3
Gradient Descent and Its Variants

Gradient descent is one of the most popular and widely used optimization algorithms, and is a first-order optimization algorithm. First-order optimization means that we calculate only the first-order derivative. As we saw in `Chapter 1`, *Introduction to Deep Learning*, we used gradient descent and calculated the first-order derivative of the loss function with respect to the weights of the network to minimize the loss.

Gradient descent is not only applicable to neural networks—it is also used in situations where we need to find the minimum of a function. In this chapter, we will go deeper into gradient descent, starting with the basics, and learn several variants of gradient descent algorithms. There are various flavors of gradient descent that are used for training neural networks. First, we will understand **Stochastic Gradient Descent** (**SGD**) and mini-batch gradient descent. Then, we'll explore how momentum is used to speed up gradient descent to attain convergence. Later in this chapter, we will learn about how to perform gradient descent in an adaptive manner by using various algorithms, such as Adagrad, Adadelta, RMSProp, Adam, Adamax, AMSGrad, and Nadam. We will take a simple linear regression equation and see how we can find the minimum of a linear regression's cost function using various types of gradient descent algorithms.

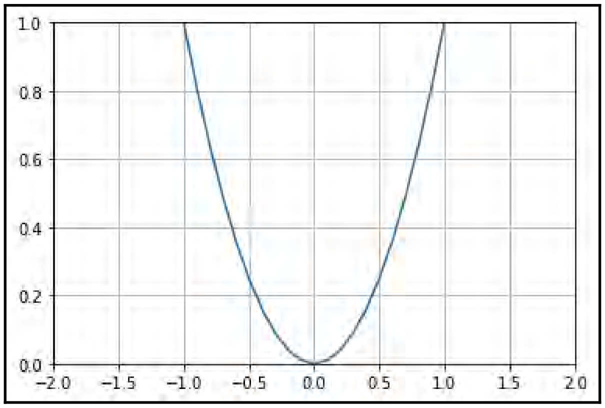
In this chapter, we will learn about the following topics:

- Demystifying gradient descent
- Gradient descent versus stochastic gradient descent
- Momentum and Nesterov accelerated gradient
- Adaptive methods of gradient descent

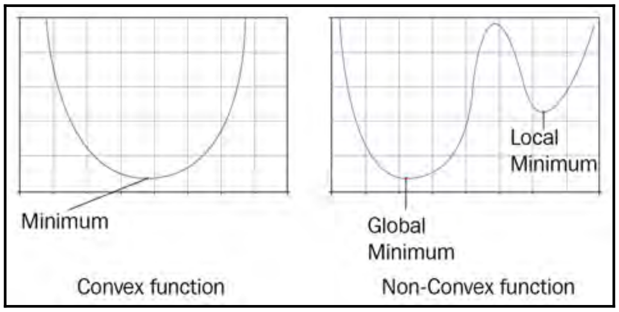
Demystifying gradient descent

Before we get into the details, let's understand the basics. What is a function in mathematics? A function represents the relation between input and output. We generally use f to denote a function. For instance, $f(x) = x^2$ implies a function that takes x as an input and returns x^2 as an output. It can also be represented as $y = x^2$.

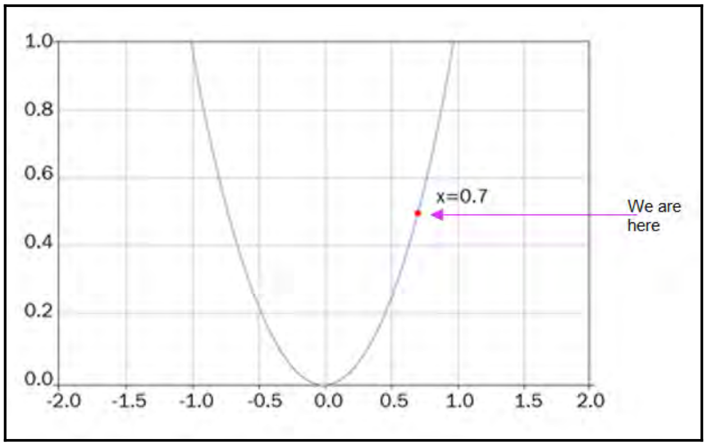
Here, we have a function, $y = x^2$, and we can plot and see what our function looks like:



The smallest value of a function is called the **minimum of a function**. As you can see in the preceding plot, the minimum of the x^2 function lies at 0. The previous function is called a **convex function**, and is where we have only one minimum value. A function is called a **non-convex function** when there is more than one minimum value. As we can see in the following diagram, a non-convex function can have many local minima and one global minimum value, whereas a convex function has only one global minimum value:



By looking at the graph of the x^2 function, we can easily say that it has its minimum value at $x = 0$. But how can we find the minimum value of a function mathematically? First, let's assume *x = 0.7*. Thus, we are at a position where *x = 0.7,* as shown in the following graph:



Now, we need to go to zero, which is our minimum value, but how can we reach it? We can reach it by calculating the derivative of the function, $y = x^2$. So, the derivative of the function, y, with respect to x, is as follows:

$$y = x^2$$

$$\frac{dy}{dx} = 2x$$

Since we are at *x = 0.7* and substituting this in the previous equation, we get the following equation:

$$\frac{dy}{dx} = 2(0.7) = 1.4$$

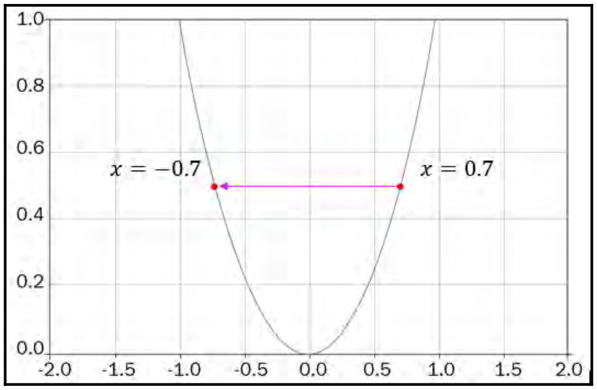
After calculating the derivative, we update our position of x according to the following update rule:

$$x = x - \frac{dy}{dx}$$

$$x = 0.7 - 1.4$$

$$x = -0.7$$

As we can see in the following graph, we were at *x = 0.7* initially, but, after computing the gradient, we are now at the updated position of *x = -0.7*. However, this is something we don't want because we missed our minimum value, which is *x = 0*, and reached somewhere else:



To avoid this, we introduce a new parameter called learning rate, α, in the update rule. It helps us to slow down our gradient steps so that we won't miss out the minimal point. We multiply the gradients by the learning rate and update the x value, as follows:

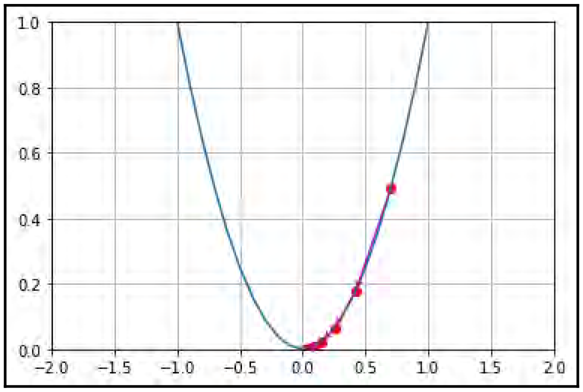
$$x = x - \alpha \frac{dy}{dx}$$

Let's say that $\alpha = 0.15$; now, we can write the following:

$$x = 0.7 - (0.15 * 1.4)$$

$$x = 0.49$$

As we can see in the following graph, after multiplying the gradients by the learning rate with the updated *x* value, we descended from the initial position, *x = 0.7*, to *x = 0.49*:

