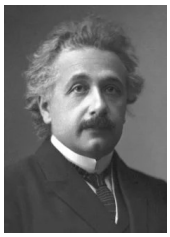
1. Optimization



I never failed in mathematics.
Before I was fifteen I had mastered differential and integral calculus.

— Albert Einstein

Gradient-based optimization is an essential tool for the field of machine learning. This chapter addresses the basic elements of calculus used to understand gradient-based methods such as gradient descent.

1.1 Fundamental Mathematical Elements

In this section, we present and define fundamental mathematical elements, including scalars, vectors, matrices, and tensors. Each of these mathematical elements is an abstraction that helps describe physical quantities and relationships.

1. Scalar: A scalar is a single number, often representing a quantity that has magnitude but no direction. Scalars are used to measure quantities such as temperature, mass, or time.

   - Example: Temperature at a point, $T = 25\ °C$.
   - Mathematical Notation: Scalars are typically denoted by lowercase or uppercase letters, e.g., a, b, c.

2. Vector: A vector is an ordered list of numbers that represents a quantity with both magnitude and direction. Vectors are often visualized as arrows in space, with the length representing the magnitude and the direction representing its orientation.

   - Example: Velocity in 3D space, $\mathbf{v} = (v_x, v_y, v_z)$.
   - Mathematical Notation: A vector is usually represented as a bold letter or with an arrow above, e.g., $\mathbf{v}$.

$$\mathbf{v} = \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix}$$

3. Matrix: A matrix is a rectangular array of numbers arranged in rows and columns. Matrices are used to represent linear transformations, systems of linear equations, and more.

   - Example: A 2x2 matrix representing a linear transformation in 2D space.

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$$

   - Mathematical Notation: Matrices are typically represented by uppercase bold letters, e.g., $\mathbf{A}$, $\mathbf{B}$.

4. Tensor: A tensor is a more general mathematical object that can be thought of as a multi-dimensional array of numbers. Tensors extend scalars, vectors, and matrices to higher dimensions and are used extensively in physics, engineering, and machine learning.

   - Example: A rank-3 tensor in 3D space.

$$\mathcal{T} = \left[t_{ijk}\right], \quad i, j, k = 1, 2, 3$$

   - Mathematical Notation: Tensors are usually denoted with calligraphic letters or bold, uppercase letters, e.g., $\mathcal{T}$, $\mathbf{T}$.
   - A scalar can be seen as a tensor of rank 0.
   - A vector is a tensor of rank 1.
   - A matrix is a tensor of rank 2.
   - Higher-rank tensors (rank 3 and above) represent more complex relationships. For example, a rank-3 tensor can be used to represent the stress or strain in a material in physics.

1.2 Convex Functions

A function $f(x)$ is convex if the line segment between any two points on the graph of the function lies above or on the graph. Mathematically, for $x_1, x_2 \in \mathbb{R}$ and $\lambda \in [0, 1]$:

$$f(\lambda x_1 + (1 - \lambda)x_2) \leq \lambda f(x_1) + (1 - \lambda)f(x_2)$$

To illustrate the formal definition of a convex function, we can create a plot showing a convex function $f(x) = x^2$ and a line segment connecting two points $(x_1, f(x_1))$ and

$(x_2, f(x_2))$ as shown in Figure 1.1. The convexity condition states that the function value at any convex combination of x_1 and x_2 is less than or equal to the corresponding convex combination of $f(x_1)$ and $f(x_2)$. Examples of convex functions are the quadratic function $f(x) = x^2$ and the exponential function $f(x) = e^x$.

