

Samuel J. Cavazos

# Algebraic Geometry & Machine Learning

in Python for parsel tongues

December 2, 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

## Part I Regression Models

<b>1</b>	<b>Linear Regression</b> .....	3
1.1	Linear Models .....	3
1.2	Univariate Linear Models .....	4
1.2.1	Gradient Descent for Univariate Linear Models .....	6
1.2.2	Univariate Linear Models with Hidden Layers .....	14
1.3	Multivariate Linear Models .....	21
1.3.1	Encoding Categorical Data .....	22
<b>A</b>	<b>Python Code</b> .....	27
A.1	Linear Regression .....	27
A.1.1	Code for 3D Gradient Descent Visualization .....	27
A.1.2	Code for Circle Classification Problem Visualizations .....	28
A.1.3	Why Training with Batches Works .....	29
A.1.4	Direct Sum vs. Direct Product of Rings .....	30
	<b>Glossary</b> .....	31
	<b>Index</b> .....	33



## Acronyms

List of abbreviations and symbols used in the book.

$\mathbb{C}$	The field of complex numbers
$\mathbb{Q}$	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.

A more general form of the linear model is:

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

where:

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

Here, the model  $M(x)$  outputs an  $m$ -dimensional vector prediction. This flexibility allows linear regression to handle scenarios where the model predicts multiple outputs simultaneously, making it applicable to a wide range of machine learning tasks. We begin by studying the univariate case.

## 1.2 Univariate Linear Models

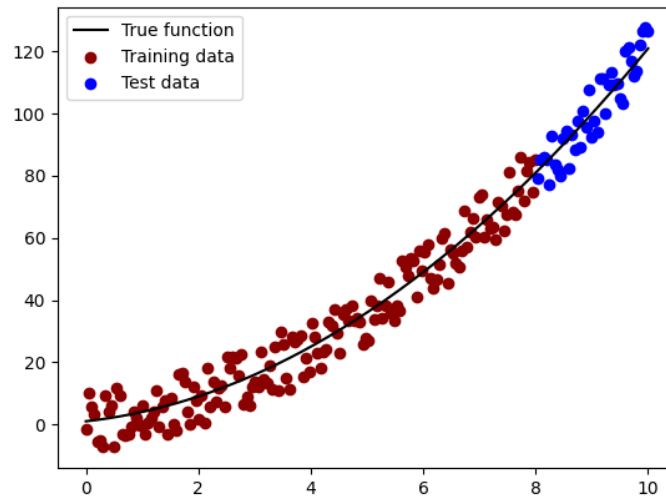
Let's construct some data to work with that follows a somewhat linear trend and build a machine-learning model from scratch. We'll take the function  $f(x) = x^2 + 2 \cdot x + 1$  over a random sample of points in  $[0, 10]$  and add some uniform noise. Next, we'll separate the synthetic data into training 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.

```

1  #!pip install matplotlib
2  #!pip install numpy
3  #!pip install sklearn
4
5  def f(x):
6      return x**2 + 2*x + 1
7
8  # Plot using matplotlib
9  import matplotlib.pyplot as plt
10 import numpy as np
11 import random
12
13 # Define the true function
14 def f(x):
15     return x**2 + 2*x + 1
16
17 # Generate data
18 np.random.seed(42)
19 x = np.linspace(0, 10, 200)
20 y_true = f(x)
21 y_data = y_true + np.random.uniform(-10, 10, size=x.shape)
22
23 # split into training and test datasets (80% training, 20% test)
24 from sklearn.model_selection import train_test_split
25 x_train, x_test, y_train, y_test = train_test_split(x, y_data,
26                                                    test_size=0.2)
27 print(f'Train size: {len(x_train)}')
28 print(f'Test size: {len(x_test)}')
29
30 # Plot the data and the true function, coloring the training and
31 # test data differently
32 plt.plot(x, y_true, color='black', label='True function')
33 plt.scatter(x_train, y_train, color='darkred', label='Training
34 data')
35 plt.scatter(x_test, y_test, color='blue', label='Test data')
36 plt.legend()
37 plt.savefig('fig1.png')
38 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  $M : [0, 10] \rightarrow \mathbb{R}$  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:

```
1 # Linear model
2 def linear_model(x, w, b):
3     return w * x + 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  $b$ .

```
1 import random
2
3 # Initialize parameters
4 w = random.uniform(-1, 1) # Random initial weight
```

```

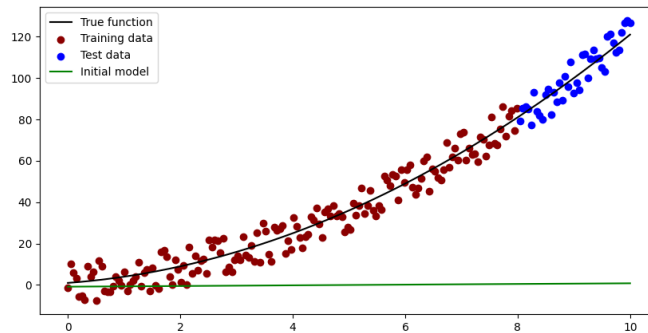
5 b = random.uniform(-1, 1) # Random initial bias
6
7 print(f'Initial weight: {w}')
8 print(f'Initial bias: {b}')

