Samuel J. Cavazos

Algebraic Geometry & Machine Learning

in Python for parseltongues

December 3, 2024

Springer Mathematics

To Adalyn & Luka

Preface

Algebraic Geometry & Machine Learning explores the fascinating intersection of two powerful fields: algebraic geometry and machine learning. While machine learning has revolutionized industries ranging from healthcare to finance, algebraic & algebraic geometry has long been a cornerstone of advanced mathematics, providing the tools to understand the curvature and structure of spaces. The synergy between these domains offers profound insights into the mathematical foundations of machine learning and equips practitioners with powerful techniques to build more robust and explainable models.

The motivation for this book stems from the desire to bridge the gap between theory and practice. As machine learning algorithms grow increasingly complex, understanding their underlying mechanics becomes not just an academic exercise but a necessity for developing effective, interpretable, and scalable solutions. Algebraic geometry provides a rigorous framework to address questions about curvature, optimization, and structure within the high-dimensional spaces where these models operate.

This book begins with the basics of machine learning, using linear regression and differential geometry as a gateway to understanding the fundamental principles of modeling and optimization. Through accessible explanations and hands-on examples, we build a foundation that extends naturally to more complex architectures, including neural networks. From there, we delve into the tools of algebraic geometry, showing how concepts such as gradients, manifolds, and geodesics inform and enhance machine learning algorithms.

Key features of this book include:

- **Practical Examples**: Python-based implementations and visualizations to solidify theoretical concepts.
- **Mathematical Rigor**: Detailed derivations and explanations that connect machine learning practices to their geometric and mathematical underpinnings.
- **Interdisciplinary Approach**: Insights from both machine learning and algebraic geometry, fostering a holistic understanding of modern AI techniques.

This book is intended for a diverse audience:

- **Mathematicians** intrigued by the applications of algebraic geometry in contemporary AI.
- **Machine Learning Engineers** seeking a deeper understanding of the mathematical principles behind their tools.
- **Students and Educators** looking for an accessible yet rigorous resource to explore the intersection of these fields.

As we journey through this book, we will not only develop a deeper appreciation for the beauty of algebraic geometry but also see how it empowers us to design better, more interpretable machine learning models. Whether you are a practitioner, researcher, or student, this book invites you to explore a rich and rewarding mathematical landscape that underpins some of the most transformative technologies of our time.

The Rio Grande Valley
2025

Samuel J. Cavazos
DHR Health

Acknowledgements

This project would not have been possible without the inspiration and support of my colleagues at DHR Health, whose insights and discussions have enriched the ideas presented here. I am especially grateful to the readers and learners who engage with this material—your curiosity and passion continue to drive this work forward.

Contents

Acronyms

List of abbreviations and symbols used in the book.

$\forall$	for all
$\in$	in, element of
$\vec{v}$	A vector
$\mathbb{C}$	The field of complex numbers
$\mathcal{L}$	Loss function
$\mathbb{Q}$	The field of rational numbers
$\mathbb{R}$	The field of real numbers
$\mathbb{Z}$	The ring of integers
$\mathbb{Z}/n\mathbb{Z}$	The ring of integers modulo n
ML	Acronym for Machine-Learning

Part I
Regression Models

Chapter 1
Linear Regression

Abstract This chapter introduces the principles of linear regression as a foundation for understanding the connection between differential geometry and machine learning. A simple linear model $M(x) = x \cdot W + b$ is constructed, and a loss function is used to quantify prediction errors. The chapter details the derivation of gradients for the loss with respect to the model parameters W and b, providing insights into how these gradients guide the optimization process.

1.1 Linear Models

Linear regressions are a fundamental tool in statistics and machine learning for modeling the relationship between a dependent variable y and one or more independent variables x. The simplest form of linear regression is a univariate linear model, which assumes a linear relationship between y and x of the form $y = x \cdot W + b$, where $W, b \in \mathbb{R}$ are real numbers. The model parameters W and b are learned from a dataset of input-output pairs $\{(x_i, y_i)\}_{i=1}^{N}$ by minimizing a loss function that quantifies the prediction errors of the model.

Let's define a more general version of the linear model. Suppose we have n samples, each with d features, and our target is to predict m outputs for each sample. We can represent the input data as a matrix $X \in \mathbb{R}^{n \times d}$ and the output data as a matrix $Y \in \mathbb{R}^{n \times m}$.

The linear model $M(X)$ is then defined as:

$$M(X) = X \cdot W^{\intercal} + b,$$

where:

- $X \in \mathbb{R}^{n \times d}$ is the input matrix,
- $W \in \mathbb{R}^{m \times d}$ is the weight matrix,
- $b \in \mathbb{R}^{n \times m}$ is the bias matrix.

The bias matrix b is constructed from the bias vector $\vec{b} \in \mathbb{R}^m$ by replicating it n times. In other words, each row of the bias matrix b is the bias vector $\vec{b}$. $\vec{b}$ is said to be *broadcasted* to the shape of b.

1.2 Univariate Linear Models

Let's construct some data to work with that follows a somewhat linear trend and build a machine-learning model from scratch. We'll take the function $f(x) = x^2 + 2 \cdot x + 1$ over a random sample of points in $[0, 10]$ and add some uniform noise. Next, we'll separate the synthetic data into training and test sets. This will allow us to train the model on the training data and evaluate its performance on the test data. Once we feel confident in the performance of our model, we may train using the entire dataset. For splitting, we'll use the `train_test_split` function from the `sklearn` library.

```
#!pip install matplotlib
#!pip install numpy
#!pip install sklearn

def f(x):
    return x**2 + 2*x + 1

# Plot using matplotlib
import matplotlib.pyplot as plt
import numpy as np
import random

# Define the true function
def f(x):
    return x**2 + 2*x + 1

# Generate data
np.random.seed(42)
x = np.linspace(0, 10, 200)
y_true = f(x)
y_data = y_true + np.random.uniform(-10, 10, size=x.shape)

# split into training and test datasets (80% training, 20% test)
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(x, y_data,
    test_size=0.2)
print(f'Train size: {len(x_train)}')
print(f'Test size: {len(x_test)}')

# Plot the data and the true function, coloring the training and
    test data differently
plt.plot(x, y_true,color='black', label='True function')
plt.scatter(x_train, y_train, color='darkred', label='Training
    data')
plt.scatter(x_test, y_test, color='blue', label='Test data')
plt.legend()
```

```
plt.savefig('fig1.png')
plt.show()
```

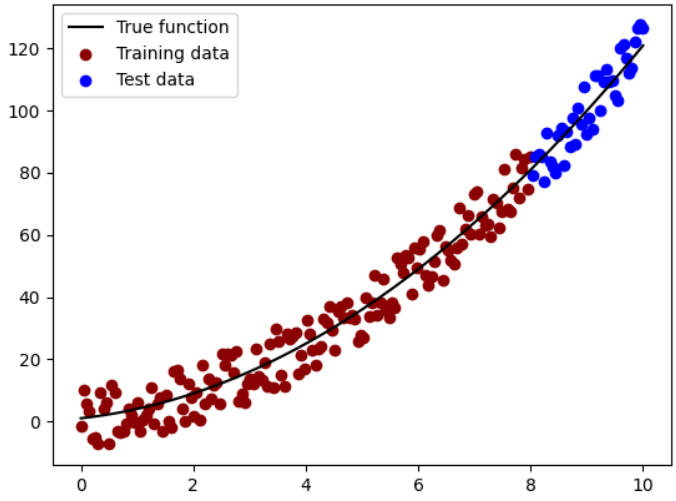


Fig. 1.1 Data generated from the function $f(x) = x^2 + 2 \cdot x + 1$ with added noise.

In Figure 1.1, the *best fit* for this data is the function we used to construct it. Of course, we usually don't know the equation for the best fit beforehand, but our goal is to create a model to approximate this line as closely as possible.

Let us start by constructing a simple machine-learning model for linear regression with no hidden layers, which essentially means there are no intermediate computations between the input and the output in our model.

Our goal is to build a machine-learning model $M : [0, 10] \longrightarrow \mathbb{R}$ of the form

$$M(x) = x \cdot W + b,$$

where $W \in \mathbb{R}$ and $b \in \mathbb{R}$. Here, W is called the *weight* and b is called the *bias*.

Here, we define our linear model:

```
# Linear model
def linear_model(x, w, b):
    return x * w + b
```

In machine-learning, a model is initialized with random weights and biases, which are then corrected during training by minimizing a *loss function*. Let's start by choosing some random W and b.

```
import random
```

```
# Initialize parameters
w = random.uniform(-1, 1)  # Random initial weight
b = random.uniform(-1, 1)  # Random initial bias

print(f'Initial weight: {w}')
print(f'Initial bias: {b}')
```

```
Initial weight: 0.5136609336515561
Initial bias: 0.39026605372156786
```

Given that the weight and bias was chosen at random, we don't expect it to perform very well on our data, and indeed that is the case, as shown in Figure 1.2.