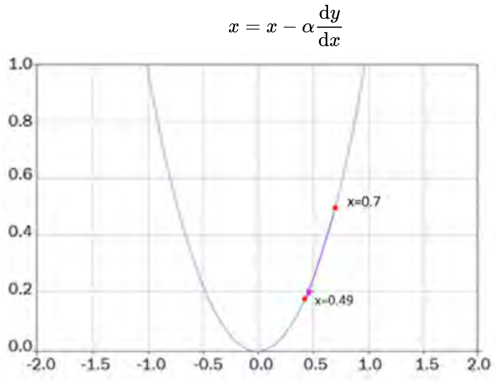


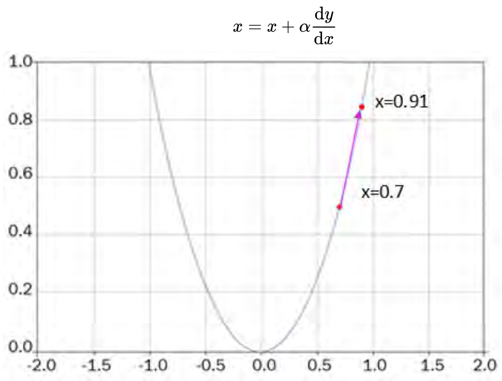
However, this still isn't our optimal minimum value. We need to go further down until we reach the minimum value; that is, $x = 0$. So, for some n number of iterations, we have to repeat the same process until we reach the minimal point. That is, for some n number of iterations, we update the value of x using the following update rule until we reach the minimal point:

$$x = x - \alpha.\frac{dy}{dx}$$

Okay – why is there a minus in the preceding equation? That is, why we are subtracting $\alpha.\frac{dy}{dx}$ from x? Why can't we add them and have our equation as $x = x + \alpha.\frac{dy}{dx}$?

This is because we are finding the minimum of a function, so we need to go downward. If we add x to $\alpha.\frac{dy}{dx}$, then we go upward on every iteration, and we cannot find the minimum value, as shown in the following graphs:



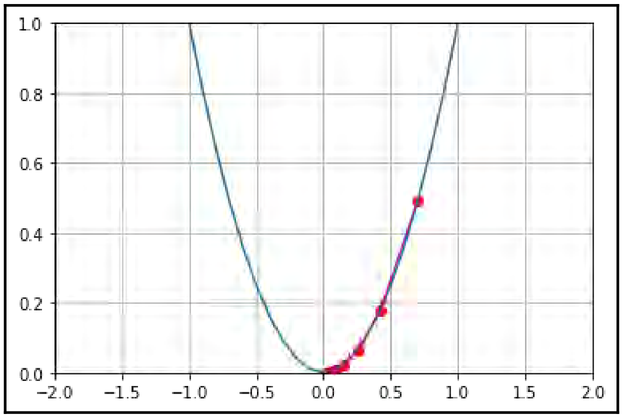


Thus, on every iteration, we compute gradients of y with respect to x, that is, $\frac{dy}{dx}$, multiply the gradients by the learning rate, that is, $\alpha.\frac{dy}{dx}$, and subtract it from the x value to get the updated x value, as follows:

$$x = x - \alpha.\frac{dy}{dx}$$

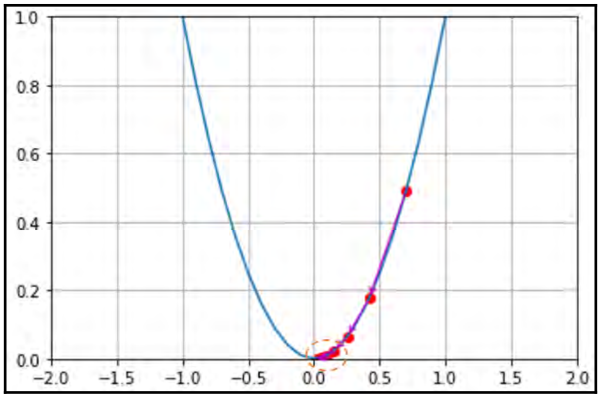
By repeating this step on every iteration, we go downward from the cost function and reach the minimum point. As we can see in the following graph, we moved downward from the initial position of 0.7 to 0.49, and then, from there, we reached 0.2.

Then, after several iterations, we reach the minimum point, which is 0.0:



We say we attained **convergence** when we reach the minimum of the function. But the question is: how do we know that we attained convergence? In our example, $y = x^2$, we know that the minimum value is 0. So, when we reach 0, we can say that we found the minimum value that we attained convergence. But how can we mathematically say that 0 is the minimum value of the function, $y = x^2$?

Let's take a closer look at the following graph, which shows how the value of x changes on every iteration. As you may notice, the value of x is 0.009 in the fifth iteration, 0.008 in the sixth iteration, and 0.007 in the seventh iteration. As you can see, there's not much difference between the fifth, sixth, and seventh iterations. When there is little change in the value of x over iterations, then we can conclude that we have attained convergence:



Okay, but what is the use of all this? Why are we trying to find the minimum of a function? When we're training a model, our goal is to minimize the loss function of the model. Thus, with gradient descent, we can find the minimum of the cost function. Finding the minimum of the cost function gives us an optimal parameter of the model with which we can obtain the minimal loss. In general, we denote the parameters of the model by θ. The following equation is called the parameter update rule or weight update rule:

$$\theta = \theta - \alpha \cdot \nabla_\theta J(\theta) \tag{1}$$

Here, we have the following:

- $\theta$is the parameter of the model
- $\alpha$is the learning rate
- $\nabla_\theta J(\theta)$is the gradient

We update the parameter of the model for several iterations according to the parameter update rule until we attain convergence.

Performing gradient descent in regression

So far, we have understood how the gradient descent algorithm finds the optimal parameters of the model. In this section, we will understand how we can use gradient descent in linear regression and find the optimal parameter.

The equation of a simple linear regression can be expressed as follows:

$$\hat{y} = mx + b$$

Thus, we have two parameters, m and b. Now, we will see how can we use gradient descent and find the optimal values for these two parameters.

Importing the libraries

First, we need to import the required libraries:

```
import warnings
warnings.filterwarnings('ignore')

import random
import math
import numpy as np
from matplotlib import pyplot as plt
%matplotlib inline
```

Preparing the dataset

Next, we will generate some random data points with `500` rows and `2` columns (*x* and *y*) and use them for training:

```
data = np.random.randn(500, 2)
```

As you can see, our data has two columns:

```
print data[0]

array([-0.08575873,  0.45157591])
```

The first column indicates the x value:

```
print data[0,0]

-0.08575873243708057
```

The second column indicates the y value:

```
print data[0,1]

0.4515759149158441
```

We know that the equation of a simple linear regression is expressed as follows:

$$\hat{y} = mx + b \tag{2}$$

Thus, we have two parameters, m and b. We store both of these parameters in an array called `theta`. First, we initialize `theta` with zeros, as follows:

```
theta = np.zeros(2)
```

The `theta[0]` function represents the value of m, while the `theta[1]` function represents the value of b:

```
print theta

array([0., 0.])
```

Defining the loss function

The **mean squared error** (**MSE**) of regression is given as follows:

$$J = \frac{1}{N}\sum_{i=1}^{N}(y - \hat{y})^2 \tag{3}$$

Here, N is the number of training samples, y is the actual value, and $\hat{y}$ is the predicted value.

The implementation of the preceding loss function is shown here. We feed the `data` and the model parameter, `theta`, to the loss function, which returns the MSE. Remember that `data[,0]` has an x value and that `data[,1]` has a y value. Similarly, `theta [0]` has a value of `m` and `theta[1]` has a value of b.