```

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:

$$\mathcal{L} = MSE = \frac{1}{N} \sum_{i=1}^N (y_{i,\text{pred}} - y_{i,\text{true}})^2.$$

```

1 # Mean Squared Error Loss
2 def mse_loss(y_pred, y_true):
3     return np.mean((y_pred - y_true)**2)
4
5 # Compute predictions for the training data
6 y_pred_train = [linear_model(p, w, b) for p in x_train]
7

```

```

8 print(f'50th sample target: {y_train[50]}')
9 print(f'50th prediction: {y_pred_train[50]}')
10 print(f'Loss at 50th sample: {mse_loss(y_pred_train[50], y_train
    [50])}')
11
12 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. Recall from vector calculus that the **gradient** of a differentiable function  $f$  is a vector field  $\nabla f$  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 concerning 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.

If  $x_i$  is the  $i$ -th data point, let  $y_{i,\text{pred}} = W \cdot x_i + b$  be the predicted value for the  $i$ -th data point while  $y_{i,\text{true}}$  denotes the true value. Mathematically, this gradient is computed as:

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial W} &= \frac{\partial}{\partial W} \left( \frac{1}{N} \sum_{i=1}^N (y_{i,\text{pred}} - y_{i,\text{true}})^2 \right) \\
&= \frac{1}{N} \sum_{i=1}^N \frac{\partial}{\partial W} (y_{i,\text{pred}} - y_{i,\text{true}})^2 \quad (\text{Additive property of derivatives}) \\
&= \frac{1}{N} \sum_{i=1}^N \frac{\partial}{\partial W} ((W \cdot x_i + b) - y_{i,\text{true}})^2 \\
&= \frac{1}{N} \sum_{i=1}^N 2 \cdot ((W \cdot x_i + b) - y_{i,\text{true}}) \cdot (x_i) \quad (\text{Chain rule}) \\
&= \frac{2}{N} \sum_{i=1}^N (y_{i,\text{pred}} - y_{i,\text{true}}) \cdot x_i.
\end{aligned}$$

Thus, we find that

$$\frac{\partial \mathcal{L}}{\partial W} = \frac{2}{N} \sum_{i=1}^N (y_{i,\text{pred}} - y_{i,\text{true}}) \cdot x_i. \quad (1.1)$$

## 2. Gradient with Respect to Bias $b$ :

Similarly, the partial derivative  $\frac{\partial \mathcal{L}}{\partial b}$  measures how the loss changes concerning the bias  $b$ . Adjusting  $b$  in the direction opposite to this gradient will also help in minimizing the loss.

This gradient is computed as:

$$\frac{\partial \mathcal{L}}{\partial b} = \frac{2}{N} \sum_{i=1}^N (y_{i,\text{pred}} - y_{i,\text{true}}). \quad (1.2)$$

Proof of this equation is left as an exercise to the reader.

With that, we can compute the gradients in Python:

```

1 # Compute gradients
2 def compute_gradients(x, y_true, w, b):
3     y_pred = linear_model(x, w, b)
4     error = y_pred - y_true
5     dw = 2 * np.mean(error * x)
6     db = 2 * np.mean(error)
7     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.

```

1 import numpy as np
2 import matplotlib.pyplot as plt

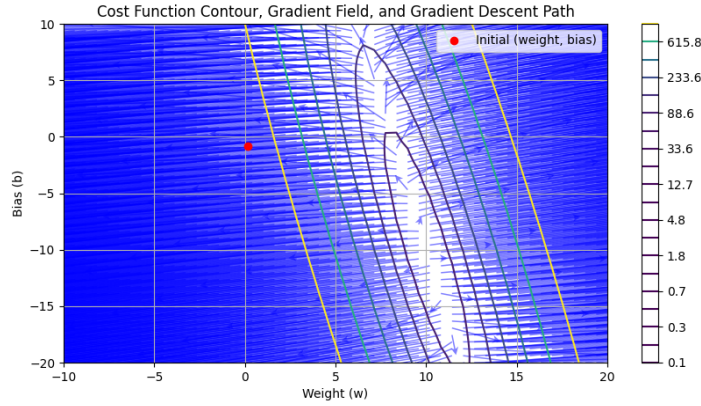
```

```

3
4 # Gradient Descent parameters
5 alpha = 0.001 # Learning rate
6 epochs = 1000 # Number of iterations
7
8 # Create a grid of w and b values for contour and quiver plotting
9 w_vals = np.linspace(-10, 20, 50)
10 b_vals = np.linspace(-20, 10, 50)
11 W, B = np.meshgrid(w_vals, b_vals)
12
13 # Compute the loss for each combination of w and b in the grid
14 Z = np.array([mse_loss(linear_model(x_train, w, b), y_train) for
15               w, b in zip(np.ravel(W), np.ravel(B))])
16 Z = Z.reshape(W.shape)
17
18 # Compute the gradient field
19 dW = np.zeros(W.shape)
20 dB = np.zeros(B.shape)
21 for i in range(W.shape[0]):
22     for j in range(W.shape[1]):
23         dw, db = compute_gradients(x_train, y_train, W[i, j], B[i, j])
24         dW[i, j] = dw
25         dB[i, j] = db
26
27 # Plot the cost function contour, gradient field, and gradient
28 # descent path
29 plt.figure(figsize=(10, 5))
30
31 # Contour plot of the loss function
32 cp = plt.contour(W, B, Z, levels=np.logspace(-1, 3, 20), cmap='
33 viridis')
34 plt.colorbar(cp)
35 plt.xlabel('Weight (w)')
36 plt.ylabel('Bias (b)')
37 plt.title('Cost Function Contour, Gradient Field, and Gradient
38 Descent Path')
39
40 # Quiver plot of the gradient field
41 plt.quiver(W, B, dW, dB, angles='xy', scale_units='xy', scale=2,
42           color='blue', alpha=0.5, headwidth=6, headlength=6)
43
44 # plot initial weight, bias
45 plt.plot(w, b, 'ro', label='Initial (weight, bias)')
46 plt.legend()
47 plt.grid(True)
48
49 plt.savefig('gradient-field-1.png')
50
51 plt.show()