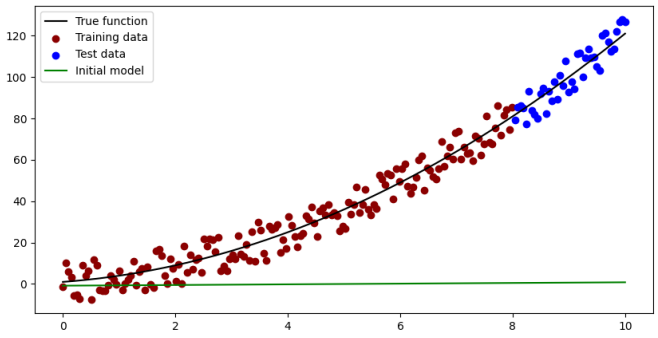


Fig. 1.2 Initial model prediction with random weight and bias.

Let's work on improving the model. Improving our model will involve tweaking W and b to better fit the model using a process called **gradient descent**.

1.2.1 Gradient Descent for Univariate Linear Models

We first define a **loss function** to measure how our model is performing. This function will quantify the difference between the model's predictions and the actual data. A common loss function for linear regression is the *Mean Squared Error* (MSE), which is defined as:

$$\mathcal{L}(Y_{\text{pred}}, Y) = \frac{1}{n}\|Y_{\text{pred}} - Y\|^2 = \frac{1}{n}\sum_{i=1}^{n}(y_{\text{pred}}^{(i)} - y^{(i)})^2, \tag{1.1}$$

where:

- $Y_{\text{pred}}, Y \in \mathbb{R}^n$ are the predicted and true values, respectively,

- n is the number of samples,
- $y_{\text{pred}}^{(i)}$ and $y^{(i)}$ are the predicted and true values for the i-th sample, respectively.

This loss function penalizes large deviations of Y_{pred} from Y by squaring the differences.

```
# Mean Squared Error Loss
def mse_loss(y_pred, y_true):
    return np.mean((y_pred - y_true)**2)

# Compute predictions for the training data
y_pred_train = [linear_model(p, w, b) for p in x_train]

print(f'50th sample target: {y_train[50]}')
print(f'50th prediction: {y_pred_train[50]}')
print(f'Loss at 50th sample: {mse_loss(y_pred_train[50], y_train
    [50])}')

print('Total Loss over all samples:', mse_loss(np.array(
    y_pred_train), np.array(y_train)))
```

```
50th sample target: -3.2562046439706913
50th prediction: 0.7774476620016353
Loss at 50th sample: 16.270350925475867
Total Loss over all samples: 2899.2086153263763
```

Our goal is to minimize this loss function.

One thing to note about this loss function is that it is a differentiable function. Recal from vector calculus that the **gradient** of a differentiable funtion f is a vector field ∇f whose value at point p is a vector that points towards the direction of steepest ascent.

Understanding the gradients of the loss function with respect to the model parameters—specifically, the weight W and bias b—is crucial in machine learning, particularly when employing optimization techniques like gradient descent. Our goal is to minimize the loss function.

The gradients $\frac{\partial \mathcal{L}}{\partial W}$ and $\frac{\partial \mathcal{L}}{\partial b}$ indicate how sensitive the loss function $\mathcal{L}$ is to changes in the parameters W and b. In essence, they provide the direction and rate at which $\mathcal{L}$ increases or decreases as we adjust these parameters.

By computing these gradients, we can iteratively update W and b to minimize the loss function, thereby improving the model's performance. This process is the foundation of the gradient descent optimization algorithm.

1.2.1.1 Gradients

1. *Gradient with Respect to Weight W:*
   The partial derivative $\frac{\partial \mathcal{L}}{\partial W}$ measures how the loss changes with respect to the weight W. A positive derivative suggests that increasing W will increase the loss,

while a negative derivative indicates that increasing W will decrease the loss. By moving W in the direction opposite to the gradient, we can reduce the loss.
For the i-th data point, let $y_{\text{pred}}^{(i)} = x_i \cdot W^\top + b$ be the predicted value while $y^{(i)}$ denotes the true value. Mathematically, this gradient is computed as:

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial W} &= \frac{\partial}{\partial W}\left(\frac{1}{n}\sum_{i=1}^{n}(y_{\text{pred}}^{(i)} - y^{(i)})^2\right) \\
&= \frac{1}{n}\sum_{i=1}^{n}\frac{\partial}{\partial W}\left((x_i \cdot W^\top + b - y^{(i)})^2\right) \quad \text{(Substitute model equation)} \\
&= \frac{1}{n}\sum_{i=1}^{n} 2 \cdot (y_{\text{pred}}^{(i)} - y^{(i)}) \cdot x_i \quad \text{(Chain rule)} \\
&= \frac{2}{n}\sum_{i=1}^{n}(y_{\text{pred}}^{(i)} - y^{(i)}) \cdot x_i.
\end{aligned}$$

Thus, we find that:

$$\frac{\partial \mathcal{L}}{\partial W} = \frac{2}{n}\sum_{i=1}^{n}(y_{\text{pred}}^{(i)} - y^{(i)}) \cdot x_i, \tag{1.2}$$

or equivalently in matrix form:

$$\frac{\partial \mathcal{L}}{\partial W} = \frac{2}{n}\left(Y_{\text{pred}} - Y\right)^\top X, \tag{1.3}$$

where X is the input matrix of shape $n \times d$, and $Y_{\text{pred}}, Y \in \mathbb{R}^n$.

2. *Gradient with Respect to Bias b:*
Similarly, the partial derivative $\frac{\partial \mathcal{L}}{\partial b}$ measures how the loss changes with respect to the bias b. Adjusting b in the direction opposite to this gradient will help minimize the loss.
This gradient is computed as:

$$\frac{\partial \mathcal{L}}{\partial b} = \frac{\partial}{\partial b}\left(\frac{1}{n}\sum_{i=1}^{n}(y_{\text{pred}}^{(i)} - y^{(i)})^2\right),$$

which simplifies to:

$$\frac{\partial \mathcal{L}}{\partial b} = \frac{2}{n}\sum_{i=1}^{n}(y_{\text{pred}}^{(i)} - y^{(i)}), \tag{1.4}$$

or equivalently:

$$\frac{\partial \mathcal{L}}{\partial b} = \frac{2}{n}\mathbf{1}^\top\left(Y_{\text{pred}} - Y\right), \tag{1.5}$$

where $\mathbf{1}$ is a vector of ones of size n.
Proof of this equation is left as an exercise for the reader.

With that, we can compute the gradients in Python:

```
# Compute gradients
def compute_gradients(x, y_true, w, b):
    y_pred = linear_model(x, w, b)
    error = y_pred - y_true
    dw = 2 * np.mean(error * x)
    db = 2 * np.mean(error)
    return dw, db
```

Now that we have a way of computing the partial derivatives of $\mathcal{L}$ with respect to W and b, we can visualize the *gradient field*. For a given $p = (W, b) \in \mathbb{R}^2$, the gradient $\nabla\mathcal{L}$ at p is a vector that points towards the rate of fastest increase. In the following code, we compute these vectors on a grid. We also include a 2D contour plot of the loss function $\mathcal{L}$. Our initial weight W and bias b are marked on the plot by a red dot.

```
import numpy as np
import matplotlib.pyplot as plt

# Gradient Descent parameters
alpha = 0.001  # Learning rate
epochs = 1000  # Number of iterations

# Create a grid of w and b values for contour and quiver plotting
w_vals = np.linspace(-10, 20, 50)
b_vals = np.linspace(-20, 10, 50)
W, B = np.meshgrid(w_vals, b_vals)

# Compute the loss for each combination of w and b in the grid
Z = np.array([mse_loss(linear_model(x_train, w, b), y_train) for
    w, b in zip(np.ravel(W), np.ravel(B))])
Z = Z.reshape(W.shape)

# Compute the gradient field
dW = np.zeros(W.shape)
dB = np.zeros(B.shape)
for i in range(W.shape[0]):
    for j in range(W.shape[1]):
        dw, db = compute_gradients(x_train, y_train, W[i, j], B[i
    , j])
        dW[i, j] = dw
        dB[i, j] = db

# Plot the cost function contour, gradient field, and gradient
    descent path
plt.figure(figsize=(10, 5))

# Contour plot of the loss function
cp = plt.contour(W, B, Z, levels=np.logspace(-1, 3, 20), cmap='
    viridis')
plt.colorbar(cp)
plt.xlabel('Weight (w)')
plt.ylabel('Bias (b)')
plt.title('Cost Function Contour, Gradient Field, and Gradient
    Descent Path')
```

```
35
36 # Quiver plot of the gradient field
37 plt.quiver(W, B, dW, dB, angles='xy', scale_units='xy', scale=2,
       color='blue', alpha=0.5,headwidth=6, headlength=6)
38 # plot initial weight, bias
39 plt.plot(w, b, 'ro', label='Initial (weight, bias)')
40 plt.legend()
41 plt.grid(True)
42
43 plt.savefig('gradient-field-1.png')
44
45 plt.show()
```

