step by randomly sampling k data points and averaging the corresponding loss gradients. This is very analogous to opinion polls, which can also be seen as sampling from a distribution on vectors and using the average of the sample as a substitute for the population average. This algorithm is called *Stochastic Gradient Descent (SGD)*. [4] This technique works for two reasons: (1) all training data points are assumed to be sampled from the same distribution; (2) the overall training loss is just the sum/average of loss for individual data points.

[4] Some authors call this the *Batch SGD* and use the name *SGD* only for the case where $k = 1$.

3.2.2 *Mini-batch Stochastic Gradient Descent*

Today, large scale machine learning is done using special-purpose processors called *Graphical Processing Units (GPUs)*. [5] These highly

[5] As the name suggests, these were originally developed for computer graphics operations, which were most oftenly used in computer games. Around 2012, deep learning experts realized their usefulness for deep learning. At that time, writing code for GPUs was extremely difficult, but today's environments have made this much easier.

specialized architectures have the ability to perform fast parallel computations. To exploit these special capabilities, a special variant of SGD — Mini-batch SGD — can be used. Here the dataset is randomly partitioned into mini batches whose size is dictated by the degree of parallelism available in the GPU, usually a power of 2, such as 256. The members of each batch are loaded onto a different processor. Together the processors compute the gradient for one mini-batch in one go, add up the gradients to perform a single iteration for the gradient descent. Then they move on to the next batch, perform another update step, and so on.

3.2.3 *Federated Learning*

This is a conceptual framework for training an ML model on data belonging to different parties, some of whom do not wish to hand the data over to a central server. Consider the following two examples:

1. Hospitals who wish to train an ML model on their pooled data, but who are forbidden by privacy laws to hand the data to other organizations.

2. Owners of Internet of Things (IoT) devices, who wish to benefit from training on their data but do not wish to submit the data.

In Federated Learning, the model is trained at a central server, whereas data remains with the data owners, who actively participate in the training. Users retrieve the current model parameters from the server and calculate the gradients *locally*. They send only the gradients, but not the data, to the server, and the overall gradient is calculated at the server as the weighted sum (or average) of the user gradients.

3.3 *Regularizers*

This section describes *regularization*, a useful idea that often improves generalization of the model. The main idea is that instead of minimizing the training loss function $\ell(\vec{\mathbf{w}})$, we minimize the function

$$\ell(\vec{\mathbf{w}}) + \lambda R(\vec{\mathbf{w}}) \tag{3.2}$$

where $\lambda > 0$ is a constant and $R(\vec{\mathbf{w}})$ is some non-negative function. $R(\vec{\mathbf{w}})$ is called a *regularizer* or sometimes *penalty*. We refer to (3.2) as the *regularized loss function*.

The most commonly used regularizer is the ℓ_2 regularizer where the squared ℓ_2 norm $R(\vec{\mathbf{w}}) = \|\vec{\mathbf{w}}\|_2^2$ of the weight vector is used.

Example 3.3.1. *Recall the sentiment prediction model using least squares loss. Suppose the training data consists of two data points:* $(\vec{\mathbf{x}}^1, y^1) = ((1,0,1),-1)$ *and* $(\vec{\mathbf{x}}^2, y^2) = ((1,1,0),+1)$. *Then the least squares loss, without any regularizer, can be written as*

$$\frac{1}{2}((-1-(w_0+w_2))^2+(1-(w_0+w_1))^2) \tag{3.3}$$

A little thought suggests that the minimum value of this loss is 0 *provided there exists* (w_0, w_1, w_2) *such that*

$$(-1-(w_0+w_2))^2 = 0 = (1-(w_0+w_1))^2.$$

You can verify that infinitely many solutions exist: all $\vec{\mathbf{w}}^* = (w_0, w_1, w_2)$ *that lie on the line* $(0,1,-1)+t(1,-1,-1)$ *where* $t \in \mathbb{R}$. *In other words, the loss has infinitely many minimizers.*

Now if impose an ℓ_2 *regularizer, the loss becomes*

$$\frac{1}{2}((-1-(w_0+w_2))^2+(1-(w_0+w_1))^2)+\lambda(w_0^2+w_1^2+w_2^2) \tag{3.4}$$

Any minimizer of this loss must make the gradient zero. In other words, the minimizer will satisfy the following system of linear equations:

$$\begin{cases} (2+2\lambda)w_0 + w_1 + w_2 = 0 \\ w_0 + (1+2\lambda)w_1 = 1 \\ w_0 + (1+2\lambda)w_2 = -1 \end{cases}$$