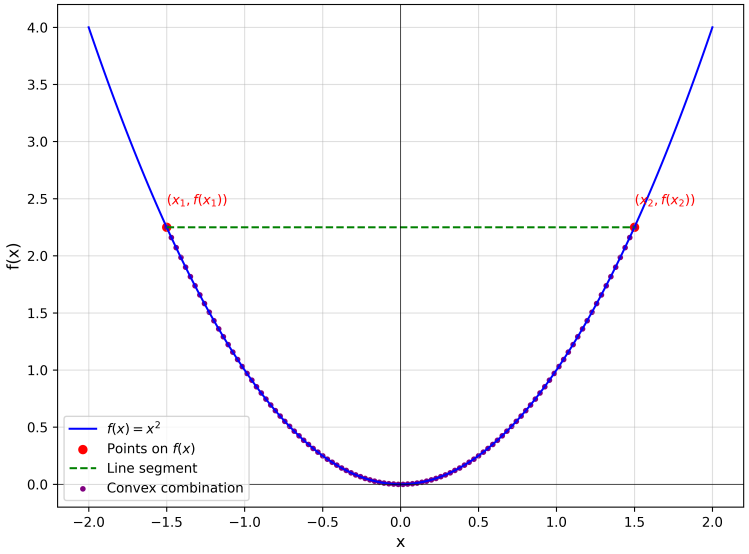


Figure 1.1: Illustration of the convex function definition. The blue curve represents $f(x) = x^2$, which is convex. The green dashed line connects the points $(x_1, f(x_1))$ and $(x_2, f(x_2))$. The purple points represent $f(\lambda x_1 + (1 - \lambda)x_2)$, which lie below or on the green line segment, satisfying the convexity condition.

A function $f(x)$ is non-convex if there exists at least one line segment between two points on the graph of the function that lies below the graph. It violates the convexity condition as shown in Figure 1.2. Examples of non-convex functions are cosine function $f(x) = \cos(x)$ and the quartic function with multiple minima $f(x) = x^4 - 4x^2 + 3$.

1.3 Derivative

The derivative in calculus is a way of measuring the rate of change of a function at a certain point. It can also be interpreted as the slope of the line that is tangent to the function's curve at that point. The derivative of a function $f(x)$ can be denoted by $f'(x)$ or $\frac{df(x)}{dx}$, where x is the input variable[1]. The derivative mathematically can

[1]A partial derivative is a derivative of a function of several variables with respect to one of those variables, while keeping the others constant. For example, if $f(x_1, x_2)$is a function of x_1 and x_2, then the

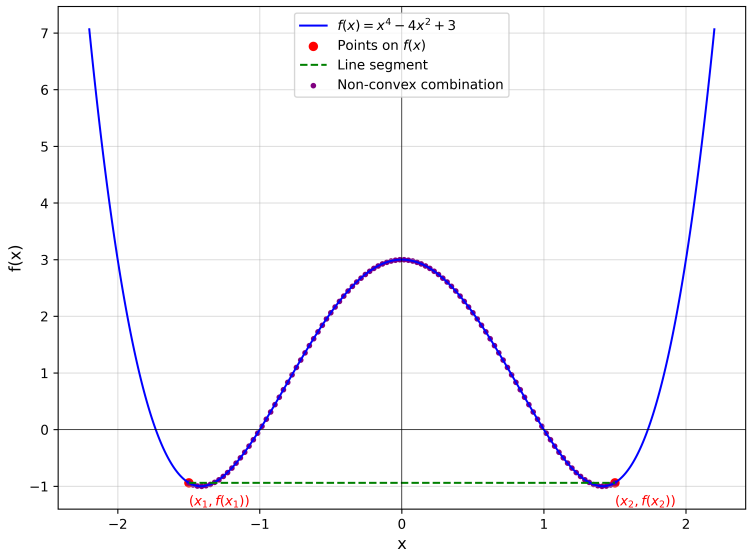


Figure 1.2: The plot shows curve with multiple local minima and maxima for the function ($f(x) = x^4 - 4x^2 + 3$), illustrating non-convexity.

be defined as follows:

$$f'(x) = \frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \tag{1.1}$$

if the limit exists. A function is not differentiable at a point if it is not continuous at that point. For example, the function $f(x) = |x|$ is not continuous at $x = 0$, so it has no derivative there.

Most machine learning algorithms are formulated as a minimization of a loss or objective function with respect to certain variables. To find a minimum of a function, there are different methods depending on the type and complexity of the function. Some of the common methods are:

- Sketching the function: This method involves plotting the graph of the function and visually identifying the lowest point on the graph. For example, the one-variable quadratic function has one global minimum[2] at $x = 1$:

$$f(x) = (x-1)^2 \tag{1.2}$$

partial derivative of f with respect to x_1 is denoted by $\frac{\partial f}{\partial x_1}$ and it is obtained by differentiating f with respect to x_1 and treating x_2 as a constant. Similarly, the partial derivative of f with respect to x_2 is denoted by $\frac{\partial f}{\partial x_2}$ and it is obtained by differentiating f with respect to x_2 and treating x_1 as a constant. Partial derivatives are used to measure the rate of change of a function along a specific direction or axis

[2]The difference between local and global minimum of a function is that a local minimum is the

where it is easy to find the minimum by inspecting the plot. However, this method is useful for simple functions that can be easily graphed, but it may not be accurate or feasible for more complicated functions.