```

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



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

optimization algorithms like gradient descent and is referred 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,  $\alpha$  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  $\alpha$  leads to slow convergence, while a large  $\alpha$  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**.

```

1 # Store parameters for plotting
2 w_history = [w]
3 b_history = [b]
4 loss_history = [mse_loss(linear_model(x_train, w, b), y_train)]
5
6 # Gradient Descent loop
7 for epoch in range(epochs):
8     dw, db = compute_gradients(x_train, y_train, w, b)
9     w = w - alpha * dw # Update the weight
10    b = b - alpha * db # Update the bias
11
12    w_history.append(w) # Add to weight tracker
13    b_history.append(b) # Add to bias tracker
14    loss_history.append(mse_loss(linear_model(x_train, w, b),
15                                y_train)) # Add overall loss to loss tracker
16
17 # Convert history lists to numpy arrays for easier slicing
18 w_history = np.array(w_history)
19 b_history = np.array(b_history)
20
21 # Compute the loss for each combination of w and b in the grid
22 Z = np.array([mse_loss(linear_model(x_train, w, b), y_train) for
23               w, b in zip(np.ravel(W), np.ravel(B))])
24 Z = Z.reshape(W.shape)
25
26 # Compute the gradient field
27 dW = np.zeros(W.shape)
28 dB = np.zeros(B.shape)
29 for i in range(W.shape[0]):
30     for j in range(W.shape[1]):
31         dw, db = compute_gradients(x_train, y_train, W[i, j], B[i, j])
32         dW[i, j] = dw
33         dB[i, j] = db
34
35 # Print initial (weight, bias)
36 print(f'Initial (weight, bias): ({w_history[0]}, {b_history[0]})')
37
38 # Print final (weight, bias)
39 print(f'Final (weight, bias): ({w_history[-1]}, {b_history[-1]})')
40
41 # Plot the cost function contour, gradient field, and gradient
42 # descent path
43 plt.figure(figsize=(10, 5))
44
45 # Contour plot of the loss function

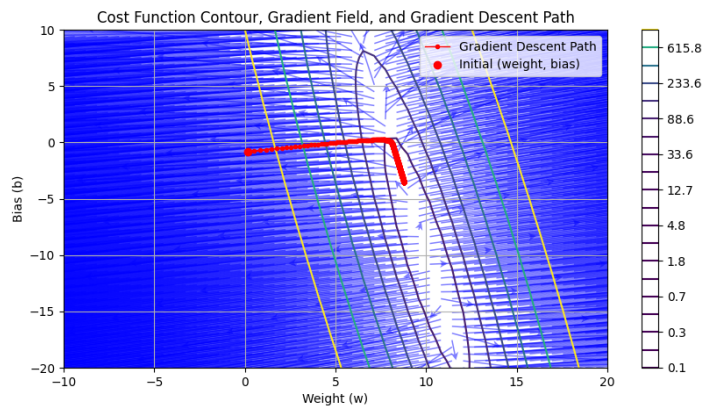
```

```

42 cp = plt.contour(W, B, Z, levels=np.logspace(-1, 3, 20), cmap='
    viridis')
43 plt.colorbar(cp)
44 plt.xlabel('Weight (w)')
45 plt.ylabel('Bias (b)')
46 plt.title('Cost Function Contour, Gradient Field, and Gradient
    Descent Path')
47
48 # Quiver plot of the gradient field
49 plt.quiver(W, B, dW, dB, angles='xy', scale_units='xy', scale=1,
    color='blue', alpha=0.5)
50
51 # Plot the gradient descent path
52 plt.plot(w_history, b_history, 'ro-', markersize=3, linewidth=1,
    label='Gradient Descent Path')
53 # Plot the initial weight, bias
54 plt.plot(w_history[0], b_history[0], 'ro', label='Initial (weight
    , bias)')
55
56 # Add arrows to indicate direction of descent
57 for i in range(1, len(w_history)):
58     plt.arrow(w_history[i-1], b_history[i-1],
59               w_history[i] - w_history[i-1],
60               b_history[i] - b_history[i-1],
61               head_width=0.05, head_length=0.1, fc='red', ec='
        red')
62
63 plt.legend()
64 plt.grid(True)
65 plt.savefig('gradient-field-2.png')
66 plt.show()