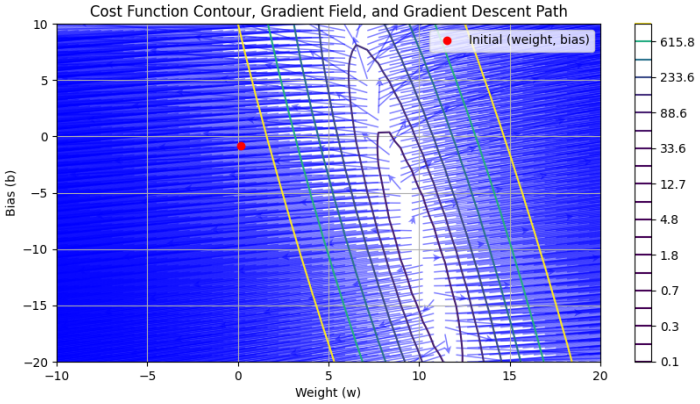


Fig. 1.3 Contour plot of the loss function, gradient field, and gradient descent path.

Since our goal is to minimize the loss function and these vectors are pointing towards the steepest ascent of the loss function with respect to W and b, we minimize by moving in the opposite direction of the gradients. This process is fundamental to optimization algorithms like gradient descent and is refered to as **backpropagation** in the realm of machine-learning.

1.2.1.2 Gradient Descent & Backward Propagation

Gradient descent is an optimization algorithm that iteratively updates the model parameters in the direction opposite to the gradients of the loss function. This process continues until the loss is minimized. **Backpropagation** is the process of computing these gradients and updating the model parameters.

The parameter updates are performed iteratively using the following rules:

1. Weight update:

$$W \leftarrow W - \alpha \frac{\partial \mathcal{L}}{\partial W}$$

Here, α is the learning rate, a hyperparameter that controls the step size of each update. The term $\frac{\partial \mathcal{L}}{\partial W}$ represents the gradient of the loss function with respect to the weight. By subtracting this scaled gradient from the current weight, we move W in the direction that decreases the loss.

2. Bias update:

$$b \leftarrow b - \alpha \frac{\partial \mathcal{L}}{\partial b}$$

Similarly, $\frac{\partial \mathcal{L}}{\partial b}$ is the gradient of the loss function with respect to the bias. Updating b in this manner adjusts the model's predictions to better fit the data.

The **learning rate** determines how large a step we take in the direction of the negative gradient. A small α leads to slow convergence, while a large α might cause overshooting the minimum, leading to divergence. Choosing an appropriate learning rate is crucial for effective training.

The gradients $\frac{\partial \mathcal{L}}{\partial W}$ and $\frac{\partial \mathcal{L}}{\partial b}$ indicate the direction in which the loss function increases most rapidly. By moving in the opposite direction (hence the subtraction), we aim to find the parameters that minimize the loss.

We repeat this process over and over again. Each time we do it is referred to as an **epoch**.

```
# Store parameters for plotting
w_history = [w]
b_history = [b]
loss_history = [mse_loss(linear_model(x_train, w, b), y_train)]

# Gradient Descent loop
for epoch in range(epochs):
    dw, db = compute_gradients(x_train, y_train, w, b)
    w = w - alpha * dw # Update the weight
    b = b - alpha * db # Update the bias

    w_history.append(w) # Add to weight tracker
    b_history.append(b) # Add to bias tracker
    loss_history.append(mse_loss(linear_model(x_train, w, b),
    y_train)) # Add overall loss to loss tracker

# Convert history lists to numpy arrays for easier slicing
w_history = np.array(w_history)
b_history = np.array(b_history)

# Compute the loss for each combination of w and b in the grid
Z = np.array([mse_loss(linear_model(x_train, w, b), y_train) for
    w, b in zip(np.ravel(W), np.ravel(B))])
Z = Z.reshape(W.shape)

# Compute the gradient field
dW = np.zeros(W.shape)
dB = np.zeros(B.shape)
for i in range(W.shape[0]):
```

```
    for j in range(W.shape[1]):
        dw, db = compute_gradients(x_train, y_train, W[i, j], B[i
    , j])
        dW[i, j] = dw
        dB[i, j] = db

# Print initial (weight, bias)
print(f'Initial (weight, bias): ({w_history[0]}, {b_history[0]})'
    )
# Print final (weight, bias)
print(f'Final (weight, bias): ({w_history[-1]}, {b_history[-1]})'
    )

# Plot the cost function contour, gradient field, and gradient
    descent path
plt.figure(figsize=(10, 5))

# Contour plot of the loss function
cp = plt.contour(W, B, Z, levels=np.logspace(-1, 3, 20), cmap='
    viridis')
plt.colorbar(cp)
plt.xlabel('Weight (w)')
plt.ylabel('Bias (b)')
plt.title('Cost Function Contour, Gradient Field, and Gradient
    Descent Path')

# Quiver plot of the gradient field
plt.quiver(W, B, dW, dB, angles='xy', scale_units='xy', scale=1,
    color='blue', alpha=0.5)

# Plot the gradient descent path
plt.plot(w_history, b_history, 'ro-', markersize=3, linewidth=1,
    label='Gradient Descent Path')
# Plot the initial weight, bias
plt.plot(w_history[0], b_history[0], 'ro', label='Initial (weight
    , bias)')

# Add arrows to indicate direction of descent
for i in range(1, len(w_history)):
    plt.arrow(w_history[i-1], b_history[i-1],
              w_history[i] - w_history[i-1],
              b_history[i] - b_history[i-1],
              head_width=0.05, head_length=0.1, fc='red', ec='
    red')

plt.legend()
plt.grid(True)
plt.savefig('gradient-field-2.png')
plt.show()
```

```
Initial (weight, bias): (0.5136609336515561, 0.39026605372156786)
Final (weight, bias): (10.472640107432522, -5.487933673141372)
```

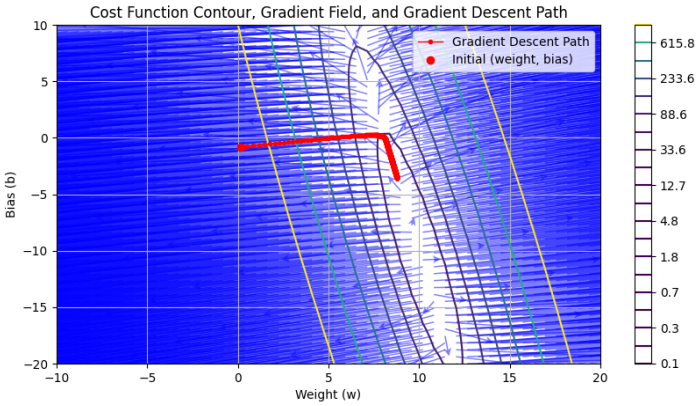


Fig. 1.4 Contour plot of the loss function, gradient field, and gradient descent path.

Since our weight W and bias b together form a point $(W, b) \in \mathbb{R}^2$, the loss function $\mathcal{L}$ forms a 3-dimensional surface. The visualization below shows the path taken during gradient descent on the surface of the loss function $\mathcal{L}$. The initial point (W, b) is in green. The path moves towards $\mathcal{L}$'s minimum.

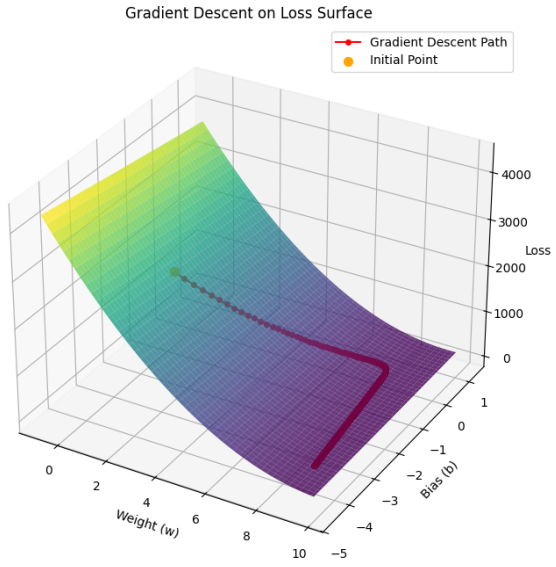


Fig. 1.5 Gradient descent path on the loss function surface. The code used to generate this visualization can be found in A.1.1 of the Appendix.