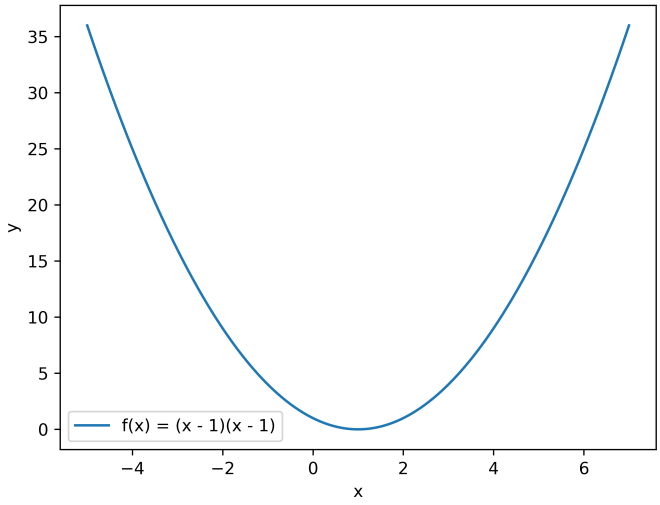


Figure 1.3: A plot of a simple quadratic function has a global minimum.

- Finding analytical solution: This method involves using calculus to find the derivative of the function and setting it equal to zero. This gives the critical points of the function, where the slope is zero or undefined. Figure 1.4 shows graphically why we set the first derivative to zero where the slope at the maximum and minimum is a horizontal line (i.e. the slope is zero).

  Then, using the second derivative test or the first derivative test, we can determine which critical points are local minima, local maxima, or neither. To find the minimum of the quadratic function in Equation (1.2), we differentiate it with respect to x and set the derivative to zero as follows (i.e. using the power rule):

$$f'(x) = 2(x - 1) = 0 \tag{1.3}$$

  Hence, the minimum happens at $x = 1$. In practice, this method for finding the minimum of a function does not scale well with the amount of training data commonly seen in machine learning problems.

point where the function value is smaller than (or equal to) the function values at nearby points, while a global minimum is the point where the function value is the smallest among all points in the domain. A function can have multiple local minima, but only one global minimum.

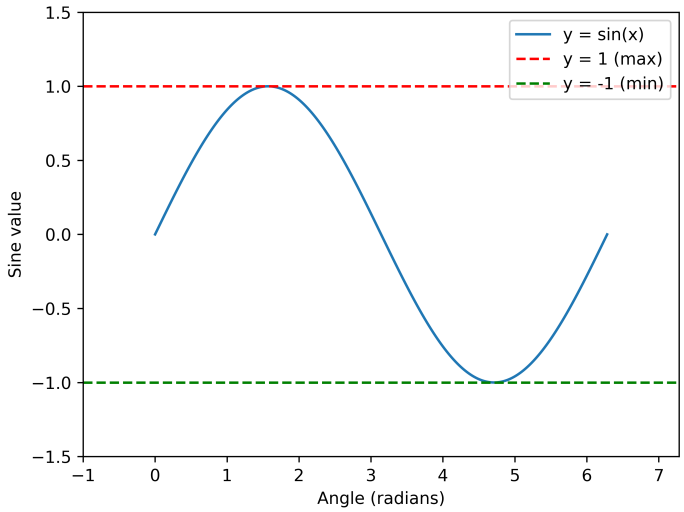


Figure 1.4: The slope at the maximum and minimum is a horizontal line (i.e. the slope is zero).

- Using gradient descent method: The gradient descent method is an iterative optimization algorithm that is used to find the minimum of a function by moving in the opposite direction of the gradient (or the slope) of the function at each point. This method will be detailed in the next section.

1.4 Gradient Descent

The gradient descent method involves starting from an initial guess and iteratively updating it by moving in the opposite direction of the gradient (the vector of partial derivatives) of the function. The gradient gives the direction of steepest ascent, so moving against it will lead to a descent. The step size is determined by a learning rate parameter that controls how fast or slow the algorithm converges. This method is useful for finding a local minimum of a function that may not have an analytical solution or may be too complex to solve by calculus. However, this method does not guarantee finding the global minimum of the function, and it may depend on the choice of initial guess and learning rate.
The gradient descent method works as follows (assuming the function has one variable only):

1. Start with an initial guess $x = x^{(0)}$ for the parameters of the function that need to be optimized.

2. Calculate the gradient of the function with respect to the parameters at the current point $g(x) = \frac{df(x)}{dx}|_{x^{(0)}}$.