You can verify that $\vec{\mathbf{w}}^{**} = \left(0, \frac{1}{1+2\lambda}, -\frac{1}{1+2\lambda}\right)$ *is the unique minimizer for any* $\lambda > 0$. *For a sufficiently small value of* λ, *the corresponding* $\vec{\mathbf{w}}^{**}$ *is close enough to the line* $(0,1,-1)+t(1,-1,-1)$. *That is, it has a non-zero training loss, but the value is very close to zero. Combined with the fact it has a relatively small norm,* $\vec{\mathbf{w}}^{**}$ *becomes the minimizer for the regularized loss.*

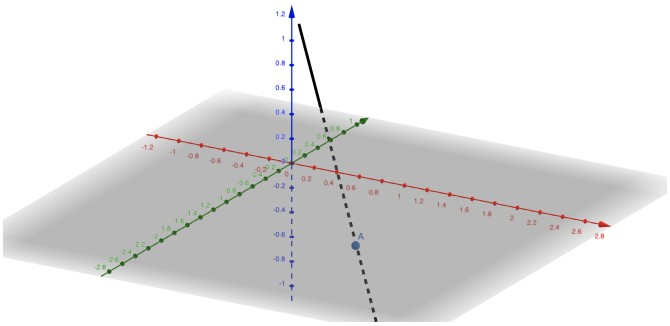

Figure 3.6: The graph of the line $(0,1,-1)+t(1,-1,-1)$ and the point $\vec{\mathbf{w}}^{**} = (0, \frac{1}{1+2\lambda}, -\frac{1}{1+2\lambda})$ when $\lambda = 0.01$

Note that if $\vec{\mathbf{w}}^*$ is the minimizer of $\ell(\vec{\mathbf{w}})$ and $\vec{\mathbf{w}}^{**}$ the minimizer of the regularized loss, then by definition of a minimizer, it always

holds that $\ell(\vec{\mathbf{w}}^*) \leq \ell(\vec{\mathbf{w}}^{**})$. In general, regularization ends up leading to training models with a *higher* value of $\ell(\vec{\mathbf{w}})$. This is considered acceptable because the models often generalize better. In other words, a slightly higher training loss is considered a price worth paying for a significantly lower test loss. This is illustrated by the example of sentiment prediction from Chapter 1. As hinted there, the results shown used a model trained with an ℓ_2 regularizer. The dataset involves 15k distinct words, so that is the number of model variables. There are 8k data points. Recall from Problem 1.2.5 that in such settings, there usually will exist a linear model that perfectly fits the data points. Indeed, we see in Table 3.1 that this is the case when we don't use a regularizer. However, using a regularizer prevents the model from perfectly fitting the training data. But the test loss drops tenfold with regularization.

	No regularizer	With ℓ_2-regularizer
Train MSE	0.0000	0.0727
Test MSE	7.9469	0.7523
Training accuracy	100.00%	99.55%
Test accuracy	61.67%	78.07%

Table 3.1: Training sentiment model on the SST with and without ℓ_2 regularizer.

3.3.1 *Effects of Regularization*

Here we briefly list some benefits of regularization.

1. Regularizers often help improve generalization. Above we saw a concrete example with the sentiment prediction model.

2. Adding a scalar multiple of $\|\vec{\mathbf{w}}\|_2^2$ to a function can speed up optimization by slightly reshaping the optimization landscape. The mathematical treatment of this is beyond the scope of this course.

3. Without a regularizer term, models such as logistic regression and soft-margin SVMs begin to lose their power. This will be explained when we discuss these models in Chapter 4.

3.3.2 *Why Does Regularization Help?*

The simplest answer is that we do not fully understand this concept yet. In this section, we present some intuitions derived from simple models, but keep in mind that these ideas might be misleading in more complicated models.

The usual explanation given is that the norm of the parameter vector controls the *expressiveness* or *complexity* of the model. Here "complexity" is being used in the sense of "complicatedness". By

trying to minimize loss as well as the norm of the parameter vector, the learned model tends to stay simple. [6] Whereas this discussion can be made fairly rigorous for linear models, it does not seem to apply to more complicated models; for instance, regularization often helps a lot in deep learning, but rigorous explanations appear to be at best incomplete and at worst incorrect there. [7]

Another explanation [8] is that a regularizer serves as a penalty for large weights and forces the model to choose smaller absolute values of parameters. According to this explanation, adding regularizers to a model penalizes higher-order terms or unnecessary variables and is able to avoid overfitting. Indeed, Figure 3.7 shows that the weights of the parameters in our sentiment model are significantly smaller when trained with a regularizer. But one lingering question with this explanation is: *How come attaching the **same** penalty to all variables forces the model to identify variables that are needed, and those that are not? What causes this disparate treatment of the variables?*