Finally, we visualize our initial (green) and final (red) linear model on a graph, alongside the data and true line of best fit (orange).

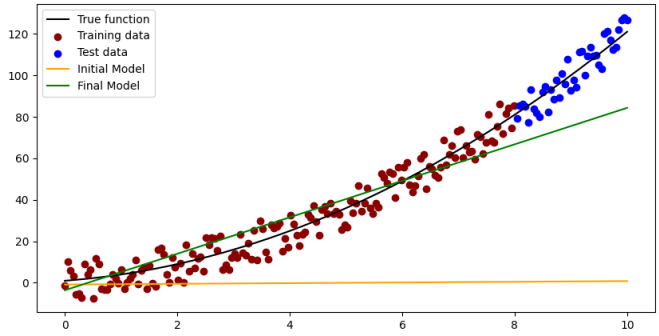


Fig. 1.6 Initial and final linear models compared to the true line of best fit.

1.2.2 Univariate Linear Models with Hidden Layers

We build upon the ideas from the previous section by incorporating a *hidden layer* into our model, allowing it to learn intermediate representations and capture more complex patters in the data. The reason why we call it a hidden layer is because it is not directly connected to the input or output of the model. This technique involves embedding our data into higher-dimensional spaces. Higher-dimensional spaces allow for the transformation of data in a way that makes patterns, relationships, or structures more linearly separable. In lower dimensions, data that appears entangled or inseparable can often be separated in a higher-dimensional space.

A classic example that demonstrates the concept of non-linear separability in lower dimensions but linear separability in higher dimensions is the *circle classification problem*. Here, data points inside a circle belong to one class, while those outside belong to another. This problem is not linearly separable in two dimensions, but becomes linearly separable when mapped to higher-dimensional spaces using the radius as a new feature. See A.1.2 in the Appendix for the code used to generate the visualizations below.

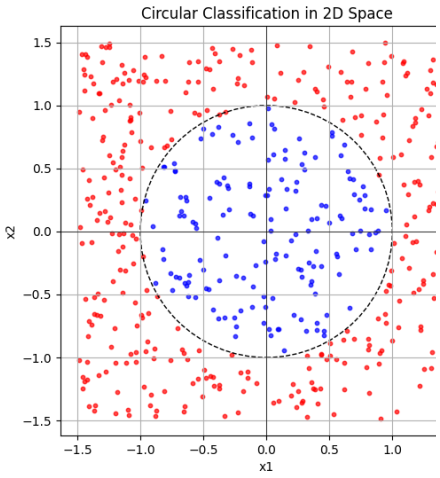


Fig. 1.7 Circle classification problem in 2D.

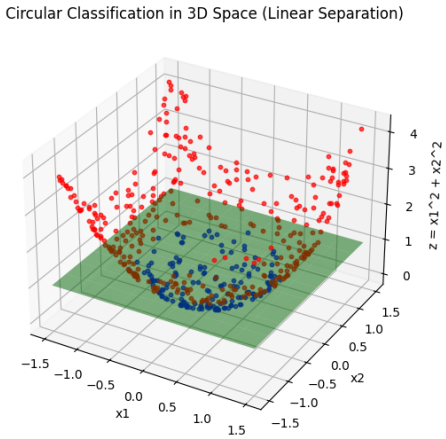


Fig. 1.8 Circle classification problem in 3D.

Model with a Hidden Layer

To extend the linear model to include a hidden layer, we define the architecture mathematically as follows:

1. **Input Layer:**

- The input matrix $X \in \mathbb{R}^{n \times d}$ represents a batch of n samples, where each row corresponds to one data sample, and d is the number of input features.
- For inference, the model operates on a single sample (a row vector $x \in \mathbb{R}^{1 \times d}$) (see A.1.3 in the Appendix for why inference on a single sample works even though the model is trained on batches).

2. **Hidden Layer:**

   - The hidden layer applies a linear transformation followed by a non-linear activation. Mathematically:

     $$Z_1 = XW_1 + \mathbf{1}b_1^\top,$$

     where:
     – $W_1 \in \mathbb{R}^{d \times h}$ is the weight matrix,
     – $b_1 \in \mathbb{R}^h$ is the bias vector (broadcasted across all samples),
     – $\mathbf{1} \in \mathbb{R}^{n \times 1}$ is a column vector of ones to handle the broadcast addition,
     – $Z_1 \in \mathbb{R}^{n \times h}$ is the resulting matrix after the linear transformation.
     The dimension h is referred to as the *hidden dimension*, or `hidden_dim`.
   - A non-linear activation function σ is applied element-wise to Z_1 to produce:

     $$A_1 = \sigma(Z_1),$$

     where $A_1 \in \mathbb{R}^{n \times h}$ is the activation output. We will discuss activation functions in more detail later.

3. **Output Layer:**

   - The output layer applies another linear transformation to the hidden representation A_1, mapping it to the final prediction:

     $$Z_2 = A_1 W_2 + \mathbf{1}b_2,$$

     where:
     – $W_2 \in \mathbb{R}^{h \times m}$ is the weight matrix,
     – $b_2 \in \mathbb{R}^m$ is the bias vector (broadcasted across all samples),
     – $Z_2 \in \mathbb{R}^{n \times m}$ is the intermediate output before any final activation.
   - If a final activation function ϕ is applied, the prediction becomes:

     $$Y_{\text{pred}} = \phi(Z_2),$$

     where ϕ is the activation function. For regression tasks, this step is often omitted, so:

     $$Y_{\text{pred}} = Z_2.$$

Summary of Transformations

Given $X \in \mathbb{R}^{n \times d}$, the transformations in the model are:

1. Hidden Layer:

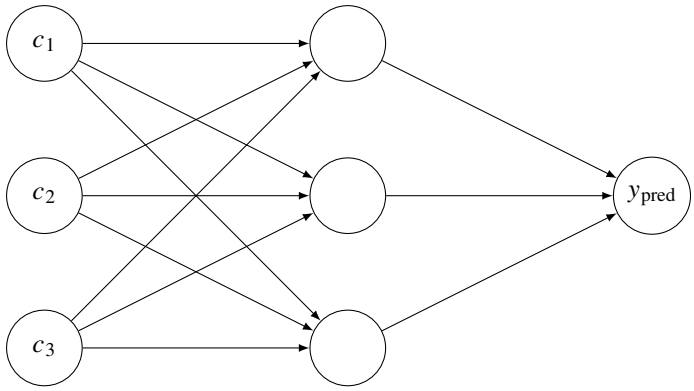
$$Z_1 = XW_1 + \mathbf{1}b_1^\top, \quad A_1 = \sigma(Z_1), \quad Z_1, A_1 \in \mathbb{R}^{n \times h}.$$

2. Output Layer:

$$Z_2 = A_1W_2 + \mathbf{1}b_2, \quad Y_{\text{pred}} = \phi(Z_2), \quad Z_2, Y_{\text{pred}} \in \mathbb{R}^{n \times m}.$$

You will often see graphs representing neural networks with nodes and edges, like the one below. Each node represents a *neuron*, and each edge represents a *connection* between neurons. The connections are weighted by the parameters W and b. The activation functions are applied at each neuron, transforming the input data.

Input Layer Hidden Layer Output Layer



The nodes labeled $c_1, \ldots, c_3$ correspond to the features (columns) of the input matrix $X \in \mathbb{R}^{n \times d}$. Mathematically, this layer simply passes the input features as is, preparing them for the hidden layer. In the hidden layer, each node aggregates contributions from all input features through a linear transformation.

Truthfully, these graphs aren't necessary for understanding the model, and in fact I still have a rather difficult time interpreting them. However, they are useful for visualizing the model and understanding the flow of data through the network. To some people.

Layers are connected by an **activation function**. Activation functions should satisfy the following criteria:

- *Non-linearity*: The function must be non-linear to allow the model to learn complex patterns.

- *Differentiability*: The function should be differentiable on its domain to faciliate gradient-based optimization methods like backpropagation. When we dive into models over more abstract rings, we will see how this relates to the concept of *derivations* over rings.
- *Bounded output*: Having a bounded output helps in stabilizing the learning process and prevents extreme activations.
- *Monotonicity*: A monotonic function ensures consistent gradients, aiding in stable and efficient training.
- *Computational efficiency*: The function should be computationally efficient to evaluate and differentiate.