```

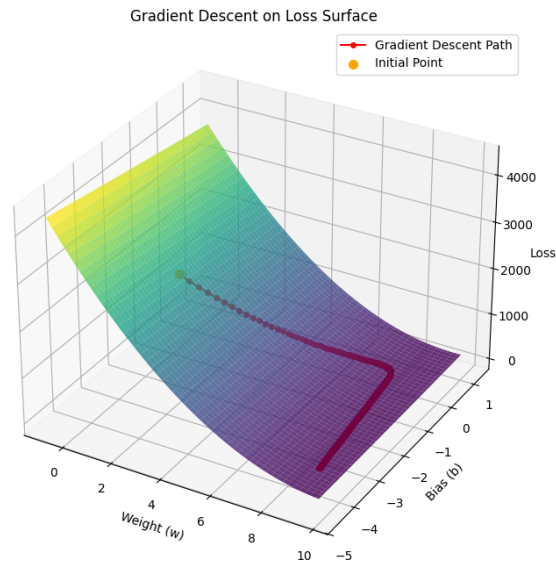
Initial (weight, bias): (0.5136609336515561, 0.39026605372156786)  
 Final (weight, bias): (10.472640107432522, -5.487933673141372)



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

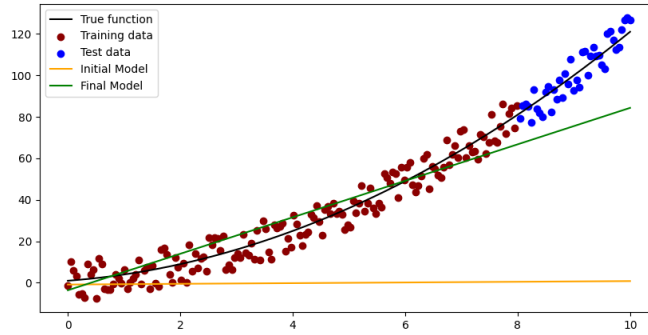


Since our weight  $W$  and bias  $b$  together form a point  $(W, b) \in \mathbb{R}^2$ , the loss function  $\mathcal{L}$  forms a 3-dimensional surface. The visualization below shows the path taken during gradient descent on the surface of the loss function  $\mathcal{L}$ . The initial point  $(W, b)$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

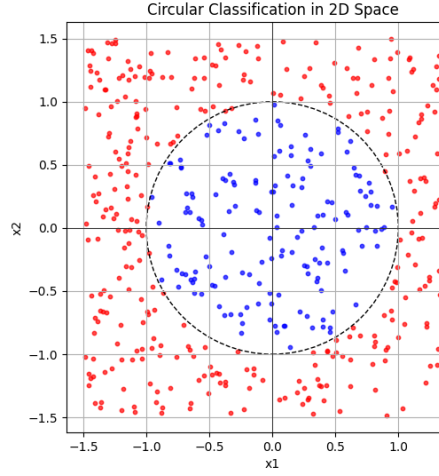


**Fig. 1.6** Initial and final linear models compared to the true line of best fit.

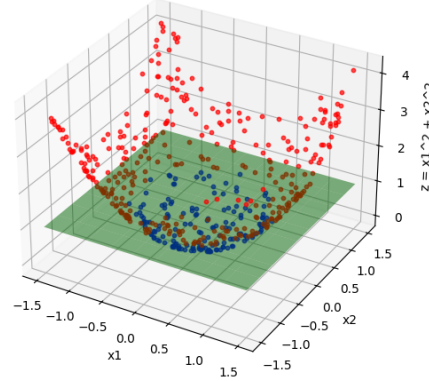
### 1.2.2 Univariate Linear Models with Hidden Layers

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 patterns 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.



Circular Classification in 3D Space (Linear Separation)

**Fig. 1.7** Circle classification problem in 2D.**Fig. 1.8** Circle classification problem in 3D.

Returning back to our linear model, let's change the model's architecture to include a hidden layer. To summarize the new model architecture:

- The *input layer* reshapes  $x$  into a column vector  $\mathbf{v} \in \mathbb{R}^{N \times 1}$ .
- The *hidden layer* maps  $\mathbf{v}$  from  $\mathbb{R}^{N \times 1}$  to  $Z_1 \in \mathbb{R}^{N \times d}$  using a linear transformation and activation function.
- The *output layer* maps the hidden representation in  $\mathbb{R}^{N \times d}$  to the final prediction  $y_{\text{pred}} \in \mathbb{R}^{N \times 1}$  using a linear transformation.

### 1. Input Layer:

- Takes a single real number  $x \in \mathbb{R}$  as input.
- During training, the inputs are reshaped into a column vector

$$\mathbf{v} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{bmatrix} \in \mathbb{R}^{N \times 1},$$

where  $N$  is the number of samples. We call  $\mathbf{v}$  a *batch*. Though we train with batches, inference is done on a single row at a time.