Let's define the loss function:

```
def loss_function(data,theta):
```

Now, we need to get the value of m and b:

```
    m = theta[0]
    b = theta[1]

    loss = 0
```

We do this for each iteration:

```
        for i in range(0, len(data)):
```

Now, we get the value of x and y:

```
            x = data[i, 0]
            y = data[i, 1]
```

Then, we predict the value of $\hat{y}$:

```
            y_hat = (m*x + b)
```

Here, we compute the loss as given in equation *(3)*:

```
            loss = loss + ((y - (y_hat)) ** 2)
```

Then, we compute the mean squared error:

```
        mse = loss / float(len(data))

        return mse
```

When we feed our randomly initialized `data` and model parameter, `theta`, `loss_function` returns the mean squared loss, as follows:

```
    loss_function(data, theta)

    1.0253548008165727
```

Now, we need to minimize this loss. In order to minimize the loss, we need to calculate the gradient of the loss function, J, with respect to the model parameters, m and b, and update the parameter according to the parameter update rule. First, we will calculate the gradients of the loss function.

Computing the gradients of the loss function

The gradients of the loss function, J, with respect to the parameter m, are given as follows:

$$\frac{dJ}{dm} = \frac{2}{N} \sum_{i=1}^{N} -x_i \left(y_i - (mx_i + b)\right) \tag{4}$$

The gradients of the loss function, J, with respect to the parameter b, are given as follows:

$$\frac{dJ}{db} = \frac{2}{N}\sum_{i=1}^{N} -\left(y_i - (mx_i + b)\right) \quad (5)$$

We define a function called `compute_gradients`, which takes the parameters, `data` and `theta` as input and returns the computed gradients:

```
def compute_gradients(data, theta):
```

Now, we need to initialize the gradients:

```
    gradients = np.zeros(2)
```

Then, we need to save the total number of data points in `N`:

```
    N = float(len(data))
```

Now, we can get the value of m and b:

```
    m = theta[0]
    b = theta[1]
```

We do the same for each iteration:

```
    for i in range(0, len(data)):
```

Then, we get the value of x and y:

```
        x = data[i, 0]
        y = data[i, 1]
```

Now, we compute the gradient of the loss with respect to m, as given in equation *(4)*:

```
        gradients[0] += - (2 / N) * x * (y - (( m* x) + b))
```

Then, we compute the gradient of the loss with respect to b, as given in equation *(5)*:

```
        gradients[1] += - (2 / N) * (y - ((theta[0] * x) + b))
```

We need to add `epsilon` to avoid division by zero error:

```
    epsilon = 1e-6
    gradients = np.divide(gradients, N + epsilon)

    return gradients
```

When we feed our randomly initialized `data` and `theta` model parameter, the `compute_gradients` function returns the gradients with respect to m, that is, $\frac{dJ}{dm}$, and gradients with respect to b, that is, $\frac{dJ}{db}$, as follows:

```
compute_gradients(data,theta)

array([-9.08423989e-05,  1.05174511e-04])
```

Updating the model parameters

Now that we've computed the gradients, we need to update our model parameters according to our update rule, as follows:

$$m = m - \alpha \frac{dJ}{dm} \tag{6}$$

$$b = b - \alpha \frac{dJ}{db} \tag{7}$$

Since we stored m in `theta[0]` and b in `theta[1]`, we can write our update equation as follows:

$$\theta = \theta - \alpha \frac{dJ}{d\theta} \tag{8}$$

As we learned in the previous section, updating gradients on just one iteration will not lead us to convergence, that is, the minimum of the cost function, so we need to compute gradients and update the model parameter for several iterations.

First, we need to set the number of iterations:

```
num_iterations = 50000
```

Now, we need to define the learning rate:

```
lr = 1e-2
```

Next, we will define a list called `loss` for storing the loss on every iteration:

```
loss = []
```

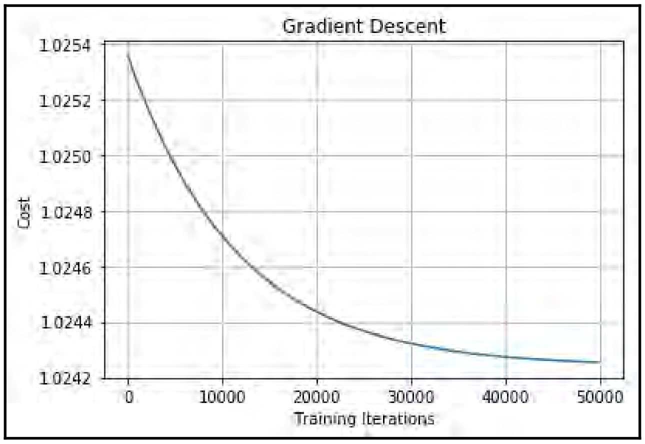
On each iteration, we will calculate and update the gradients according to our parameter update rule from equation *(8)*:

```
theta = np.zeros(2)

for t in range(num_iterations):
    #compute gradients
    gradients = compute_gradients(data, theta)
    #update parameter
    theta = theta - (lr*gradients)
    #store the loss
    loss.append(loss_function(data,theta))
```

Now, we need to plot the `loss` (`Cost`) function:

```
plt.plot(loss)
plt.grid()
plt.xlabel('Training Iterations')
plt.ylabel('Cost')
plt.title('Gradient Descent')
```

The following plot shows how the loss (**Cost**) decreases over the training iterations:



Thus, we learned that gradient descent can be used to find the optimal parameters of the model, which we can then use to minimize the loss. In the next section, we will learn about several variants of the gradient descent algorithm.

Gradient descent versus stochastic gradient descent

We update the parameter of the model multiple times with our parameter update equation *(1)* until we find the optimal parameter value. In gradient descent, to perform a single parameter update, we iterate through all the data points in our training set. So, every time we update the parameters of the model, we iterate through all the data points in the training set. Updating the parameters of the model only after iterating through all the data points in the training set makes gradient descent very slow and it will increase the training time, especially when we have a large dataset.

Let's say we have a training set with 1 million data points. We know that we update the parameters of the model multiple times to find the optimal parameter value. So, even to perform a single parameter update, we go through all 1 million data points in our training set and then update the model parameters. This will definitely make the training slow. This is because we can't just find the optimal parameter with a single update; we need to update the parameters of the model several times to find the optimal value. So, if we iterate through all 1 million data points in our training set for every parameter update, it will definitely slow down our training.

Thus, to combat this, we introduce **stochastic gradient descent** (**SGD**). Unlike gradient descent, we don't have to wait to update the parameter of the model after iterating all the data points in our training set; we just update the parameters of the model after iterating through every single data point in our training set.

Since we update the parameters of the model in SGD after iterating every single data point, it will learn the optimal parameter of the model faster compared to gradient descent, and this will minimize the training time.

How is SGD useful? When we have a huge dataset, by using the vanilla gradient descent method, we update the parameters only after iterating through all the data points in that huge dataset. So, after many iterations over the whole dataset, we reach convergence and, apparently, it takes a long time. But, in SGD, we update the parameters after iterating through every single training sample. That is, we are learning to find the optimal parameters right from the first training sample, which helps to attain convergence faster compared to the vanilla gradient descent method.