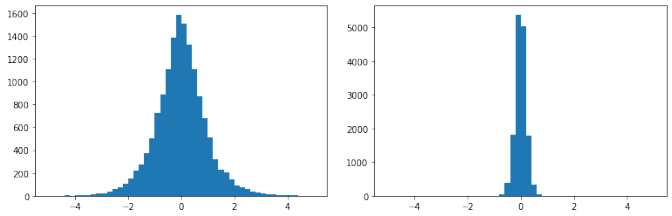

Figure 3.7: The histogram of weights of the parameters in the sentiment prediction model with (right) or without (left) an ℓ_2 regularizer.

Now consider this explanation — ℓ_2 regularization introduces a new dynamic to gradient descent, whereby gradient updates have to constantly battle against a rescaling that is always trying to whittle *all* variables down to zero. The effort succeeds only for variables where gradient updates are pushing hardest to make them nonzero. Therefore, the weights for "necessary" variables survive, while "unnecessary" variables are thrown away. To say this more precisely, consider the regularized loss $\ell(\vec{\mathbf{w}}) + \lambda \|\vec{\mathbf{w}}\|_2^2$ whose gradient is

$$\nabla \ell + 2\lambda \vec{\mathbf{w}}$$

Thus the update rule in gradient descent can be written as

$$\vec{\mathbf{w}}^{t+1} \leftarrow \vec{\mathbf{w}}^t - \eta(\nabla \ell + 2\lambda \vec{\mathbf{w}}^t)$$

where $\vec{\mathbf{w}}^t$ denotes the weight vector at the t-th time step. This update rule can be rewritten as

$$\vec{\mathbf{w}}^{t+1} \leftarrow \vec{\mathbf{w}}^t(1 - 2\eta\lambda) - \eta \nabla \ell \tag{3.5}$$

The first term is *down-scaling*: if for example $\eta = \lambda = 0.1$, this amounts to multiplying the current vector by 0.98, and this of course will make $\vec{\mathbf{w}}$ very small in a few hundred iterations.

[6] Recall the famous *Occam's Razor* for judging goodness of scientific theories: The simpler the theory that explains the known facts, the more likely it is to be correct. An ML model can be seen as a "theory" about relationships in the data, and thus the simplest theory is to be preferred.

[7] See the blog https://www.offconvex.org for posts about generalization and deep learning. They also discuss how other ideas such as VC dimension, which we did not cover in this course, also do not apply in deep learning.

[8] See the online lecture video by Andrew Ng. https://www.youtube.com/watch?v=QjOILAQ0EFg

The second term is the gradient update to $\vec{\mathbf{w}}$. It can counteract the down-scaling by making the variables larger. But notice that the amount of change is based on how much each of the coordinates contribute to reducing the loss. Variables that are not useful will tend not to get increased by the gradient update and thus will keep getting down-scaled to low values. [9] The choice of λ mediates between these two processes.

[9] It is one of those "use it or lose it" situations!

3.4 *Gradient Descent in Python Programming*

In this section, we briefly discuss how to implement the Gradient Descent algorithm in Python. It is customary to use the *numpy* package to speed up computation and the *matplotlib* package for visualization.

```
# import necessary packages
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import cm # colormap

# initialize variables
num_iter = ...
x = np.zeros(num_iter + 1)
y = np.zeros(num_iter + 1)
x[0], y[0], eta = ...

# define functions to calculate f and grad_f
def f(x, y):
    ...
    return f

def grad_f(x, y):
    ...
    return grad_f

# run Gradient Descent
for i in range(num_iter):
    grad_x, grad_y = grad_f(x[i], y[i])
    x[i + 1] = x[i] - eta * grad_x
    y[i + 1] = y[i] - eta * grad_y

# plot the surface
xmin, xmax, ymin, ymax, n = ...
X, Y = np.meshgrid(np.linspace(xmin, xmax, n),
                   np.linspace(ymin, ymax, n))
Z = f(X, Y)

fig = plt.figure(figsize=(12, 10))
ax = fig.add_subplot(projection='3d')
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')
ax.view_init(elev=ax.elev, azim=ax.azim)
ax.plot_surface(X, Y, Z, cmap=cm.coolwarm, alpha=0.5)

# plot the trajectory of Gradient Descent
ax.plot(x, y, f(x, y), color='orange', markerfacecolor='black',
                       markeredgecolor='k', marker='o', markersize=5)
```

We first start off by importing necessary packages and initializing variables. The following code initializes *numpy* arrays of length *num_iter* + 1, with all entries initialized to 0

```
x = np.zeros(num_iter + 1)
y = np.zeros(num_iter + 1)
```

Sometimes, it is useful to make use of *np.ones()*, which will generate arrays filled with entries equal to 1.