To see why training with batches still allows us to inference on a single row, see A.1.3 in the Appendix.

### 2. Hidden Layer:

The hidden layer maps the input  $\mathbf{v}$  from  $\mathbb{R}^{N \times 1}$  to  $Z_1 \in \mathbb{R}^{N \times d}$ , where  $d$  is the hidden dimension. We first perform a linear transformation with the input vector:

$$Z_1 = \mathbf{v} \cdot W_1 + b_1,$$

where

- $W_1 \in \mathbb{R}^{1 \times d}$  is the weight matrix,
- $b_1 \in \mathbb{R}^{1 \times d}$  is the bias vector.
- $Z_1 \in \mathbb{R}^{N \times d}$  is the resulting representation of our vector.

Once the linear transformation is complete, we apply a non-linear activation function to the result. This function introduces non-linearity into the model, allowing it to learn complex patterns in the data:

$$A_1 = \sigma(Z_1), \quad A_1 \in \mathbb{R}^{N \times d}.$$

We will discuss activation functions in more detail soon.

### 3. Output Layer:

The hidden layer's output  $A_1 \in \mathbb{R}^{N \times d}$  is then passed through the output layer, which maps it to the final prediction  $y_{\text{pred}} \in \mathbb{R}^{N \times 1}$  using a linear transformation:

$$Z_2 = A_1 \cdot W_2 + b_2,$$

where

- $W_2 \in \mathbb{R}^{d \times 1}$  is the weight matrix,
- $b_2 \in \mathbb{R}^{1 \times 1}$  is the bias vector,
- $Z_2 \in \mathbb{R}^{N \times 1}$  is the intermediate prediction.

In some applications,  $Z_2$  is passed through a final activation function to produce the final prediction  $y_{\text{pred}}$ , though for simple univariate linear regression, this step is often omitted, so

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

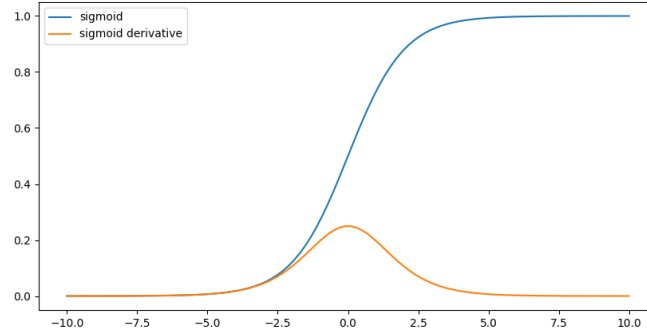
Layers are connected by an **activation function**. Activation functions should satisfy the following criteria:

- *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 facilitate 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.

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 import random
4
5 # Reshape data for neural network
6 x = x.reshape(-1, 1)
7 y_data = y_data.reshape(-1, 1)
8
9 # Initialize parameters
10 input_dim = x.shape[1] # Number of input features
11 hidden_dim = 2         # Number of neurons in the hidden layer
12 output_dim = y_data.shape[1] # Number of output neurons
13
14 # Weights and biases
15 np.random.seed(42)
16 W1 = np.random.randn(input_dim, hidden_dim) * 0.01
17 b1 = np.zeros((1, hidden_dim))
18 W2 = np.random.randn(hidden_dim, output_dim) * 0.01
19 b2 = np.zeros((1, output_dim))
20
21 # Fetch initial model predictions
22 Z1 = np.dot(x, W1) + b1
23 A1 = sigmoid(Z1)

```