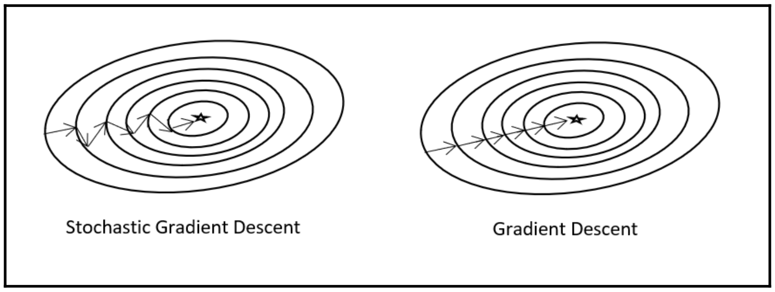
We know that the epoch specifies the number of times the neural network sees the whole training data. Thus, in gradient descent, on each epoch, we perform the parameter update. This means that, after every epoch, the neural networks see the whole training data. We perform the parameter update for each epoch as follows:

$$\theta = \theta - \alpha \cdot \nabla_\theta J(\theta)$$

However, in stochastic gradient descent, we don't have to wait until the completion of each epoch to update the parameters. That is, we don't have to wait until the neural network sees the whole training data to update the parameters. Instead, we update the parameters of the network right from seeing a single training sample for each epoch:

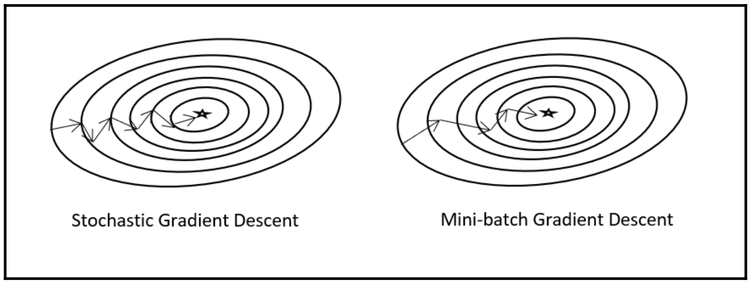
$$\theta = \theta - \alpha \cdot \nabla_\theta J(\theta)$$

The following contour plot shows how gradient descent and stochastic gradient descent perform the parameter updates and find the minimum cost. The star symbol in the center of the plot denotes the position where we have a minimum cost. As you can see, SGD reaches convergence faster than vanilla gradient descent. You can also observe the oscillations in the gradient steps on SGD; this is because we are updating the parameter for every training sample, so, the gradient step in SGD changes frequently compared to the vanilla gradient descent:



There is also another variant of gradient descent called **mini-batch gradient descent**. It takes the pros of both vanilla gradient descent and stochastic gradient descent. In SGD, we saw that we update the parameter of the model for every training sample. However, in mini-batch gradient descent, instead of updating the parameters after iterating each training sample, we update the parameters after iterating some batches of data points. Let's say the batch size is 50, which means that we update the parameter of the model after iterating through 50 data points instead of updating the parameter after iterating through each individual data point.

The following diagram shows the contour plot of SGD and mini-batch gradient descent:



Here are the differences between these types of gradient descent in a nutshell:

- **Gradient descent**: Updates the parameters of the model after iterating through all the data points in the training set
- **Stochastic gradient descent**: Updates the parameter of the model after iterating through every single data point in the training set
- **Mini-batch gradient descent**: Updates the parameters of the model after iterating *n* number of data points in the training set

TIP

Mini-batch gradient descent is preferred over vanilla gradient descent and SGD for large datasets since mini-batch gradient descent outperforms the other two.

The code for mini-batch gradient descent is as follows.

First, we need to define the `minibatch` function:

```
def minibatch(data, theta, lr = 1e-2, minibatch_ratio = 0.01,
num_iterations = 1000):
```

Next, we will define `minibatch_size` by multiplying the length of data by `minibatch_ratio`:

```
    minibatch_size = int(math.ceil(len(data) * minibatch_ratio))
```

Now, on each iteration, we perform the following:

```
    for t in range(num_iterations):
```

Next, select `sample_size`:

```
            sample_size = random.sample(range(len(data)), minibatch_size)
            np.random.shuffle(data)
```

Now, sample the data based on `sample_size`:

```
            sample_data = data[0:sample_size[0], :]
```

Compute the gradients for `sample_data` with respect to `theta`:

```
            grad = compute_gradients(sample_data, theta)
```

After computing the gradients for the sampled data with the given mini-batch size, we update the model parameter, `theta`, as follows:

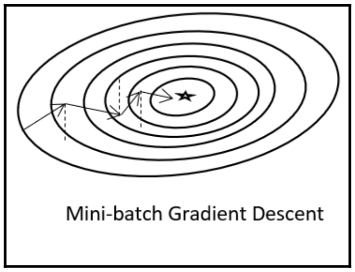
```
            theta = theta - (lr * grad)

    return theta
```

Momentum-based gradient descent

In this section, we will learn about two new variants of gradient descent, called **momentum** and Nesterov accelerated gradient.

Gradient descent with momentum

We have a problem with SGD and mini-batch gradient descent due to the oscillations in the parameter update. Take a look at the following plot, which shows how mini-batch gradient descent is attaining convergence. As you can see, there are oscillations in the gradient steps. The oscillations are shown by the dotted line. As you may notice, it is making a gradient step toward one direction, and then taking a different direction, and so on, until it reaches convergence:



This oscillation occurs because, since we update the parameters after iterating every n number of data points, the direction of the update will have some variance, and this leads to oscillations in every gradient step. Due to this oscillation, it is hard to reach convergence, and it slows down the process of attaining it.

To alleviate this, we'll introduce a new technique called **momentum**. If we can understand what the right direction is for the gradient steps to attain convergence faster, then we can make our gradient steps navigate in that direction and reduce the oscillation in the irrelevant directions; that is, we can reduce taking directions that do not lead us to convergence.

So, how can we do this? We basically take a fraction of the parameter update from the previous gradient step and add it to the current gradient step. In physics, momentum keeps an object moving after a force is applied. Here, the momentum keeps our gradient moving toward the direction that leads to convergence.

If you take a look at the following equation, you can see we are basically taking the parameter update from the previous step, v_{t-1}, and adding it to the current gradient step, $\nabla_\theta J(\theta)$. How much information we want to take from the previous gradient step depends on the factor, that is, γ, and the learning rate, which is denoted by η:

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$$

In the preceding equation, v_t is called velocity, and it accelerates gradients in the direction that leads to convergence. It also reduces oscillations in an irrelevant direction by adding a fraction of a parameter update from the previous step to the current step.

Thus, the parameter update equation with momentum is expressed as follows:

$$\boxed{\theta = \theta - v_t}$$

By doing this, performing mini-batch gradient descent with momentum helps us to reduce oscillations in gradient steps and attain convergence faster.

Now, let's look at the implementation of momentum.

First, we define the `momentum` function, as follows:

```
def momentum(data, theta, lr = 1e-2, gamma = 0.9, num_iterations = 1000):
```

Then, we initialize `vt` with zeros:

```
        vt = np.zeros(theta.shape[0])
```

The following code is executed to cover the range for each iteration:

```
        for t in range(num_iterations):
```

Now, we compute `gradients` with respect to `theta`:

```
            gradients = compute_gradients(data, theta)
```

Next, we update `vt` to be $v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$:

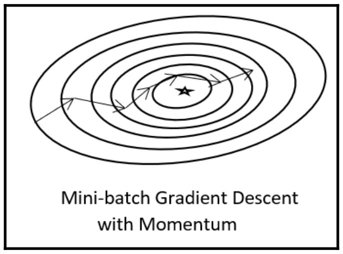
```
            vt = gamma * vt + lr * gradients
```

Now, we update the model parameter, `theta`, as $\theta = \theta - v_t$:

```
            theta = theta - vt

     return theta
```

Nesterov accelerated gradient

One problem with momentum is that it might miss out the minimum value. That is, as we move closer toward convergence (the minimum point), the value for momentum will be high. When the value of momentum is high while we are near to attaining convergence, then the momentum actually pushes the gradient step high and it might miss out on the actual minimum value; that is, it might overshoot the minimum value when the momentum is high when we are near to convergence, as shown in the following diagram:



To overcome this, Nesterov introduced a new method called **Nesterov accelerated gradient** (**NAG**).

The fundamental motivation behind Nesterov momentum is that, instead of calculating the gradient at the current position, we calculate gradients at the position where the momentum would take us to, and we call that position the lookahead position.