We then define functions that will calculate the values of f and ∇f given an array of data points (x, y).

```
def f(x, y):
    ...
    return f

def grad_f(x, y):
    ...
    return grad_f
```

This allows us to run the Gradient Descent algorithm as in

```
for i in range(num_iter):
    grad_x, grad_y = grad_f(x[i], y[i])
    x[i + 1] = x[i] - eta * grad_x
    y[i + 1] = y[i] - eta * grad_y
```

Here we iteratively update the value of (x, y) using $\nabla f(x, y)$ and store each of the points in the array x and y.

We next plot the surface of the function $f(x, y)$. To start, we first create a grid of (x, y) points to evaluate $f(x, y)$ at.

```
X, Y = np.meshgrid(np.linspace(xmin, xmax, n),
                   np.linspace(ymin, ymax, n))
Z = f(X, Y)
```

The function call *np.linspace(min, max, n)* generates an array of n equally spaced values from *min* to *max*. For example, the code

```
np.linspace(-2, 2, 5)
```

will create an array $[-2, -1, 0, 1, 2]$. Then *np.meshgrid(x, y)* will create a grid from the array of x values and the array of y values. We can now perform the $3D$ plotting with the following code.

```
fig = plt.figure(figsize=(12, 10))
ax = fig.add_subplot(projection='3d')
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')
ax.view_init(elev=ax.elev, azim=ax.azim)
ax.plot_surface(X, Y, Z, cmap=cm.coolwarm, alpha=0.5)
```

Feel free to change the values of the optional parameters to understand their purpose. Unlike the code for plotting a scatter plot of a

linear regression in Chapter 1, here we create an object of the *Axes* class with the function *plt.figure().gca()*[10]. Then we call its instance methods to add features to it (*e.g.*, *x*-, *y*-, *z*-labels).

Finally, we can plot the trajectory of the Gradient Descent algorithm with the code

```
ax.plot(x, y, f(x, y), color='orange', markerfacecolor='black',
                       markeredgecolor='k', marker='o', markersize=5)
```

You can alternatively call

```
ax.scatter(x, y, f(x, y))
```

but the names of optional parameters might be slightly different.

[10] You can read more about the differences between these two *matplotlib* interfaces at https://matplotlib.org/matplotblog/posts/pyplot-vs-object-oriented-interface/

3.4.1 *Using Machine Learning Packages*

When the function f is simple and it is possible to calculate ∇f by hand, we can implement the Gradient Descent algorithm by hand as in the previous subsection. However, in most ML programs, the loss function f is very high-dimensional, and it is difficult to write a single function to directly compute the gradient ∇f. Instead, we can make use of functions defined in popular ML packages. Here, we introduce one such package called PyTorch:

- *torch*: This is a popular package used for designing and training machine learning models. PyTorch uses an object-oriented interface for user convenience and provides access to optimized array data structures called *tensors* to make computations faster and more efficient. The package also provides support for GPU training. [11]

[11] Documentation is available at https://pytorch.org/docs/stable/index.html

Using PyTorch, Gradient Descent can be implemented in just a few lines:

```
import torch
model = ...
opt = torch.optim.SGD(model.parameters(), lr=0.1)
```

The code above will create an instance of the *Optimzer* class, which has pre-defined methods that will compute the gradients and automate the Gradient Descent process.

3.4.2 *Beyond Vanilla Gradient Descent*

If you visit the documentation for *torch.optim*, [12] you may notice that there are other algorithms listed as an alternative to the Stochastic Gradient Descent. A lot of these algorithms are extensions of the GD algorithm we explained throughout this chapter, which have proven to be more effective than the vanilla GD algorithm in certain cases

[12] https://pytorch.org/docs/stable/optim.html

(*e.g.*, Adam, Adagrad, Nesterov momentum). For example, these algorithms may choose to add a *momentum* to the gradient, so that the rate of change of f will be accelerated if it has been updating in the same direction in the recent few steps. These algorithms may also choose to use a different learning rate for each of the model parameters. In particular, an appropriate learning rate can be computed based on the mean and the variance of the gradient values from the recent few steps.