```

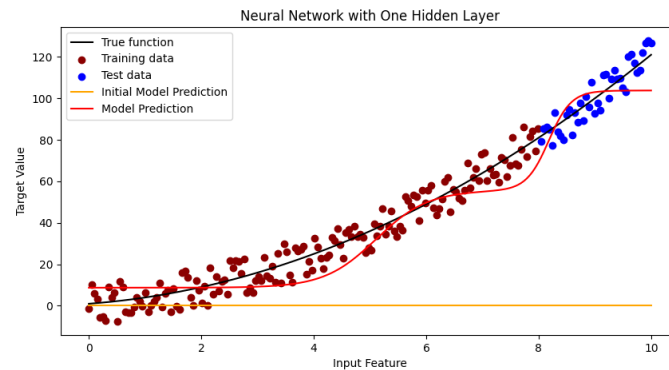
24 Z2 = np.dot(A1, W2) + b2
25 y_pred_initial = Z2
26
27 # Learning rate
28 alpha = 0.01
29
30 # Training loop
31 epochs = 10000
32 m = x.shape[0] # Number of training examples
33 loss_history = []
34
35 for epoch in range(epochs):
36     # Forward propagation
37     Z1 = np.dot(x, W1) + b1
38     A1 = sigmoid(Z1)
39     Z2 = np.dot(A1, W2) + b2
40     y_pred = Z2 # Linear activation for output layer
41
42     # Compute loss (Mean Squared Error)
43     loss = (1 / (2 * m)) * np.sum((y_pred - y_data) ** 2)
44     loss_history.append(loss)
45
46     # Backward propagation
47     dZ2 = y_pred - y_data
48     dW2 = (1 / m) * np.dot(A1.T, dZ2)
49     db2 = (1 / m) * np.sum(dZ2, axis=0, keepdims=True)
50     dA1 = np.dot(dZ2, W2.T)
51     dZ1 = dA1 * sigmoid_derivative(Z1)
52     dW1 = (1 / m) * np.dot(x.T, dZ1)
53     db1 = (1 / m) * np.sum(dZ1, axis=0, keepdims=True)
54
55     # Update parameters
56     W1 = W1 - alpha * dW1
57     b1 = b1 - alpha * db1
58     W2 = W2 - alpha * dW2
59     b2 = b2 - alpha * db2
60
61     # Print loss every 1000 epochs
62     if epoch % 1000 == 0:
63         print(f'Epoch {epoch}, Loss: {loss}')
64
65 # Predictions
66 Z1 = np.dot(x, W1) + b1
67 A1 = sigmoid(Z1)
68 Z2 = np.dot(A1, W2) + b2
69 y_pred = Z2
70
71 # Plotting
72 plt.figure(figsize=(10, 5))
73 plt.plot(x, y_true, color='black', label='True function')
74 plt.scatter(x_train, y_train, color='darkred', label='Training
    data')
75 plt.scatter(x_test, y_test, color='blue', label='Test data')

```

```

76 plt.plot(x, y_pred_initial, label='Initial Model Prediction',
77          color='orange')
78 plt.plot(x, y_pred, label='Model Prediction', color='red')
79 plt.xlabel('Input Feature')
80 plt.ylabel('Target Value')
81 plt.title('Neural Network with One Hidden Layer')
82 plt.legend()
83 plt.savefig('neural-network1.png')
84 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  $d = 10$ .

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 import random
4
5 # Reshape data for neural network
6 x = x.reshape(-1, 1)
7 y_data = y_data.reshape(-1, 1)
8
9 # Initialize parameters
10 input_dim = x.shape[1] # Number of input features
11 hidden_dim = 10        # Number of neurons in the hidden layer
12 output_dim = y_data.shape[1] # Number of output neurons
13
14 # Weights and biases
15 np.random.seed(42)
16 W1 = np.random.randn(input_dim, hidden_dim) * 0.01
17 b1 = np.zeros((1, hidden_dim))
18 W2 = np.random.randn(hidden_dim, output_dim) * 0.01

```

```

19 b2 = np.zeros((1, output_dim))
20
21 # Fetch initial model predictions
22 Z1 = np.dot(x, W1) + b1
23 A1 = sigmoid(Z1)
24 Z2 = np.dot(A1, W2) + b2
25 y_pred_initial = Z2
26
27 # Learning rate
28 alpha = 0.01
29
30 # Training loop
31 epochs = 10000
32 m = x.shape[0] # Number of training examples
33 loss_history = []
34
35 for epoch in range(epochs):
36     # Forward propagation
37     Z1 = np.dot(x, W1) + b1
38     A1 = sigmoid(Z1)
39     Z2 = np.dot(A1, W2) + b2
40     y_pred = Z2 # Linear activation for output layer
41
42     # Compute loss (Mean Squared Error)
43     loss = (1 / (2 * m)) * np.sum((y_pred - y_data) ** 2)
44     loss_history.append(loss)
45
46     # Backward propagation
47     dZ2 = y_pred - y_data
48     dW2 = (1 / m) * np.dot(A1.T, dZ2)
49     db2 = (1 / m) * np.sum(dZ2, axis=0, keepdims=True)
50     dA1 = np.dot(dZ2, W2.T)
51     dZ1 = dA1 * sigmoid_derivative(Z1)
52     dW1 = (1 / m) * np.dot(x.T, dZ1)
53     db1 = (1 / m) * np.sum(dZ1, axis=0, keepdims=True)
54
55     # Update parameters
56     W1 -= alpha * dW1
57     b1 -= alpha * db1
58     W2 -= alpha * dW2
59     b2 -= alpha * db2
60
61     # Print loss every 1000 epochs
62     if epoch % 1000 == 0:
63         print(f'Epoch {epoch}, Loss: {loss}')
64
65 # Predictions
66 Z1 = np.dot(x, W1) + b1
67 A1 = sigmoid(Z1)
68 Z2 = np.dot(A1, W2) + b2
69 y_pred = Z2
70
71 # Plotting