What does this mean, though? In the *Gradient descent with momentum* section, we learned about the following equation:

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta) \tag{9}$$

The preceding equation tells us that we are basically pushing the current gradient step, $\nabla_\theta J(\theta)$, to a new position using a fraction of the parameter update from the previous step, γv_{t-1}, which will help us to attain convergence. However, when the momentum is high, this new position will actually overshoot the minimum value.

Thus, before making a gradient step with momentum and reaching a new position, if we understand which position the momentum will take us to, then we can avoid overshooting the minimum value. If we find out that momentum will take us to the position that actually misses the minimum value, then we can slow down the momentum and try to reach the minimum value.

But how can we find the position that the momentum will take us to? In equation *(2)*, instead of calculating gradients with respect to the current gradient step, $\nabla_\theta J(\theta)$, we calculate gradients with respect to $\nabla_\theta J(\theta - \gamma v_{t-1})$. The term, $\theta - \gamma v_{t-1}$, basically tells us the approximate position of where our next gradient step is going to be. We call this the lookahead position. This gives us an idea of where our next gradient step is going to be.

So, we can rewrite our v_t equation according to NAG as follows:

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1})$$

We update our parameter as follows:

$$\boxed{\theta = \theta - v_t}$$

Updating the parameters with the preceding equation prevents us from missing the minimum value by slowing down the momentum when the gradient steps are near to convergence. The Nesterov accelerated method is implemented as follows.

First, we define the `NAG` function:

```
def NAG(data, theta, lr = 1e-2, gamma = 0.9, num_iterations = 1000):
```

Then, we initialize the value of `vt` with zeros:

```
        vt = np.zeros(theta.shape[0])
```

For every iteration, we perform the following steps:

```
        for t in range(num_iterations):
```

Now, we need to compute the gradients with respect to $\theta - \gamma v_{t-1}$:

```
            gradients = compute_gradients(data, theta - gamma * vt)
```

Then, we update `vt` as $v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1})$:

```
            vt = gamma * vt + lr * gradients
```

Now, we update the model parameter, `theta`, to $\theta = \theta - v_t$:

```
            theta = theta - vt

        return theta
```

Adaptive methods of gradient descent

In this section, we will learn about several adaptive versions of gradient descent.

Setting a learning rate adaptively using Adagrad

When we build a deep neural network, we have many parameters. Parameters are basically the weights of the network, so when we build a network with many layers, we will have many weights, say, $\theta^1, \theta^2, \theta^3 .. \theta^i .. \theta^n$. Our goal is to find the optimal values for all these weights. In all of the previous methods we learned about, the learning rate was a common value for all the parameters of the network. However **Adagrad** (short for **adaptive gradient**) adaptively sets the learning rate according to a parameter.

Parameters that have frequent updates or high gradients will have a slower learning rate, while a parameter that has an infrequent update or small gradients will also have a slower learning rate. But why do we have to do this? It is because parameters that have infrequent updates implies that they are not trained enough, so we set a high learning rate for them, and parameters that have frequent updates implies that they are trained enough, so we set their learning rate to a low value so that we don't overshoot the minimum.

Now, let's see how Adagrad adaptively changes the learning rate. Previously, we represented the gradient with $\nabla_\theta J(\theta)$. For simplicity, from now on in this chapter, we'll represent gradients with g. So, the gradient of a parameter, θ^i, at an iteration, t, can be represented as follows:

$$g_t^i = \nabla_\theta J(\theta_t^i)$$

Therefore, we can rewrite our update equation with g as the gradient notation as follows:

$$\theta_t^i = \theta_{t-1}^i - \eta \cdot g_t^i$$

Now, for every iteration, t, to update a parameter, θ^i, we divide the learning rate by the sum of squares of all previous gradients of the parameter, θ^i, as follows:

$$\theta_t^i = \theta_{t-1}^i - \frac{\eta}{\sqrt{\sum_{\tau=1}^{t}(g_\tau^i)^2 + \epsilon}} \cdot g_t^i$$

Here, $\sqrt{\sum_{\tau=1}^{t}(g_\tau^i)^2 + \epsilon}$ implies the sum of squares of all previous gradients of the parameter θ^i. We added ϵ just to avoid the division by zero error. We typically set the value of ϵ to a small number. The question that arises here is, why are we dividing the learning rate by a sum of squares of all the previous gradients?

We learned that parameters that have frequent updates or high gradients will have a slower learning rate, while parameters that have an infrequent update or small gradients will also have a high learning rate.

The sum, $\sqrt{\sum_{\tau=1}^{t}(g_\tau^i)^2 + \epsilon}$, actually scales our learning rate. That is, when the sum of the squared past gradients has a high value, we are basically dividing the learning rate by a high value, so our learning rate will become less. Similarly, if the sum of the squared past gradients has a low value, we are dividing the learning rate by a lower value, so our learning rate value will become high. This implies that the learning rate is inversely proportional to the sum of the squares of all the previous gradients of the parameter.

Here, our update equation is expressed as follows:

$$\boxed{\theta_t^i = \theta_{t-1}^i - \frac{\eta}{\sqrt{\sum_{\tau=1}^t (g_\tau^i)^2 + \epsilon}} \cdot g_t^i}$$

In a nutshell, in Adagrad, we set the learning rate to a low value when the previous gradient value is high, and to a high value when the past gradient value is lower. This means that our learning rate value changes according to the past gradient updates of the parameters.

Now that we have learned how the Adagrad algorithm works, let's strengthen our knowledge by implementing it. The code for the Adagrad algorithm is given as follows.

First, define the `AdaGrad` function:

```
def AdaGrad(data, theta, lr = 1e-2, epsilon = 1e-8, num_iterations =
10000):
```

Define the variable called `gradients_sum` to hold the sum of gradients and initialize them with zeros:

```
    gradients_sum = np.zeros(theta.shape[0])
```

For every iteration, we perform the following steps:

```
    for t in range(num_iterations):
```

Then, we compute the `gradients` of loss with respect to `theta`:

```
        gradients = compute_gradients(data, theta)
```

Now, we calculate the sum of the gradients squared, that is, $\sum_{\tau=1}^{t} (g_\tau^i)^2$:

```
        gradients_sum += gradients ** 2
```

Afterward, we compute the gradient updates, that is, $\frac{g_t^i}{\sqrt{\sum_{\tau=1}^t (g_\tau^i)^2 + \epsilon}}$:

```
        gradient_update = gradients / (np.sqrt(gradients_sum + epsilon))
```

Now, update the `theta` model parameter so that it's $\theta_t^i = \theta_{t-1}^i - \frac{\eta}{\sqrt{\sum_{\tau=1}^{t}(g_\tau^i)^2 + \epsilon}} \cdot g_t^i$:

```
            theta = theta - (lr * gradient_update)

        return theta
```

Again, there is a shortcoming associated with the Adagrad method. For every iteration, we are accumulating and summing all the past squared gradients. So, on every iteration, our sum of the squared past gradients value will increase. When the sum of the squared past gradient value is high, we will have a large number in the denominator. When we divide the learning rate by a very large number, then the learning rate will become very small. So, over several iterations, the learning rate starts decaying and becomes an infinitesimally small number – that is, our learning rate will be monotonically decreasing. When the learning rate reaches a very low value, then it takes a long time to attain convergence.

In the next section, we will see how Adadelta tackles this shortcoming.

Doing away with the learning rate using Adadelta

Adadelta is an enhancement of the Adagrad algorithm. In Adagrad, we noticed the problem of the learning rate diminishing to a very low number. Although Adagrad learns the learning rate adaptively, we still need to set the initial learning rate manually. However, in Adadelta, we don't need the learning rate at all. So how does the Adadelta algorithm learn?

In Adadelta, instead of taking the sum of all the squared past gradients, we can set a window of size w and take the sum of squared past gradients only from that window. In Adagrad, we took the sum of all the squared past gradients and it led to the learning rate diminishing to a low number. To avoid that, we take the sum of the squared past gradients only from a window.