3. Update the parameters by subtracting a fraction of the gradient from the current values. The fraction is called the learning rate and it controls how big or small the steps are.

$$x^{(1)} = x^{(0)} - \eta g(x) \tag{1.4}$$

   where $\eta > 0$ is the learning rate. When the gradient is positive (ascending), it means that the function is increasing in that direction. Therefore, we move against the gradient direction to find a lower point on the function, aiming to eventually reach a local minimum. Similarly, when the gradient is negative (descending), it means the function is already decreasing in that direction, but we still move against the gradient to continue finding a lower point and approach the local minimum. This way, we hope to eventually reach a local minimum of the function as well. This behavior is shown in Figure 1.5. In the next section, we show mathematically why we need to subtract a fraction of the gradient from the current values.

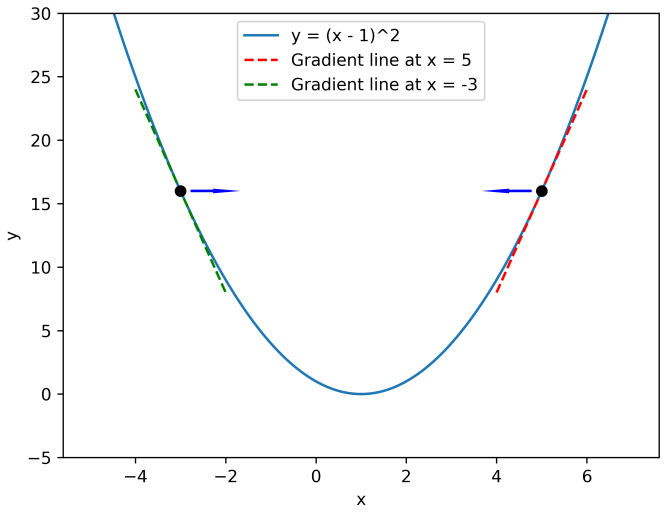


Figure 1.5: Function plot with gradient lines (ascending at $x = 5$ and descending at $x = -3$) and arrows pointing towards the minimum.

4. Repeat steps 2 and 3 until the gradient is close to zero or a maximum number of iterations is reached.

The gradient descent method is widely used in machine learning to train models by minimizing a loss function that measures the difference between the predicted and actual outputs. The gradient descent method can be applied to different types of functions, such as linear, quadratic, or non-linear functions. There are also different variants of the gradient descent method, such as batch gradient descent, stochastic gradient descent, mini-batch gradient descent, and momentum gradient descent, that differ in how they calculate and update the gradients.

1.4.1 Examples

In this subsection, I will use Python code to illustrate how the gradient descent algorithm works and how it can be applied to different functions of one variable or two variables.
The gradient descent algorithm is a method to find the minimum of a function by taking small steps in the direction of the steepest decrease. To apply this algorithm to the function $f(x) = (x-1)^2$, we need to first find its derivative, which is $g(x) = 2(x-1)$. The algorithm starts with an initial guess for x =3, and computes the value of $g(x)$ at $x = 3$. Then it updates x by subtracting an $\eta g(x)$ from it. This gives a new value for x that is closer to the minimum of the function. The algorithm repeats this process until it converges to a value of x that makes $g(x)$ very close to zero or a maximum number of epochs is reached. This value of $x = 1$ is the minimum of the quadratic function $f(x) = (x-1)^2$. The described algorithm can be implemented as a Python code as follows:

```
def grad(x):
    return 2.0 * (x-1.0)

x = 3.0
eta = 0.001
epochs = 50000
for i in range(epochs):
    x -= eta * grad(x)

print(x)
```

Listing 1.1: Python example for finding the minimum of a quadratic function in one variable.