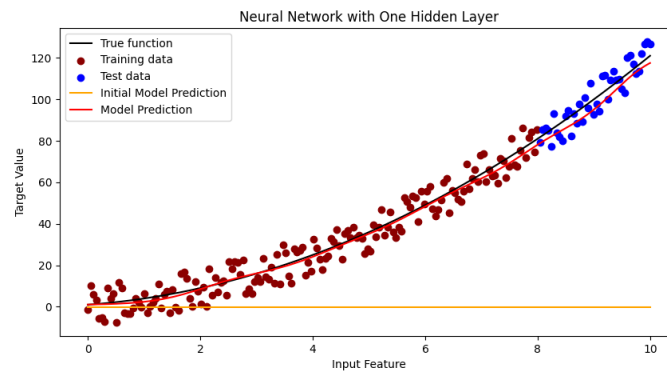
```



```

72 plt.scatter(x, y_data, label='Noisy Data', color='blue', alpha
    =0.5)
73 plt.plot(x, y_pred_initial, label='Initial Model Prediction',
    color='orange')
74 plt.plot(x, y_true, label='True Function', color='green')
75 plt.plot(x, y_pred, label='Model Prediction', color='red')
76 plt.xlabel('Input Feature')
77 plt.ylabel('Target Value')
78 plt.title('Neural Network with One Hidden Layer')
79 plt.legend()
80 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 univariate case:

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

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.

```
1 # %!pip install seaborn
2
3 # Import iris dataset using seaborn
4 import seaborn as sns
5 data = sns.load_dataset('tips')
6
7 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 to 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  $f : X \rightarrow Y$  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 essence, 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`:

```

1 # Label Encode the 'day' column
2 from sklearn.preprocessing import LabelEncoder
3 le = LabelEncoder()
4 data['day_encoded'] = le.fit_transform(data['day'])
5
6 # Save the encoder to a pickle file
7 import pickle
8 with open('label_encoder.pkl', 'wb') as f:
9     pickle.dump(le, f)
10
11 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. This 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.

```

1 # Ordinal encode the 'day' column
2 from sklearn.preprocessing import OrdinalEncoder
3 oe = OrdinalEncoder(categories=[['Thur', 'Fri', 'Sat', 'Sun']])
4 data['day_OrdinalEncoded'] = oe.fit_transform(data[['day']])
5
6 # Save the encoder to a pickle file
7 with open('ordinal_encoder.pkl', 'wb') as f:
8     pickle.dump(oe, f)
9
10 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.

```

1 # One-Hot encode the 'day' column
2 from sklearn.preprocessing import OneHotEncoder
3
4 ohe = OneHotEncoder(sparse_output=False, categories=[['Thur', 'Fri', 'Sat', 'Sun']])
5 day_ohe = ohe.fit_transform(data[['day']])
6 day_ohe_df = pd.DataFrame(day_ohe, columns=[f'day_{day}' for day
7     in ohe.categories[0]])
8 # Merge the one-hot encoded columns with the original dataframe
9 data = pd.concat([data, day_ohe_df], axis=1)
10 print(data.sample(5).to_latex(index=False, float_format="%0.2f"))

```

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.

#### 1.3.1.4 Encoding using Rings

Let's now approach encoding categorical data from a more abstract perspective. Let's begin by defining a *ring*.

**Definition 1.2** A **ring** is a set  $R$  equipped with two binary operations  $+$  and  $\cdot$  such that:

1.  $(R, +)$  is an abelian group, i.e.
  - $(a + b) + c = a + (b + c)$  for all  $a, b, c \in R$  (associativity of addition),
  - $a + b = b + a$  for all  $a, b \in R$  (commutativity of addition),
  - There exists an element  $0 \in R$  such that  $a + 0 = a$  for all  $a \in R$  (additive identity),
  - For every  $a \in R$ , there exists an element  $-a \in R$  such that  $a + (-a) = 0$  (additive inverse).
2.  $(R, \cdot)$  is a monoid, i.e.
  - $(a \cdot b) \cdot c = a \cdot (b \cdot c)$  for all  $a, b, c \in R$  (associativity of multiplication),
  - There exists an element  $1 \in R$  such that  $a \cdot 1 = a$  for all  $a \in R$  (multiplicative identity).
3. Multiplication distributes over addition:
  - $a \cdot (b + c) = a \cdot b + a \cdot c$  for all  $a, b, c \in R$  (left distributivity).
  - $(a + b) \cdot c = a \cdot c + b \cdot c$  for all  $a, b, c \in R$  (right distributivity).

*Example 1.1* The integers  $\mathbb{Z}$  form a ring under addition and multiplication. The integers modulo  $n$ , denoted  $\mathbb{Z}/n\mathbb{Z}$ , also form a ring under addition and multiplication modulo  $n$ .

*Example 1.2* The abelian group  $\mathbb{Z}/\mathbb{Z}_2$ , the integers modulo 2, forms a ring under addition and multiplication modulo 2. Two elements,  $[x], [y] \in \mathbb{Z}/\mathbb{Z}_2$ , are equivalent in  $\mathbb{Z}/\mathbb{Z}_2$  if  $x - y$  is divisible by 2. For example,  $[3] = [1]$  in  $\mathbb{Z}/\mathbb{Z}_2$  since  $3 - 1 = 2$  is divisible by 2.

When context is clear, we drop the brackets around the equivalence classes and write  $x$  instead of  $[x]$ , which leads to the infamous:

$$1 + 1 = 0.$$

Since  $1 + 1 = 2$  over  $\mathbb{Z}$  and  $2 - 0 = 2$  is divisible by 2, the equivalence classes  $[2] = [0]$  are the same so that  $[1] + [1] = [0]$  in  $\mathbb{Z}/\mathbb{Z}_2$ .

The previous example is a ring with only two elements. Recall the one-hot encoding of the days of the week: Thursday was encoded as  $(1, 0, 0, 0)$ , Friday as  $(0, 1, 0, 0)$ , and so on. Each slot in the vector is either a 0 or a 1, depending on what is being encoded. This encoding embeds our categorical data into the direct product of rings:

$$\prod_{i=1}^c \mathbb{Z}/\mathbb{Z}_2,$$

where  $c$  is the number of categories.

Some might refer to this as a direct sum, but this terminology should be avoided in the context of rings. While  $\bigoplus_{i=1}^c \mathbb{Z}/\mathbb{Z}_2$  is valid in the category of abelian groups, in ring theory, the direct sum and direct product differ. Though for finite rings, the direct sum and direct product coincide.

The *direct sum* and *direct product* are two closely related constructions that differ primarily in the context of infinite indexing sets. However, their behavior can diverge even for finite cases when we are careful about their definitions in different categories, such as rings or abelian groups (see A.1.4 in the Appendix for more information).

## Appendix A

### Python Code

#### A.1 Linear Regression

##### A.1.1 Code for 3D Gradient Descent Visualization

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 from mpl_toolkits.mplot3d import Axes3D
4
5 # Assuming w_history, b_history, loss_history, x, and y_data are
   already defined
6 # Create a grid of w and b values for contour plotting
7 w_vals = np.linspace(min(w_history) - 1, max(w_history) + 1, 100)
8 b_vals = np.linspace(min(b_history) - 1, max(b_history) + 1, 100)
9 W, B = np.meshgrid(w_vals, b_vals)
10
11 # Compute the loss for each combination of w and b in the grid
12 Z = np.array([mse_loss(linear_model(x, w, b), y_data) for w, b in
   zip(np.ravel(W), np.ravel(B))])
13 Z = Z.reshape(W.shape)
14
15 # Create the figure and 3D axis
16 fig = plt.figure(figsize=(10, 8))
17 ax = fig.add_subplot(111, projection='3d')
18
19 # Plot the surface
20 surf = ax.plot_surface(W, B, Z, cmap='viridis', alpha=0.8)
21
22 # Plot the gradient descent path
23 ax.plot(w_history, b_history, loss_history, color='red', marker='
   o', markersize=4, label='Gradient Descent Path')
24
25 # Highlight the initial point
26 ax.scatter(w_history[0], b_history[0], loss_history[0], color='
   orange', s=50, label='Initial Point')
27
```

```

28 # Add labels and a legend
29 ax.set_title('Gradient Descent on Loss Surface')
30 ax.set_xlabel('Weight (w)')
31 ax.set_ylabel('Bias (b)')
32 ax.set_zlabel('Loss')
33 ax.legend()
34
35 plt.savefig('gradient-descent-3d.png')
36 # Show the plot
37 plt.show()