If w is the window size, then our parameter update equation becomes the following:

$$\theta_t^i = \theta_{t-1}^i - \frac{\eta}{\sqrt{\sum_{\tau=t-w+1}^{t}(g_\tau^i)^2 + \epsilon}} \cdot g_t^i$$

However, the problem is that, although we are taking gradients only from within a window, w, squaring and storing all the gradients from the window in each iteration is inefficient. So, instead of doing that, we can take the running average of gradients.

We compute the running average of gradients at an iteration, t, $E[g^2]_t$, by adding the previous running average of gradients, $E[g^2]_{t-1}$, and current gradients, g_t^2:

$$E[g^2]_t = E[g^2]_{t-1} + g_t^2$$

Instead of just taking the running average, we take the exponentially decaying running average of gradients, as follows:

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma) g_t^2 \tag{10}$$

Here, γ is called the exponential decaying rate and is similar to the one we saw in momentum – that is, it is used for deciding how much information from the previous running average of gradients should be added.

Now, our update equation becomes the following:

$$\theta_t^i = \theta_{t-1}^i - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} \cdot g_t^i$$

For notation simplicity, let's denote $-\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} \cdot g_t^i$ as $\nabla\theta_t$ so that we can rewrite the previous update equation as follows:

$$\theta_t^i = \theta_{t-1}^i + \nabla\theta_t \tag{11}$$

From the previous equation, we can infer the following:

$$\nabla\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} \cdot g_t^i \tag{12}$$

If you look at the denominator in the previous equation, we are basically computing the root mean squared of gradients up to an iteration, t, so we can simply write that in shorthand as follows:

$$RMS[g_t] = \sqrt{E[g^2]_t + \epsilon} \tag{13}$$

By substituting equation *(13)* in equation *(12)*, we can write the following:

$$\nabla\theta_t = -\frac{\eta}{RMS[g_t]} \cdot g_t^i \tag{14}$$

However, we still have the learning rate, η, term in our equation. How can we do away with that? We can do so by making the units of the parameter update in accordance with the parameter. As you may have noticed, the units of θ_t and $\nabla\theta_t$ don't really match. To combat this, we compute the exponentially decaying average of the parameter updates,$\nabla\theta_t$, as we computed an exponentially decaying average of gradients, g_t, in equation *(10)*. So, we can write the following:

$$E[\Delta\theta^2]_t = \gamma E[\Delta\theta^2]_{t-1} + (1-\gamma)\Delta\theta_t^2$$

It's like the RMS of gradients, $RMS[g_t]$, which is similar to equation *(13)*. We can write the RMS of the parameter update as follows:

$$RMS[\Delta\theta]_t = \sqrt{E[\Delta\theta^2]_t + \epsilon}$$

However, the RMS value for the parameter update, $\Delta\theta_t$ is not known, that is, $RMS[\Delta\theta]_t$ is not known, so we can just approximate it by considering until the previous update, $RMS[\Delta\theta]_{t-1}$.

Now, we just replace our learning rate with the RMS value of the parameter updates. That is, we replace η with $RMS[\Delta\theta]_{t-1}$ in equation *(14)* and write the following:

$$\nabla\theta_t = -\frac{RMS[\Delta\theta]_{t-1}}{RMS[g_t]} \cdot g_t^i \tag{15}$$

Substituting equation *(15)* in equation *(11)*, our final update equation becomes the following:

$$\theta_t^i = \theta_{t-1}^i - \frac{RMS[\Delta\theta]_{t-1}}{RMS[g_t]} \cdot g_t^i$$

$$\boxed{\theta_t^i = \theta_{t-1}^i + \nabla\theta_t}$$

Now, let's understand the Adadelta algorithm by implementing it.

First, we define the AdaDelta function:

```
def AdaDelta(data, theta, gamma = 0.9, epsilon = 1e-5, num_iterations =
1000):
```

Then, we initialize the E_grad2 variable with zero for storing the running average of gradients, and E_delta_theta2 with zero for storing the running average of the parameter update, as follows:

```
    # running average of gradients
    E_grad2 = np.zeros(theta.shape[0])

    #running average of parameter update
    E_delta_theta2 = np.zeros(theta.shape[0])
```

For every iteration, we perform the following steps:

```
    for t in range(num_iterations):
```

Now, we need to compute the gradients with respect to theta:

```
        gradients = compute_gradients(data, theta)
```

Then, we can compute the running average of gradients:

```
        E_grad2 = (gamma * E_grad2) + ((1. - gamma) * (gradients ** 2))
```

Here, we will compute delta_theta, that is, $\nabla\theta_t = -\frac{RMS[\Delta\theta]_{t-1}}{RMS[g_t]} \cdot g_t^i$:

```
        delta_theta = - (np.sqrt(E_delta_theta2 + epsilon)) /
(np.sqrt(E_grad2 + epsilon)) * gradients
```

Now, we can compute the running average of the parameter update, $E[\Delta\theta^2]_t = \gamma E[\Delta\theta^2]_{t-1} + (1-\gamma)\Delta\theta_t^2$:

```
        E_delta_theta2 = (gamma * E_delta_theta2) + ((1. - gamma) *
(delta_theta ** 2))
```

Next, we will update the parameter of the model, theta, so that it's $\theta_t^i = \theta_{t-1}^i + \nabla\theta_t$:

```
        theta = theta + delta_theta

    return theta
```

Overcoming the limitations of Adagrad using RMSProp

Similar to Adadelta, RMSProp was introduced to combat the decaying learning rate problem of Adagrad. So, in RMSProp, we compute the exponentially decaying running average of gradients as follows:

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma)g_t^2$$

Instead of taking the sum of the square of all the past gradients, we use this running average of gradients. This means that our update equation becomes the following:

$$\boxed{\theta_t^i = \theta_{t-1}^i - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} \cdot g_t^i}$$

It is recommended to assign a value of learning η to `0.9`. Now, we will learn how to implement RMSProp in Python.

First, we need to define the `RMSProp` function:

```
def RMSProp(data, theta, lr = 1e-2, gamma = 0.9, epsilon = 1e-6,
num_iterations = 1000):
```

Now, we need to initialize the `E_grad2` variable with zeros to store the running average of gradients:

```
    E_grad2 = np.zeros(theta.shape[0])
```

For every iteration, we perform the following steps:

```
    for t in range(num_iterations):
```

Then, we compute the `gradients` with respect to `theta`:

```
        gradients = compute_gradients(data, theta)
```

Next, we compute the running average of the gradients, that is, $E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma)g_t^2$:

```
        E_grad2 = (gamma * E_grad2) + ((1. - gamma) * (gradients ** 2))
```

Now, we update the parameter of the model, `theta`, so that it's $\theta_t^i = \theta_{t-1}^i - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} \cdot g_t^i$:

```
        theta = theta - (lr / (np.sqrt(E_grad2 + epsilon)) * gradients)
    return theta
```

Adaptive moment estimation

Adaptive moment estimation, known as **Adam** for short, is one of the most popularly used algorithms for optimizing a neural network. While reading about RMSProp, we learned that we compute the running average of squared gradients to avoid the diminishing learning rate problem:

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma) g_t^2$$

The final updated equation of RMSprop is given as follows:

$$\theta_t^i = \theta_{t-1}^i - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} \cdot g_t^i$$

Similar to this, in Adam, we also compute the running average of the squared gradients. However, along with computing the running average of the squared gradients, we also compute the running average of the gradients.

The running average of gradients is given as follows:

$$E[g]_t = \beta_1 E[g]_{t-1} + (1-\beta_1) g_t \qquad (16)$$

The running average of squared gradients is given as follows:

$$E[g^2]_t = \beta_2 E[g^2]_{t-1} + (1-\beta_2) g_t^2 \qquad (17)$$

Since a lot of literature and libraries represent the decaying rate in Adam as β instead of γ, we'll also use β to represent the decaying rate in Adam. Thus, β_1 and β_2 in equations *(16)* and *(17)* denote the exponential decay rates for the running average of the gradients and the squared gradients, respectively.