On the other hand, we can use two different learning rates η_{x1}, η_{x2} for the function $f(x1, x2) = (x1-2)^2 + 10*(x2+3)^2$ because the function has different scales and curvatures along the $x1$ and $x2$ directions. If we use a single learning rate for both variables (e.g. $\eta_{x1} = \eta_{x2} = 0.1$), we might encounter a convergence problem. By using different learning rates for each variable $\eta_{x1} = 0.1$ and $\eta_{x2} = 0.05$, we can adjust the step size according to the shape of the function and find the minimum

more efficiently and accurately[3]. A Python implementation to find the minimum of this function is:

```
#Define the function of two variables
def f(x1, x2):
    return (x1 - 2)**2 + 10.0 * ((x2 + 3)**2)

#Define the partial derivatives of the function
def df_dx1(x1, x2):
    return 2 * (x1 - 2)

def df_dx2(x1, x2):
    return 20.0 * (x2 + 3)

#Define the learning rates for each variable
eta_x1 = 0.1 # Learning rate for x1
eta_x2 = 0.05 # Learning rate for x2

#Define the initial values for x1 and x2
x1 = 0.0
x2 = 0.0

#Define the tolerance for convergence
epsilon = 0.000001

#Define a variable to store the previous value of the function
prev_f = f(x1, x2)

#Start the gradient descent loop
steps = 0
while True:
    steps += 1
    #Update x1 and x2 using the gradient and the learning rates
    x1 = x1 - eta_x1 * df_dx1(x1, x2)
    x2 = x2 - eta_x2 * df_dx2(x1, x2)

    #Compute the current value of the function
    curr_f = f(x1, x2)

    #Check if the function value has decreased sufficiently
    if abs(curr_f - prev_f) < epsilon:
        break # Exit the loop

    #Update the previous value of the function
    prev_f = curr_f

#Print the final values of x1 and x2 and the minimum value of the
    function
print("x1 =", x1)
```

[3]You can play with the learning rates to study the convergence properties.

```
print("x2 =", x2)
print("f(x1, x2) =", curr_f)
print("steps for convergence =", steps)
```

Listing 1.2: Python example for finding the minimum of a quadratic function in two variables.

The adaptive learning rate is a technique that adjusts the learning rate dynamically based on the progress of the gradient descent algorithm. The idea is to use a larger learning rate when the function is far from the minimum and a smaller learning rate when the function is close to the minimum. This way, we can speed up the convergence and avoid overshooting or oscillating. One method to implement the adaptive learning rate is to use the Hessian matrix, which is the matrix of second-order partial derivatives of the function. The Hessian matrix captures the curvature of the function and can be used to scale the gradient vector according to the shape of the function. By using the inverse of the Hessian matrix as a multiplier for the gradient vector, we can obtain a more accurate direction and step size for each iteration of the gradient descent algorithm. The next section will detail these techniques.

1.5 Gradient Descent using Taylor's Series

In mathematics, Taylor's series can be used to make a first-order approximation to a scalar loss function $f(\mathbf{x})$ around the current point vector $\mathbf{x}^{(t)} \in \mathbb{R}^d$ given the first derivative of the function at that point:

$$f(\mathbf{x}^{(t+1)}) \approx f(\mathbf{x}^{(t)}) + (\mathbf{x}^{(t+1)} - \mathbf{x}^{(t)})^T \mathbf{g}, \tag{1.5}$$

where $\mathbf{g}$ is the gradient vector at the point $\mathbf{x}^{(t)}$. It can be written as well as follows:

$$f(\mathbf{x}^{(t)} + \Delta\mathbf{x}) \approx f(\mathbf{x}^{(t)}) + \Delta\mathbf{x}^T \mathbf{g}, \tag{1.6}$$

where $\Delta\mathbf{x} = \mathbf{x}^{(t+1)} - \mathbf{x}^{(t)}$. In order to decrease the loss function $f(\mathbf{x}^{(t)} + \Delta\mathbf{x})$, the term $\Delta\mathbf{x}^T\mathbf{g}$ has to be a negative value. Hence,

$$\Delta\mathbf{x}^T\mathbf{g} < 0 \tag{1.7}$$

Let's consider the cosine of angle between the two vectors $\Delta\mathbf{x}^T$ and $\mathbf{g}$:

$$\cos\theta = \frac{\Delta\mathbf{x}^T\mathbf{g}}{|\Delta\mathbf{x}^T||\mathbf{g}|} \tag{1.8}$$

$\cos\theta$ lies between -1 and 1 i.e. $-1 \leq \cos\theta \leq +1$. Hence,

$$-|\Delta\mathbf{x}^T||\mathbf{g}| \leq \Delta\mathbf{x}^T\mathbf{g} \leq |\Delta\mathbf{x}^T||\mathbf{g}| \tag{1.9}$$