A good example of such a function is the *sigmoid* activation function, defined as

$$\sigma(z) = \frac{1}{1 + e^{-z}}.$$

This function maps any real number to the range $(0, 1)$, making it useful for many regression and classification problems. The sigmoid function is differentiable, monotonic, and computationally efficient, making it a popular choice in neural networks.

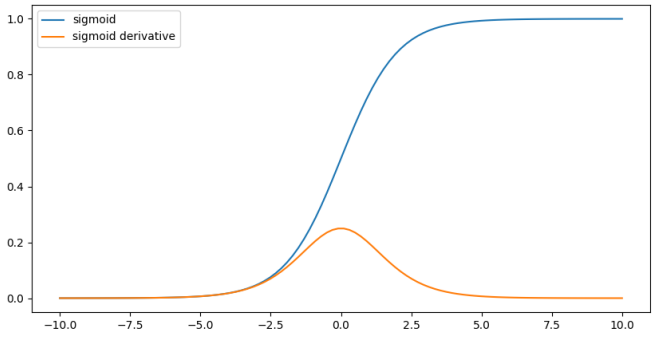


Fig. 1.9 The sigmoid activation function and its derivative.

Let's implement a univariate linear model with a hidden layer with hidden dimension $d = 2$ and a sigmoid activation function. We will use the same data as before.

```
import numpy as np
import matplotlib.pyplot as plt
import random

# Reshape data for neural network
x = x.reshape(-1, 1)
y_data = y_data.reshape(-1, 1)

```

```
# Initialize parameters
input_dim = x.shape[1]  # Number of input features
hidden_dim = 2          # Number of neurons in the hidden layer
output_dim = y_data.shape[1]  # Number of output neurons

# Weights and biases
np.random.seed(42)
W1 = np.random.randn(input_dim, hidden_dim) * 0.01
b1 = np.zeros((1, hidden_dim))
W2 = np.random.randn(hidden_dim, output_dim) * 0.01
b2 = np.zeros((1, output_dim))

# Fetch initial model predictions
Z1 = np.dot(x, W1) + b1
A1 = sigmoid(Z1)
Z2 = np.dot(A1, W2) + b2
y_pred_initial = Z2

# Learning rate
alpha = 0.01

# Training loop
epochs = 10000
m = x.shape[0]  # Number of training examples
loss_history = []

for epoch in range(epochs):
    # Forward propagation
    Z1 = np.dot(x, W1) + b1
    A1 = sigmoid(Z1)
    Z2 = np.dot(A1, W2) + b2
    y_pred = Z2  # Linear activation for output layer

    # Compute loss (Mean Squared Error)
    loss = (1 / (2 * m)) * np.sum((y_pred - y_data) ** 2)
    loss_history.append(loss)

    # Backward propagation
    dZ2 = y_pred - y_data
    dW2 = (1 / m) * np.dot(A1.T, dZ2)
    db2 = (1 / m) * np.sum(dZ2, axis=0, keepdims=True)
    dA1 = np.dot(dZ2, W2.T)
    dZ1 = dA1 * sigmoid_derivative(Z1)
    dW1 = (1 / m) * np.dot(x.T, dZ1)
    db1 = (1 / m) * np.sum(dZ1, axis=0, keepdims=True)

    # Update parameters
    W1 = W1 - alpha * dW1
    b1 = b1 - alpha * db1
    W2 = W2 - alpha * dW2
    b2 =  b2 - alpha * db2

    # Print loss every 1000 epochs
    if epoch % 1000 == 0:
```

```
        print(f'Epoch {epoch}, Loss: {loss}')

# Predictions
Z1 = np.dot(x, W1) + b1
A1 = sigmoid(Z1)
Z2 = np.dot(A1, W2) + b2
y_pred = Z2

# Plotting
plt.figure(figsize=(10, 5))
plt.plot(x, y_true, color='black', label='True function')
plt.scatter(x_train, y_train, color='darkred', label='Training
    data')
plt.scatter(x_test, y_test, color='blue', label='Test data')
plt.plot(x, y_pred_initial, label='Initial Model Prediction',
    color='orange')
plt.plot(x, y_pred, label='Model Prediction', color='red')
plt.xlabel('Input Feature')
plt.ylabel('Target Value')
plt.title('Neural Network with One Hidden Layer')
plt.legend()
plt.savefig('neural-network1.png')
plt.show()
```

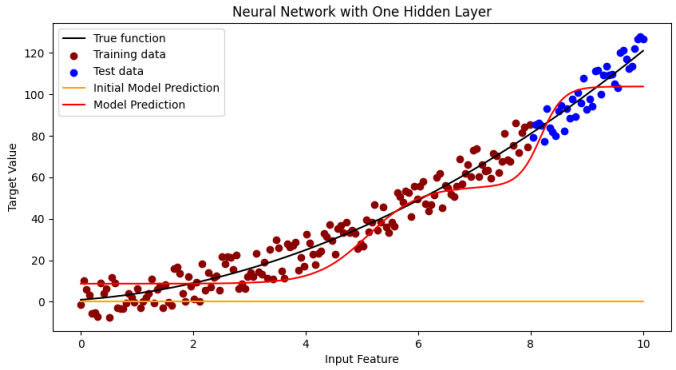


Fig. 1.10 Univariate linear model with a single hidden layer of dimension $d = 2$. The initial model's state is shown, as well as its final state after training.

As you can see, the model has learned that the data is not linear and has adjusted its weights and biases to better fit the data.

Now let's train a similar model but with a hidden dimension of $d = 10$.

```
import numpy as np
import matplotlib.pyplot as plt
import random

```

```
# Reshape data for neural network
x = x.reshape(-1, 1)
y_data = y_data.reshape(-1, 1)

# Initialize parameters
input_dim = x.shape[1]  # Number of input features
hidden_dim = 10         # Number of neurons in the hidden layer
output_dim = y_data.shape[1]  # Number of output neurons

# Weights and biases
np.random.seed(42)
W1 = np.random.randn(input_dim, hidden_dim) * 0.01
b1 = np.zeros((1, hidden_dim))
W2 = np.random.randn(hidden_dim, output_dim) * 0.01
b2 = np.zeros((1, output_dim))

# Fetch initial model predictions
Z1 = np.dot(x, W1) + b1
A1 = sigmoid(Z1)
Z2 = np.dot(A1, W2) + b2
y_pred_initial = Z2

# Learning rate
alpha = 0.01

# Training loop
epochs = 10000
m = x.shape[0]  # Number of training examples
loss_history = []

for epoch in range(epochs):
    # Forward propagation
    Z1 = np.dot(x, W1) + b1
    A1 = sigmoid(Z1)
    Z2 = np.dot(A1, W2) + b2
    y_pred = Z2  # Linear activation for output layer

    # Compute loss (Mean Squared Error)
    loss = (1 / (2 * m)) * np.sum((y_pred - y_data) ** 2)
    loss_history.append(loss)

    # Backward propagation
    dZ2 = y_pred - y_data
    dW2 = (1 / m) * np.dot(A1.T, dZ2)
    db2 = (1 / m) * np.sum(dZ2, axis=0, keepdims=True)
    dA1 = np.dot(dZ2, W2.T)
    dZ1 = dA1 * sigmoid_derivative(Z1)
    dW1 = (1 / m) * np.dot(x.T, dZ1)
    db1 = (1 / m) * np.sum(dZ1, axis=0, keepdims=True)

    # Update parameters
    W1 -= alpha * dW1
    b1 -= alpha * db1
    W2 -= alpha * dW2
```

```
    b2 -= alpha * db2

    # Print loss every 1000 epochs
    if epoch % 1000 == 0:
        print(f'Epoch {epoch}, Loss: {loss}')

# Predictions
Z1 = np.dot(x, W1) + b1
A1 = sigmoid(Z1)
Z2 = np.dot(A1, W2) + b2
y_pred = Z2

# Plotting
plt.scatter(x, y_data, label='Noisy Data', color='blue', alpha
    =0.5)
plt.plot(x, y_pred_initial, label='Initial Model Prediction',
    color='orange')
plt.plot(x, y_true, label='True Function', color='green')
plt.plot(x, y_pred, label='Model Prediction', color='red')
plt.xlabel('Input Feature')
plt.ylabel('Target Value')
plt.title('Neural Network with One Hidden Layer')
plt.legend()
plt.show()
```

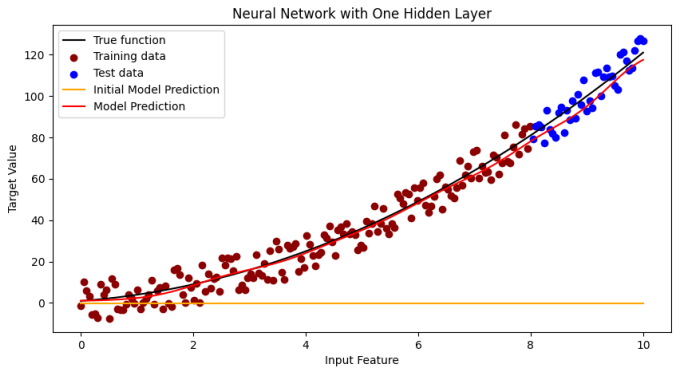


Fig. 1.11 Univariate linear model with a single hidden layer of dimension $d = 10$. The initial model's state is shown, as well as its final state after training.