So, our updated equation becomes the following:

$$\theta_t^i = \theta_{t-1}^i - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} \cdot E[g]_t$$

The running average of the gradients and running average of the squared gradients are basically the first and second moments of those gradients. That is, they are the mean and uncentered variance of our gradients, respectively. So, for notation simplicity, let's denote $E[g]_t$ as m_t and $E[g^2]_t$ as v_t.

Therefore, we can rewrite equations *(16)* and *(17)* as follows:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$

We begin by setting the initial moments estimates to zero. That is, we initialize m_t and v_t with zeros. When the initial estimates are set to 0, they remain very small, even after many iterations. This means that they would be biased toward 0, especially when β_1 and β_2 are close to 1. So, to combat this, we compute the bias-corrected estimates of m_t and v_t by just dividing them by $1-\beta^t$, as follows:

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

Here, $\hat{m}_t$ and $\hat{v}_t$ are the bias-corrected estimates of m_t and v_t, respectively.

So, our final update equation is given as follows:

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

Now, let's understand how to implement Adam in Python.

First, let's define the `Adam` function, as follows:

```
def Adam(data, theta, lr = 1e-2, beta1 = 0.9, beta2 = 0.9, epsilon = 1e-6,
num_iterations = 1000):
```

Then, we initialize the first moment, `mt`, and the second moment, `vt`, with `zeros`:

```
    mt = np.zeros(theta.shape[0])
    vt = np.zeros(theta.shape[0])
```

For every iteration, we perform the following steps:

```
    for t in range(num_iterations):
```

Next, we compute the `gradients` with respect to `theta`:

```
        gradients = compute_gradients(data, theta)
```

Then, we update the first moment, `mt`, so that it's $m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$:

```
        mt = beta1 * mt + (1. - beta1) * gradients
```

Next, we update the second moment, `vt`, so that it's $v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$:

```
        vt = beta2 * vt + (1. - beta2) * gradients ** 2
```

Now, we compute the bias-corrected estimate of `mt`, that is, $\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$:

```
        mt_hat = mt / (1. - beta1 ** (t+1))
```

Next, we compute the bias-corrected estimate of `vt`, that is, $\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$:

```
        vt_hat = vt / (1. - beta2 ** (t+1))
```

Finally, we update the model parameter, `theta`, so that it's $\theta_t = \theta_{t-1} - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$:

```
        theta = theta - (lr / (np.sqrt(vt_hat) + epsilon)) * mt_hat

    return theta
```

Adamax – Adam based on infinity-norm

Now, we will look at a small variant of the Adam algorithm called **Adamax**. Let's recall the equation of the second-order moment in Adam:

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

As you may have noticed from the preceding equation, we scale the gradients inversely proportional to the L^2 norm of the current and past gradients (L^2 norm basically means the square of values):

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2)|g_t|^2$$

Instead of having just L^2, can we generalize it to the L^p norm? In general, when we have a large p for norm, our update would become unstable. However, when we set the p value to ∞, that is, when L^∞, the v_t equation becomes simple and stable. Instead of just parameterizing the gradients, g_t, alone, we also parameterize the decay rate, β_2. Thus, we can write the following:

$$v_t = \beta_2^\infty v_{t-1} + (1 - \beta_2^\infty)|g_t|^\infty$$

When we set the limits, p tends to reach infinity, and then we get the following final equation:

$$v_t = max(\beta_2^{t-1}|g_1|, \beta_2^{t-2}|g_2| \ldots \beta_2|g_{t-1}|, |g_t|)$$

You can check the paper listed in the *Further reading* section at the end of this chapter to see how exactly this is derived.

We can rewrite the preceding equation as a simple recursive equation, as follows:

$$v_t = max(\beta_2 \cdot v_{t-1}, |g_t|)$$

Computing m_t is similar to what we saw in the *Adaptive moment estimation* section, so we can write the following directly:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

By doing this, we can compute the bias-corrected estimate of m_t:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

Therefore, the final update equation becomes the following:

$$\boxed{\theta_t = \theta_{t-1} - \frac{\eta}{v_t}\hat{m}_t}$$

To better understand the Adamax algorithm, let's code it, step by step.

First, we define the Adamax function, as follows:

```
def Adamax(data, theta, lr = 1e-2, beta1 = 0.9, beta2 = 0.999, epsilon =
1e-6, num_iterations = 1000):
```

Then, we initialize the first moment, mt, and the second moment, vt, with zeros:

```
    mt = np.zeros(theta.shape[0])
    vt = np.zeros(theta.shape[0])
```

For every iteration, we perform the following steps:

```
    for t in range(num_iterations):
```

Now, we can compute the gradients with respect to theta, as follows:

```
        gradients = compute_gradients(data, theta)
```

Then, we compute the first moment, mt, as $m_t = \beta_1 m_{t-1} + (1-\beta_1)g_t$:

```
        mt = beta1 * mt + (1. - beta1) * gradients
```

Next, we compute the second moment, vt, as $v_t = max(\beta_2 \cdot v_{t-1}, |g_t|)$:

```
        vt = np.maximum(beta2 * vt, np.abs(gradients))
```

Now, we can compute the bias-corrected estimate of mt; that is, $\hat{m}_t = \frac{m_t}{1-\beta_1^t}$:

```
        mt_hat = mt / (1. - beta1 ** (t+1))
```

Update the model parameter, theta, so that it's $\theta_t = \theta_{t-1} - \frac{\eta}{v_t}\hat{m}_t$:

```
        theta = theta - ((lr / (vt + epsilon)) * mt_hat)
    return theta
```

Adaptive moment estimation with AMSGrad

One problem with the Adam algorithm is that it sometimes fails to attain optimal convergence, or it reaches a suboptimal solution. It has been noted that, in some settings, Adam fails to attain convergence or reach the suboptimal solution instead of a global optimal solution. This is due to exponentially moving the averages of gradients. Remember when we used the exponential moving averages of gradients in Adam to avoid the problem of learning rate decay?

However, the problem is that since we are taking an exponential moving average of gradients, we miss out information about the gradients that occur infrequently.

To resolve this issue, the authors of AMSGrad made a small change to the Adam algorithm. Recall the second-order moment estimates we saw in Adam, as follows:

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

In AMSGrad, we use a slightly modified version of v_t. Instead of using v_t directly, we take the maximum value of v_t until the previous step, as follows:

$$\hat{v}_t = \max(\hat{v}_{t-1}, v_t)$$

This will retain the informative gradients instead of being phased out due to the exponential moving average.

So, our final update equation becomes the following:

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

Now, let's understand how to code AMSGrad in Python.

First, we define the `AMSGrad` function, as follows:

```
def AMSGrad(data, theta, lr = 1e-2, beta1 = 0.9, beta2 = 0.9, epsilon =
1e-6, num_iterations = 1000):
```

Then, we initialize the first moment, `mt`, the second moment, `vt`, and the modified version of `vt`, that is, `vt_hat`, with `zeros`, as follows:

```
    mt = np.zeros(theta.shape[0])
    vt = np.zeros(theta.shape[0])
    vt_hat = np.zeros(theta.shape[0])
```

For every iteration, we perform the following steps:

```
    for t in range(num_iterations):
```

Now, we can compute the gradients with respect to `theta`:

```
        gradients = compute_gradients(data, theta)
```

Then, we compute the first moment, `mt`, as $m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$:

```
        mt = beta1 * mt + (1. - beta1) * gradients
```

Next, we update the second moment, `vt`, as $v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$:

```
       vt = beta2 * vt + (1. - beta2) * gradients ** 2
```

In AMSGrad, we use a slightly modified version of v_t. Instead of using v_t directly, we take the maximum value of v_t until the previous step. Thus, $\hat{v}_t = \max(\hat{v}_{t-1}, v_t)$ is implemented as follows:

```
        vt_hat = np.maximum(vt_hat,vt)
```

Here, we will compute the bias-corrected estimate of `mt`, that is, $\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$:

```
        mt_hat = mt / (1. - beta1 ** (t+1))
```

Now, we can update the model parameter, `theta`, so that it's $\theta_t = \theta_{t-1} - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$:

```
         theta = theta - (lr / (np.sqrt(vt_hat) + epsilon)) * mt_hat

    return theta
```

Nadam – adding NAG to ADAM

Nadam is another small extension of the Adam method. As the name suggests, here, we incorporate NAG into Adam. First, let's recall what we learned about in Adam.