Now we want the dot product to be as negative as possible (so that loss can be as low as possible). We can set the dot product to be $-|\Delta\mathbf{x}^T||\mathbf{g}|$ where $cos\theta$ has to be equal to -1 corresponds to $\theta = 180^o$. Therefore,

$$\Delta\mathbf{x} = -\mathbf{g} \tag{1.10}$$

This result explains why we move in the opposite direction of the gradient as we described in Section 1.4.
Intuitively, since the first-order approximation is good only for small $\Delta\mathbf{x}$, we want to choose a small $\eta > 0$ to make $\Delta\mathbf{x}$ small in magnitude. η is called the learning rate. Hence,

$$\Delta\mathbf{x} = -\eta\mathbf{g} \tag{1.11}$$

1.6 Gradient Descent Limitations

The gradient of a function at a specific point represents the direction of the steepest descent of the function at that point. In other words, it points in the direction in which the function decreases most rapidly. On the other hand, the contours[4] of a function are curves along which the function has the same value, so there is no change in the function value as you move along the contour.
Since the gradient points in the direction of maximum decrease, it is orthogonal (perpendicular) to the direction along which there is no change in the function value, which is the contour. This is true for any scalar function.
One limitation of the gradient descent is the zigzag effect. The zigzag effect occurs because the gradient at each point points in the direction of the steepest descent when moving downhill and may not necessarily point directly toward the minimum. As the algorithm moves along the steepest slope, it overshoots the direction of the minimum, and in the next step, it must correct its course. This leads to a back-and-forth zigzagging pattern as the algorithm iteratively converges to the minimum as shown in Figure 1.6. This zigzag effect slows down the convergence of the algorithm. Zigzag effect of gradient descent does not happen for the loss functions that have circular contours or equal curvature in all dimensions or directions. For example, a function like $f(x, y) = x^2 + y^2$ has circular contours and does not have zigzag effect. A straight line to the minimum for these functions is followed by gradient descent as shown in Figure 1.7.
Hessian-based optimization methods, like Newton's method or Quasi-Newton methods (such as BFGS and L-BFGS), make use of second-order information to help guide the optimization process more effectively [1]. They use the Hessian matrix or

[4] A contour of a function (e.g. Rosenbrock function) is a curve connecting points with the same function value. A contour is defined mathematically as the set of points (x, y) such that $f(x, y) = c$, where c is a constant.

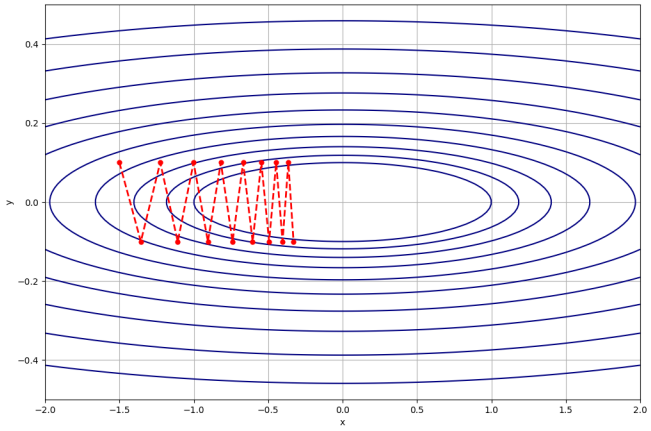


Figure 1.6: The zigzag effect of the gradient descent algorithm.

its approximation to adjust the step size and direction, which can reduce the zigzagging effect and potentially lead to faster convergence. Using Taylor's expansion, the second-order approximation is given by

$$f(\mathbf{x}) \approx f(\mathbf{x}^{(t)}) + (\mathbf{x} - \mathbf{x}^{(t)})^T \mathbf{g} + \frac{1}{2}(\mathbf{x} - \mathbf{x}^{(t)})^T \mathbf{H}(\mathbf{x} - \mathbf{x}^{(t)}), \tag{1.12}$$

where $\mathbf{H}$ is the Hessian matrix at the point $\mathbf{x}^{(t)}$. The local Hessian matrix is a symmetric matrix and is defined by

$$\mathbf{H} \equiv \frac{\partial f(\mathbf{x})}{\partial \mathbf{x}_i \partial \mathbf{x}_j}|_{\mathbf{x}^{(t)}} \tag{1.13}$$

$$\mathbf{H} = \begin{bmatrix} \frac{\partial^2 f(\mathbf{x})}{\partial \mathbf{x}_1^2} & \frac{\partial^2 f(\mathbf{x})}{\partial \mathbf{x}_1 \partial \mathbf{x}_2} & \cdots \\ \frac{\partial^2 f(\mathbf{x})}{\partial \mathbf{x}_2 \partial \mathbf{x}_1} & \frac{\partial^2 f(\mathbf{x})}{\partial \mathbf{x}_2^2} & \cdots \\ \vdots & \vdots & \ddots \end{bmatrix} \tag{1.14}$$