As you can see, the higher-dimensional hidden layer allows the model to learn more complex patterns in the data and follows the true line of best fit more closely than the model with $d = 2$. Furthermore, it also performs well on the test data which was excluded during training.

1.3 Multivariate Linear Models

The next step in our journey is the multivariate linear model with no hidden layers. This model is an extension of the univariate model, allowing for multiple input features. The model definition is the same as that of the univarite case:

$$M(x) = W^\intercal \cdot x + b,$$

where:

- W is a **weight matrix** of shape $d \times m$,
- x is a d-dimensional input vector,
- b is an m-dimensional **bias vector**.

If we n samples with d features each, we can represent the input data as a matrix $X \in \mathbb{R}^{n \times d}$, where each row X_i represents a sample.

In the previous section, we utilized concepts from differential geometry to minimize the loss function and visualize the process of gradient descent. This was feasible since the model was simple and the loss function was differentiable. However, as we move to more complex models, we may not have the luxury offered by real-valued functions.

Let's look at the *Tips* dataset from the `seaborn` package as an example.

```
# %!pip install seaborn

# Import iris dataset using seaborn
import seaborn as sns
data = sns.load_dataset('tips')

data
```

	total_bill	tip	sex	smoker	day	time	size
242	17.82	1.75	Male	No	Sat	Dinner	2
73	25.28	5.00	Female	Yes	Sat	Dinner	2
43	9.68	1.32	Male	No	Sun	Dinner	2
86	13.03	2.00	Male	No	Thur	Lunch	2
105	15.36	1.64	Male	Yes	Sat	Dinner	2

Table 1.1 Sample data from `seaborn`'s *Tips* dataset

Our goal is the predict the tip amount based on the information available. The first thing you'll notice is that this dataset contains non-numerical features. How would we include such features in our model?

1.3.1 Encoding Categorical Data

Since our machine-learning algorithm works with numerical data, we need to convert the categorical data in the dataset into a numerical format. Two common techniques for this are **one-hot encoding** and **label encoding**.

One might numerically encode the `day` column so that Sunday is 0, Monday is 1, and so on. In doing so, you are defining an *encoding*, a numerical representation of the string. Your encoding is ordered in a sense. But when dealing with more general categorical data, we don't always have a natural ordering. Once we touch on the concept of *Rings*, we will see that the days of the week form a ring under addition modulo 7, denoted $\mathbb{Z}/\mathbb{Z}_7$. For now, we will proceed not as mathematicians but as machine-learning engineers.

Definition 1.1 An **embedding** is a structure-preserving map from one mathematical object to another. X is said to be embedded in Y if there exists an injective function $f : X \to Y$ such that $f(x) = f(y)$ if and only if $x = y$.

1.3.1.1 Label Encoding Categorical Data

Label Encoding is a technique that assigns a unique integer to each category in the data. This method is simpler than one-hot encoding but may introduce an ordinal relationship between the categories that does not exist in the data. In essense, we are creating an embedding of the categorical data into $\mathbb{Z}$, the integers.

You can also encode the `day` column in the *Tips* dataset using `scikit-learn`'s `LabelEncoder`:

```
# Label Encode the ‘day‘ column
from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
data['day_encoded'] = le.fit_transform(data['day'])

# Save the encoder to a pickle file
import pickle
with open('label_encoder.pkl', 'wb') as f:
    pickle.dump(le, f)

data.sample(5)
```

total_bill	tip	sex	smoker	day	time	size	day_LabelEncoded
17.59	2.64	Male	No	Sat	Dinner	3	1
38.07	4.00	Male	No	Sun	Dinner	3	2
22.76	3.00	Male	No	Thur	Lunch	2	3
13.81	2.00	Male	No	Sun	Dinner	2	2
23.95	2.55	Male	No	Sun	Dinner	2	2

The result is `day_LabelEncoded`, a new column that contains the encoded values for the `day` column. However, this encoding method introduces an ordinal relationship between the days that does not exist in the data. For example, Thursday was encoded as 3 while Friday was encoded as 0. Ths embedding doesn't preserve the ordered structure of the days of the week.

1.3.1.2 Ordinal Encoding Categorical Data

Ordinal Encoding is similar to label encoding, but assigns an integer to each category in the data based on the order in which they appear. This method preserves the ordinal relationship between the categories but may introduce an artificial ranking that does not exist in the data.

```
# Ordinal encode the ‘day‘ column
from sklearn.preprocessing import OrdinalEncoder
oe = OrdinalEncoder(categories=[['Thur', 'Fri', 'Sat', 'Sun']])
data['day_OrdinalEncoded'] = oe.fit_transform(data[['day']])

# Save the encoder to a pickle file
with open('ordinal_encoder.pkl', 'wb') as f:
    pickle.dump(oe, f)

data.sample(5)
```

total_bill	tip	sex	smoker	day	time	size	day_LabelEncoded	day_OrdinalEncoded
15.42	1.57	Male	No	Sun	Dinner	2	2	3
48.33	9.00	Male	No	Sat	Dinner	4	1	2
7.74	1.44	Male	Yes	Sat	Dinner	2	1	2
18.64	1.36	Female	No	Thur	Lunch	3	3	0
34.65	3.68	Male	Yes	Sun	Dinner	4	2	3

This does a better job at capturing the ordinal relationship between the days of the week since now Thursday comes before Friday. Still, it doesn't capture the cyclical nature of the days of the week. At the end of the week, we return to the beginning of the week, but our embedding doesn't reflect this. This is because our data inherits the structure of the ring we have embedded it in. In this case, we have embedded the days of the week in $\mathbb{Z}$, the integers, which is not cyclical.

There is a third approach we could take: *one-hot encoding*. It is a mathematically interesting approach.

1.3.1.3 One-Hot Encoding Categorical Data

One-Hot Encoding is a technique that creates a binary column for each category in the data. Each column represents a category, and a 1 in that column indicates the

presence of the category in the data. This method preserves the categorical nature of the data without introducing an ordinal relationship between the categories.

```
# One-Hot encode the ‘day‘ column
from sklearn.preprocessing import OneHotEncoder

ohe = OneHotEncoder(sparse_output=False,categories=[['Thur', 'Fri', 'Sat', 'Sun']])
day_ohe = ohe.fit_transform(data[['day']])
day_ohe_df = pd.DataFrame(day_ohe, columns=[f'day_{day}' for day in ohe.categories[0]])
# Merge the one-hot encoded columns with the original dataframe
data = pd.concat([data, day_ohe_df], axis=1)

print(data.sample(5).to_latex(index=False, float_format="%.2f"))
```

total_bill	tip	sex	smoker	day	time	size	day_Thur	day_Fri	day_Sat	day_Sun
12.60	1.00	Male	Yes	Sat	Dinner	2	0	0	1	0
13.42	1.68	Female	No	Thur	Lunch	2	1	0	0	0
10.07	1.83	Female	No	Thur	Lunch	1	1	0	0	0
23.95	2.55	Male	No	Sun	Dinner	2	0	0	0	1
16.43	2.30	Female	No	Thur	Lunch	2	1	0	0	0

One-hot encoding the day column resulted in four new columns, each representing a day of the week. For `day_Fri`, a 1 indicates that the day is Friday, while 0 indicates that it is not. Thursday is represented by $(1, 0, 0, 0)$, Friday by $(0, 1, 0, 0)$, and so on.

This method preserves the categorical nature of the data without introducing an ordinal relationship between the categories. This still doesn't capture the cyclic nature of the days of the week, but it does a better job at separating the days of the week into distinct categories without introducing an ordinal relationship between them.

Let's train a multivariate linear model on the *Tips* dataset using one-hot encoding. The process is similar to the univariate case, but we now have multiple input features. Our input data X is now a matrix of shape $n \times d$, where n is the number of samples and d is the number of features, rather than a vector like in the univariate case. Still, the matrix algebra is the same.