We calculated the first and second moments as follows:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

Then, we calculated the bias-corrected estimates of the first and second moments, as follows:

$$\hat{m_t} = \frac{m_t}{1-\beta_1^t}$$

$$\hat{v_t} = \frac{v_t}{1-\beta_2^t}$$

Our final update equation of Adam is expressed as follows:

$$\theta_t = \theta_{t-1} - \frac{\eta}{\sqrt{v_t}+\epsilon} m_t$$

Now, we will see how Nadam modifies Adam to use Nesterov momentum. In Adam, we compute the first moment as follows:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$

We change this first moment so that it's Nesterov accelerated momentum. That is, instead of using the previous momentum, we use the current momentum and use that as a lookahead:

$$\tilde{m_t} = \beta_1^{t+1} m_t + (1-\beta_1^t) g_t$$

We can't compute the bias-corrected estimates in the same way as we computed them in Adam because, here, g_t comes from the current step, and m_t comes from the subsequent step. Therefore, we change the bias-corrected estimate step, as follows:

$$\hat{m_t} = \frac{m_t}{1-\prod_{i=1}^{t+1} \beta_1^i}$$

$$\hat{g_t} = \frac{g_t}{1-\prod_{i=1}^{t} \beta_1^i}$$

Thus, we can rewrite our first-moment equation as follows:

$$\tilde{m_t} = \beta_1^{t+1} \hat{m_t} + (1-\beta_1^t) \hat{g_t}$$

Therefore, our final update equation becomes the following:

$$\boxed{\theta_t = \theta_{t-1} - \frac{\eta}{\sqrt{v_t} + \epsilon}\tilde{m_t}}$$

Now let's see how we can implement the Nadam algorithm in Python.

First, we define the `nadam` function:

```
def nadam(data, theta, lr = 1e-2, beta1 = 0.9, beta2 = 0.999, epsilon =
1e-6, num_iterations = 500):
```

Then, we initialize the first moment, `mt`, and the second moment, `vt`, with zeros:

```
    mt = np.zeros(theta.shape[0])
    vt = np.zeros(theta.shape[0])
```

Next, we set `beta_prod` to `1`:

```
    beta_prod = 1
```

For every iteration, we perform the following steps:

```
    for t in range(num_iterations):
```

Then, we compute the gradients with respect to `theta`:

```
        gradients = compute_gradients(data, theta)
```

Afterward, we compute the first moment, `mt`, so that it's $m_t = \beta_1 m_{t-1} + (1 - \beta_1)g_t$:

```
        mt = beta1 * mt + (1. - beta1) * gradients
```

Now, we can update the second moment, `vt`, so that its' $v_t = \beta_2 v_{t-1} + (1 - \beta_2)g_t^2$:

```
        vt = beta2 * vt + (1. - beta2) * gradients ** 2
```

Now, we compute `beta_prod`; that is, $\prod_{i=1}^{t+1} \beta_1^i$:

```
        beta_prod = beta_prod * (beta1)
```

Next, we compute the bias-corrected estimate of mt so that it's $\hat{m_t} = \frac{m_t}{1 - \prod_{i=1}^{t+1} \beta_1^i}$:

```
        mt_hat = mt / (1. - beta_prod)
```

Then, we compute the bias-corrected estimate of gt so that it's $\hat{g_t} = \frac{g_t}{1 - \prod_{i=1}^{t} \beta_1^i}$:

```
        g_hat = grad / (1. - beta_prod)
```

From here, we compute the bias-corrected estimate of vt so that it's $\hat{v_t} = \frac{v_t}{1 - \beta_2^t}$:

```
        vt_hat = vt / (1. - beta2 ** (t))
```

Now, we compute mt_tilde so that it's $\tilde{m_t} = \beta_1^{t+1} \hat{m_t} + (1 - \beta_1^t) \hat{g_t}$:

```
        mt_tilde = (1-beta1**t+1) * mt_hat + ((beta1**t)* g_hat)
```

Finally, we update the model parameter, theta, by using $\theta_t = \theta_{t-1} - \frac{\eta}{\sqrt{v_t} + \epsilon} \tilde{m_t}$:

```
        theta = theta - (lr / (np.sqrt(vt_hat) + epsilon)) * mt_hat
    return theta
```

By doing this, we have learned about various popular variants of gradient descent algorithms that are used for training neural networks. The complete code to perform regression with all the variants of regression is available as a Jupyter Notebook at http://bit.ly/2XoW0vH.

Summary

We started off this chapter by learning about what convex and non-convex functions are. Then, we explored how we can find the minimum of a function using gradient descent. We learned how gradient descent minimizes a loss function by computing optimal parameters through gradient descent. Later, we looked at SGD, where we update the parameters of the model after iterating through each and every data point, and then we learned about mini-batch SGD, where we update the parameters after iterating through a batch of data points.

Going forward, we learned how momentum is used to reduce oscillations in gradient steps and attain convergence faster. Following this, we understood Nesterov momentum, where, instead of calculating the gradient at the current position, we calculate the gradient at the position the momentum will take us to.

We also learned about the Adagrad method, where we set the learning rate low for parameters that have frequent updates, and high for parameters that have infrequent updates. Next, we learned about the Adadelta method, where we completely do away with the learning rate and use an exponentially decaying average of gradients. We then learned about the Adam method, where we use both first and second momentum estimates to update gradients.

Following this, we explored variants of Adam, such as Adamax, where we generalized the L^2norm of Adam to L^∞, and AMSGrad, where we combated the problem of Adam reaching a suboptimal solution. At the end of this chapter, we learned about Nadam, where we incorporated Nesterov momentum into the Adam algorithm.

In the next chapter, we will learn about one of the most widely used deep learning algorithms, called **recurrent neural networks** (**RNNs**), and how to use them to generate song lyrics.

Questions

Let's recap on gradient descent by answering the following questions:

1. How does SGD differ from vanilla gradient descent?
2. Explain mini-batch gradient descent.
3. Why do we need momentum?
4. What is the motivation behind NAG?
5. How does Adagrad set the learning rate adaptively?
6. What is the update rule of Adadelta?
7. How does RMSProp overcome the limitations of Adagrad?
8. Define the update equation of Adam.

Further reading

For further information, refer to the following links:

- *Adaptive Subgradient Methods for Online Learning and Stochastic Optimization*, by John Duchi et al., `http://www.jmlr.org/papers/volume12/duchi11a/duchi11a.pdf`
- *Adadelta: An Adaptive Learning Rate Method*, by Matthew D. Zeiler, `https://arxiv.org/pdf/1212.5701.pdf`
- *Adam: A Method For Stochastic Optimization*, by Diederik P. Kingma and Jimmy Lei Ba, `https://arxiv.org/pdf/1412.6980.pdf`
- *On the Convergence of Adam and Beyond*, by Sashank J. Reddi, Satyen Kale, and Sanjiv Kumar, `https://openreview.net/pdf?id=ryQu7f-RZ`
- *Incorporating Nesterov Momentum into Adam*, by Timothy Dozat, `http://cs229.stanford.edu/proj2015/054_report.pdf`