The basic idea of Newton's method is to minimize the quadratic approximation of the cost function $f(\mathbf{x})$ around the current point $\mathbf{x}^{(t)}$. Equation (1.12) can be rewritten as follows:

$$\Delta f(\mathbf{x}^{(t)}) = f(\mathbf{x}) - f(\mathbf{x}^{(t)}) \approx \Delta \mathbf{x}^T \mathbf{g} + \frac{1}{2}\Delta \mathbf{x}^T \mathbf{H} \Delta \mathbf{x} \tag{1.15}$$

Differentiating the above equation with respect $\Delta \mathbf{x}$ and setting the output to zero to get the minimum:

$$\mathbf{g} + \mathbf{H}\Delta \mathbf{x} = 0, \tag{1.16}$$

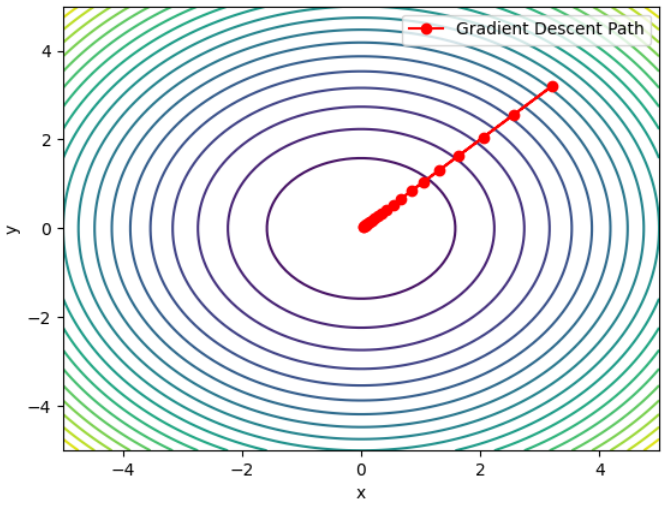


Figure 1.7: The zigzag effect of the gradient descent algorithm does not happen for the functions that have circular contours.

Hence, the Newton's update rule is given by

$$\Delta \mathbf{x} = -\eta \mathbf{H}^{-1}\mathbf{g} \tag{1.17}$$

When the matrix $\mathbf{H}$ equals to the identity matrix[5] (i.e. taking the same step in each direction), we reach the gradient descent update rule described in Equation (1.11). It's important to note that while Hessian-based methods can help overcome the zigzagging effect, they often come with their own challenges, such as increased computational complexity and the need to compute or approximate the Hessian matrix. In practice, these trade-offs need to be considered when selecting an optimization algorithm for a particular problem [2].
Adaptive learning rate methods overcome the zigzag effect in the gradient descent algorithm as well. They are addressed in the next subsection.

1.6.1 Adaptive learning Rate

Adaptive learning rate methods overcome the zigzag effect in gradient descent by adjusting the learning rate for each parameter during the optimization process. These methods take into account the history of gradients, the magnitude of the gradients,

[5]An identity matrix is a square matrix in which all the elements of the principal diagonal are ones and all other elements are zero.

or both, to determine an appropriate learning rate for each parameter. As a result, adaptive learning rate methods can effectively navigate the loss surface and reduce oscillations and zigzagging.
Some popular adaptive learning rate methods include:

- **Momentum**: Momentum can reduce the zigzag problem by accumulating a vector that smooths out the gradient updates and aligns them with a consistent direction. Hence, it converges faster and more reliably than the gradient descent [3]. The momentum with gradient descent algorithm updates the parameters $\mathbf{x}$ as follows:

$$m^{(t)} = \beta m^{(t-1)} + (1-\beta) g^{(t)} \tag{1.18}$$
$$\mathbf{x}^{(t+1)} = \mathbf{x}^{(t)} - \eta m^{(t)} \tag{1.19}$$

  where $g^{(t)}$ is the gradient of the loss function at time step t, $m^{(t)}$ is the vector that accumulates the past gradients, β is the momentum coefficient, and η is the learning rate.