Our goal is to build a model that predicts the tip amount based on the available features. We've already encoded the `day` column. We'll have to encode the other features as well. As it turns out, the other features each have only two categories, so we can just encode them using label encoding, which will give us a binary column for each feature.

```
# Import iris dataset using seaborn
import seaborn as sns
data = sns.load_dataset('tips')

# One-Hot encode the ‘day‘ column
from sklearn.preprocessing import OneHotEncoder
```

```python
import pandas as pd
ohe = OneHotEncoder(sparse_output=False,categories=[['Thur', 'Fri', 'Sat', 'Sun']])
day_ohe = ohe.fit_transform(data[['day']])
day_ohe_df = pd.DataFrame(day_ohe, columns=[f'day_{day}' for day in ohe.categories[0]])
# Convert the one-hot encoded columns to integers
day_ohe_df = day_ohe_df.astype(int)
# Merge the one-hot encoded columns with the original dataframe
data = pd.concat([data, day_ohe_df], axis=1)

# LabelEncode the `sex` column
from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
data['sex_encoded'] = le.fit_transform(data['sex'])
# Print the transformation of the `sex` column
print('sex encoding:')
print(f"Male: {le.transform(['Male'])} Female: {le.transform(['Female'])}\n")

# LabelEncode the `smoker` column
data['smoker_encoded'] = le.fit_transform(data['smoker'])
# Print the transformation of the `smoker` column
print('smoker encoding:')
print(f"Yes: {le.transform(['Yes'])} No: {le.transform(['No'])}\n")

# LabelEncode the `time` column
data['time_encoded'] = le.fit_transform(data['time'])
# Print the transformation of the `time` column
print('time encoding:')
print(f"Lunch: {le.transform(['Lunch'])} Dinner: {le.transform(['Dinner'])} \n")

features = ['total_bill','sex_encoded','smoker_encoded','time_encoded','day_Thur','day_Fri','day_Sat','day_Sun','size']
target = 'tip'

print(data[features + [target]].sample(5).to_latex())
```

```
  sex encoding:
Male: 1 Female: 0

smoker encoding:
Yes: 1 No: 0

time encoding:
Lunch: 1 Dinner: 0
```

Our input matrix X is now a matrix of shape $n \times d$, where n is the number of samples and d is the number of features:

total_bill	sex_encoded	smoker_encoded	time_encoded	day_Thur	day_Fri	day_Sat	day_Sun	size	tip
12.02	1	0	0	0	0	1	0	2	1.97
29.85	0	0	0	0	0	0	1	5	5.14
19.44	1	1	1	1	0	0	0	2	3.00
35.83	0	0	0	0	0	1	0	3	4.67
27.20	1	0	1	1	0	0	0	4	4.00
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Table 1.2 Random sample of the encoded *Tips* dataset

$$X = \begin{pmatrix} 70 & 12.02 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 2 \\ 155 & 29.85 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 5 \\ 80 & 19.44 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 2 \\ 238 & 35.83 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 3 \\ 77 & 27.20 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 4 \\ \vdots & & \vdots & & & & & & \vdots & \end{pmatrix}$$

We need to create a model of the form

$$M(X) = X \cdot W^\intercal + b,$$

where:

- $X \in \mathbb{R}^{n \times d}$ is the input matrix,
- $W \in \mathbb{R}^{m \times d}$ is the weight matrix,
- $b \in \mathbb{R}^{n \times m}$ is the bias matrix, constructed by broadcasting the bias row vector $\vec{b} \in \mathbb{R}^m$ n times.

In our current application, we have $d = 9$ features, and since for each row we want to output a single number (the tip amount), we have $m = 1$. Therefore, our weight matrix W will be of shape $m \times d = 1 \times 9$, and thus $W^\intercal$ will be of shape $d \times m = 9 \times 1$. Our bias vector $\vec{b} \in \mathbb{R}^m = \mathbb{R}^1$ will be of shape 1×1, and our bias matrix b will be of shape $n \times 1$.

As before, we initialize a random weight W and bias b.

```
X = data[features].values
print(f'Shape of X: {X.shape}')

y = data[target].values
print(f'Shape of y: {y.shape}')

# Initialize random weight and bias for a linear model
import numpy as np
W = np.random.randn(X.shape[1])
print(f'Shape of W: {W.shape}')

b = np.random.randn(1)
print(f'Shape of b: {b.shape}')
```

```
 Shape of X: (244, 9)
Shape of y: (244,)
Shape of W: (9,)
Shape of b: (1,)
```

Now we need a loss function as before. We will use the mean squared error loss function, defined as:

$$\mathcal{L}(y_{\text{pred}}, y) = \frac{1}{n}\sum_{i=1}^{n}(y_{\text{pred}}^{(i)} - y^{(i)})^2 = \frac{1}{n}\|Y_{\text{pred}} - Y\|^2,$$

where $y_{\text{pred}}^{(i)} = X \cdot W^\intercal + b$ is the predicted tip amount for the i-th sample, and $y^{(i)}$ is the actual tip amount for the i-th sample.

Though our model M is taking in a matrix X of size $n \times d$, it will still work on a single row with d columns (see A.1.3 in the appendix). In this way, we can view our model as a map $M : \mathbb{R}^d \to \mathbb{R}^m$.

1.3.1.4 The Jacobian Matrix

Suppose $f : \mathbb{R}^d \to \mathbb{R}^m$ is a function such that each of its first-order derivatives exists over $\mathbb{R}^d$. The *Jacobian matrix* of f is a matrix of partial derivatives:

$$J_f = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_d} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_d} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \frac{\partial f_m}{\partial x_2} & \cdots & \frac{\partial f_m}{\partial x_d} \end{pmatrix}. \tag{1.6}$$

Given some point $p \in \mathbb{R}^d$, the Jacobian matrix evaluated at p is denoted $J_f(p)$. The Jacobian matrix is a generalization of the gradient to multivariate functions. The gradient is a special case of the Jacobian matrix when $m = 1$, where the Jacobian (a row vector) is the transpose of the gradient (a column vector).

Example 1.1 Suppose $f : \mathbb{R}^2 \to \mathbb{R}^3$ is defined as

$$f(x_1, x_2) = \begin{bmatrix} f_1(x_1, x_2) \\ f_2(x_1, x_2) \\ f_3(x_1, x_2) \end{bmatrix} = \begin{bmatrix} x_1^2 + x_2 \\ \sin(x_1 x_2) \\ e^{x_1 - x_2} \end{bmatrix}.$$

Then the Jacobian matrix J_f of f is

$$J_f = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} \\ \frac{\partial f_3}{\partial x_1} & \frac{\partial f_3}{\partial x_2} \end{pmatrix} = \begin{pmatrix} 2x_1 & 1 \\ x_2 \cos(x_1 x_2) & x_1 \cos(x_1 x_2) \\ e^{x_1 - x_2} & -e^{x_1 - x_2} \end{pmatrix}.$$

At the point $p = (1, 2) \in \mathbb{R}^2$, the Jacobian matrix J_f evaluated at p is

$$J_f(1, 2) = \begin{pmatrix} 2 & 1 \\ 2\cos(2) & \cos(2) \\ e^{-1} & -e^{-1} \end{pmatrix} \approx \begin{pmatrix} 2 & 1 \\ -0.832 & -0.416 \\ 0.367 & -0.367 \end{pmatrix}$$

Example 1.2 (Gradient of Mean Squared Error) Let's consider the loss function (MSE) from our linear model:

$$\mathcal{L}(y_{\text{pred}}, y) = \frac{1}{n} \sum_{i=1}^{n} (y_{\text{pred}}^{(i)} - y^{(i)})^2,$$

where $y_{\text{pred}}^{(i)} = X \cdot W^\intercal + b$ is the predicted tip amount for the i-th sample, and $y^{(i)}$ is the actual tip amount for the i-th sample.

The Jacobian of $\mathcal{L}$ with respect to the parameters W and b is:

$$J_{\mathcal{L}} = \begin{pmatrix} \frac{\partial \mathcal{L}}{\partial W} \\ \frac{\partial \mathcal{L}}{\partial b} \end{pmatrix} = \begin{pmatrix} \frac{2}{n} \sum_{i=1}^{n} (y_{\text{pred}}^{(i)} - y^{(i)}) X_{i,:} \\ \frac{2}{n} \sum_{i=1}^{n} (y_{\text{pred}}^{(i)} - y^{(i)}) \end{pmatrix}.$$

Notice that the partial derivatives are very similar to what we computed in the univariate case. The only difference is that we are now working with matrices and vectors instead of scalars.

Proof We need to derive two partial derivatives: $\frac{\partial \mathcal{L}}{\partial W}$ and $\frac{\partial \mathcal{L}}{\partial b}$.

- **Partial Derivative with Respect to W:** Using the chain rule:

$$\frac{\partial \mathcal{L}}{\partial W} = \frac{\partial \mathcal{L}}{\partial y_{\text{pred}}} \frac{\partial y_{\text{pred}}}{\partial W}.$$

Now,

$$\frac{\partial \mathcal{L}}{\partial y_{\text{pred}}} = \frac{1}{n} \sum_{i=1}^{n} 2(y_{\text{pred}}^{(i)} - y^{(i)}) = \frac{2}{n} \sum_{i=1}^{n} (y_{\text{pred}}^{(i)} - y^{(i)}),$$
$$\frac{\partial y_{\text{pred}}}{\partial W} = \frac{\partial}{\partial W} (X \cdot W^\intercal + b) = X.$$

- **Partial Derivative with Respect to b:** □

Appendix A
Python Code

A.1 Linear Regression

A.1.1 Code for 3D Gradient Descent Visualization

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# Assuming w_history, b_history, loss_history, x, and y_data are already defined
# Create a grid of w and b values for contour plotting
w_vals = np.linspace(min(w_history) - 1, max(w_history) + 1, 100)
b_vals = np.linspace(min(b_history) - 1, max(b_history) + 1, 100)
W, B = np.meshgrid(w_vals, b_vals)

# Compute the loss for each combination of w and b in the grid
Z = np.array([mse_loss(linear_model(x, w, b), y_data) for w, b in zip(np.ravel(W), np.ravel(B))])
Z = Z.reshape(W.shape)

# Create the figure and 3D axis
fig = plt.figure(figsize=(10, 8))
ax = fig.add_subplot(111, projection='3d')

# Plot the surface
surf = ax.plot_surface(W, B, Z, cmap='viridis', alpha=0.8)

# Plot the gradient descent path
ax.plot(w_history, b_history, loss_history, color='red', marker='o', markersize=4, label='Gradient Descent Path')

# Highlight the initial point
ax.scatter(w_history[0], b_history[0], loss_history[0], color='orange', s=50, label='Initial Point')

```

```
# Add labels and a legend
ax.set_title('Gradient Descent on Loss Surface')
ax.set_xlabel('Weight (w)')
ax.set_ylabel('Bias (b)')
ax.set_zlabel('Loss')
ax.legend()

plt.savefig('gradient-descent-3d.png')
# Show the plot
plt.show()
```

A.1.2 Code for Circle Classification Problem Visualizations

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# Generate circular data
np.random.seed(42)
n_samples = 500
radius = 1.0

# Generate random points in 2D space
X = np.random.uniform(-1.5, 1.5, (n_samples, 2))

# Assign class based on distance from origin
y = np.array([1 if np.linalg.norm(point) < radius else 0 for
    point in X])

# 2D Visualization
fig, ax = plt.subplots(figsize=(6, 6))
for point, label in zip(X, y):
    color = 'blue' if label == 1 else 'red'
    ax.scatter(point[0], point[1], color=color, s=10, alpha=0.7)
circle = plt.Circle((0, 0), radius, color='black', fill=False,
    linestyle='--')
ax.add_artist(circle)
ax.axhline(0, color='black', linewidth=0.5)
ax.axvline(0, color='black', linewidth=0.5)
ax.set_title("Circular Classification in 2D Space")
ax.set_xlabel("x1")
ax.set_ylabel("x2")
ax.set_aspect('equal', adjustable='datalim')
plt.grid()
plt.show()

# Map data to a higher-dimensional space: z = x1^2 + x2^2
z = np.sum(X**2, axis=1).reshape(-1, 1)
X_3D = np.hstack((X, z))

# 3D Visualization
fig = plt.figure(figsize=(6, 6))
ax = fig.add_subplot(111, projection='3d')
```

```
for point, label in zip(X_3D, y):
    color = 'blue' if label == 1 else 'red'
    ax.scatter(point[0], point[1], point[2], color=color, s=10,
    alpha=0.7)
ax.set_title("Circular Classification in 3D Space (Linear
    Separation)")
ax.set_xlabel("x1")
ax.set_ylabel("x2")
ax.set_zlabel("z = x1^2 + x2^2")

# Add a separating plane
xx, yy = np.linspace(-1.5, 1.5, 10), np.linspace(-1.5, 1.5, 10)
XX, YY = np.meshgrid(xx, yy)
ZZ = radius**2 * np.ones_like(XX)
ax.plot_surface(XX, YY, ZZ, alpha=0.5, color='green')
plt.show()
```

A.1.3 Why Training with Batches Works

During training, the input x is reshaped into a column vector $\vec{v} \in \mathbb{N} \times \not{\mathbb{F}}$, where N is the number of samples. Though our final model will process a single number x during inference. The reason this works lies in how matrix operations and broadcasting work. Let's look at a simple example to gain understanding:

1. **Training with a column vector:**
   Let's say we have three univariate data samples:

x	y
3	-1
5	1
7	0

   Our goal is to determine the relationship between x and y using a linear model $y = x \cdot W + b$. Let's choose 2 as the hidden dimension, so that $W, b \in \mathbb{R}^{1\times 2}$. We start by reshaping the input x into a column vector $\vec{v}$:

$$\vec{v} = \begin{bmatrix} 3 \\ 5 \\ 7 \end{bmatrix} \in \mathbb{R}^{3\times 1}$$

   We initialize with random weight and bias:

$$W_1 = \begin{bmatrix} 2 & -1 \end{bmatrix} \in \mathbb{R}^{1\times 2}, \quad b_1 = \begin{bmatrix} 1 & 0 \end{bmatrix} \in \mathbb{R}^{1\times 2}.$$

   Then:

$$Z_1 = \vec{v} \cdot W_1 + b_1 = \begin{bmatrix} 3 \\ 5 \\ 7 \end{bmatrix} \begin{bmatrix} 2 & -1 \end{bmatrix} + \begin{bmatrix} 1 & 0 \end{bmatrix} = \begin{bmatrix} 7 & -3 \\ 11 & -5 \\ 15 & -7 \end{bmatrix}.$$

2. **Inference with a single number:**

Now that we have a model (an untrained model, but still a model), we can use it to predict the output for a new input $x = 3$. We reshape x into a column vector $\vec{v} = \begin{bmatrix} 3 \end{bmatrix} \in \mathbb{R}^{1\times 1}$ and compute the output:

$$Z_1 = \vec{v} \cdot W_1 + b_1 = \begin{bmatrix} 3 \end{bmatrix} \begin{bmatrix} 2 & -1 \end{bmatrix} + \begin{bmatrix} 1 & 0 \end{bmatrix} = \begin{bmatrix} 7 & -3 \end{bmatrix}.$$

A.1.4 Direct Sum vs. Direct Product of Rings

Definition A.1 The **direct product** of two rings R and S, written $R \times S$, is the Cartesian product of their elements with componentwise addition and multiplication:

$$(r_1, s_1) + (r_2, s_2) = (r_1 + r_2, s_1 + s_2), \quad (r_1, s_1) \cdot (r_2, s_2) = (r_1 \cdot r_2, s_1 \cdot s_2).$$

This results in a new ring $R \times S$, where R and S are naturally embedded as subrings:

$$R \hookrightarrow R \times S, \quad r \mapsto (r, 0), \quad S \hookrightarrow R \times S, \quad s \mapsto (0, s).$$

For finite sets, the direct product is the same as the direct sum. For infinite sets, the direct product contains **all possible tuples**, even those with infinitely many nonzero entries, which is not allowed in the direct sum.

Definition A.2 The **direct sum** of two rings R and S, written $R \bigoplus S$, is also the cartesian product of their elements, but with an additional restriction: in the context of abelian groups, only *finitely many components are allowed to be nonzero* in the infinite case.

In the context of rings, $R \bigoplus S$ is typically defined the same as $R \times S$ for finite cases since the restriction is automatically satisfied.

In machine-learning, we are mostly working with a finite number of categorical variables, so the direct sum and direct product coincide.

Glossary

These definitions need to be redefined to make sense in the context of the book. Deciphering the jargon is a key part of understanding machine learning. Here are some common terms you might encounter:

Batch A batch is a set of training examples used in one iteration of model training. The batch size is the number of examples in a batch.

Broadcast Broadcasting a vector involves duplicating it along a specified dimension. This is a common operation in neural networks, where a vector is broadcast to match the dimensions of a matrix.

Hidden Dimension The hidden dimension is the number of neurons in a hidden layer of a neural network. Mathematically, it is a dimension used to construct a weight matrix and bias vector.

Hidden Layer A hidden layer is a layer in a neural network that is neither an input nor an output layer. It is used to transform the input into a form that is more suitable for the output layer. Mathematically, it is a layer of neurons that applies a non-linear transformation to the input.

Neuron A neuron is a single unit in a neural network that takes input, applies a transformation, and produces an output. Mathematically, it is a function that takes a weighted sum of inputs and applies an activation function. For example, a sigmoid neuron applies the sigmoid function to the weighted sum of inputs.

Index