- **AdaGrad**: AdaGrad accumulates the squared gradients for each parameter in a diagonal matrix and uses this information to adapt the learning rate for each parameter. Parameters with larger accumulated squared gradients have their learning rate reduced, while those with smaller accumulated squared gradients have their learning rate increased. This makes AdaGrad well-suited for problems with sparse gradients or features that occur with varying frequency [4]. The AdaGrad algorithm updates the variables $\mathbf{x}$ as follows:

$$v^{(t)} = v^{(t-1)} + g^{(t)}.g^{(t)} \tag{1.20}$$
$$\mathbf{x}^{(t+1)} = \mathbf{x}^{(t)} - \frac{\eta}{\sqrt{v^{(t)}} + \epsilon} g^{(t)} \tag{1.21}$$

  where $g^{(t)}$ is the gradient of the loss function at time step t, $v^{(t)}$ is the sum of the squares of the gradients up to time step t, η is the learning rate, and ϵ is a small constant to prevent division by zero

- **RMSprop**: RMSprop is an improvement over Adagrad that uses an exponentially decaying average of the squared gradients instead of the cumulative sum. This makes RMSprop more robust to situations where the accumulated squared gradients can grow indefinitely, causing the learning rate to shrink too much [5]. The RMSprop algorithm updates the variables $\mathbf{x}$ as follows:

$$v^{(t)} = \beta v^{(t-1)} + (1-\beta) g^{(t)} . g^{(t)} \tag{1.22}$$

$$\mathbf{x}^{(t+1)} = \mathbf{x}^{(t)} - \eta \frac{g^{(t)}}{\sqrt{v^{(t)}} + \epsilon} \tag{1.23}$$

where $g^{(t)}$ is the gradient of the loss function at time step t, $v^{(t)}$ is the estimate of the second moment of the gradients, β is the decay rate for the moment, η is the learning rate, and ϵ is a small constant to prevent division by zero.

- **Adam**: Adam combines the ideas of RMSprop and momentum-based methods. It computes the first moment (mean) and the second moment (uncentered variance) of the gradients, and uses these moments to adapt the learning rate for each parameter. This helps Adam to navigate the loss surface more effectively, reducing zigzagging and achieving faster convergence [6]. The Adam algorithm updates the variables $\mathbf{x}$ as follows:

$$m^{(t)} = \beta_1 m^{(t-1)} + (1-\beta_1) g^{(t)} \tag{1.24}$$

$$v^{(t)} = \beta_2 v^{(t-1)} + (1-\beta_2) g^{(t)} . g^{(t)} \tag{1.25}$$

$$\hat{m}^{(t)} = \frac{m^{(t)}}{1-\beta_1^t} \tag{1.26}$$

$$\hat{v}^{(t)} = \frac{v^{(t)}}{1-\beta_2^t} \tag{1.27}$$

$$\mathbf{x}^{(t+1)} = \mathbf{x}^{(t)} - \eta \frac{\hat{m}^{(t)}}{\sqrt{\hat{v}^{(t)}} + \epsilon} \tag{1.28}$$

where $g^{(t)}$ is the gradient of the loss function at time step t, $m^{(t)}$ and $v^{(t)}$ are the estimates of the first and second moments of the gradients, $\hat{m}^{(t)}$ and $\hat{v}^{(t)}$ are the bias-corrected estimates[6], β_1 and β_2 are the decay rates for the moments, η is the learning rate, and ϵ is a small constant to prevent division by zero. Although the Adam algorithm is the state-of-the-art learning algorithm, the algorithms suffer from worse generalization[7] performance than stochastic

[6] The Adam algorithm suffers from a bias problem due to the initialization of the first and second moment estimates at zero. This means that the algorithm tends to underestimate the true values of these moments at the beginning of the training, leading to inaccurate gradient updates. To overcome this problem, the Adam algorithm uses a bias correction term that divides each moment estimate by a factor that accounts for the decay rates of the moments. This way, the algorithm can adjust for the bias and converge faster and more reliably.

[7] The generalization is measured using the test set performance.

gradient descent despite their faster training speed [7]. Hence, a practical recipe for training is to start with Adam for a few epochs and then switch to the gradient descent algorithm. In addition, the Adam algorithm accumulates two statistics for each variable or parameter during the optimization process.

1.7 Assignment

Using Adam algorithm, find the minimum of the function $f(x1, x2) = (x1 - 2)^2 + 10 * (x2 + 3)^2$.