```

### A.1.2 Code for Circle Classification Problem Visualizations

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 from mpl_toolkits.mplot3d import Axes3D
4
5 # Generate circular data
6 np.random.seed(42)
7 n_samples = 500
8 radius = 1.0
9
10 # Generate random points in 2D space
11 X = np.random.uniform(-1.5, 1.5, (n_samples, 2))
12
13 # Assign class based on distance from origin
14 y = np.array([1 if np.linalg.norm(point) < radius else 0 for
15               point in X])
16
17 # 2D Visualization
18 fig, ax = plt.subplots(figsize=(6, 6))
19 for point, label in zip(X, y):
20     color = 'blue' if label == 1 else 'red'
21     ax.scatter(point[0], point[1], color=color, s=10, alpha=0.7)
22 circle = plt.Circle((0, 0), radius, color='black', fill=False,
23                     linestyle='--')
24 ax.add_artist(circle)
25 ax.axhline(0, color='black', linewidth=0.5)
26 ax.axvline(0, color='black', linewidth=0.5)
27 ax.set_title("Circular Classification in 2D Space")
28 ax.set_xlabel("x1")
29 ax.set_ylabel("x2")
30 ax.set_aspect('equal', adjustable='datalim')
31 plt.grid()
32 plt.show()
33
34 # Map data to a higher-dimensional space:  $z = x_1^2 + x_2^2$ 
35 z = np.sum(X**2, axis=1).reshape(-1, 1)
36 X_3D = np.hstack((X, z))
37
38 # 3D Visualization
39 fig = plt.figure(figsize=(6, 6))
40 ax = fig.add_subplot(111, projection='3d')

```



```

39 for point, label in zip(X_3D, y):
40     color = 'blue' if label == 1 else 'red'
41     ax.scatter(point[0], point[1], point[2], color=color, s=10,
42               alpha=0.7)
43 ax.set_title("Circular Classification in 3D Space (Linear
44             Separation)")
45 ax.set_xlabel("x1")
46 ax.set_ylabel("x2")
47 ax.set_zlabel("z = x1^2 + x2^2")
48
49 # Add a separating plane
50 xx, yy = np.linspace(-1.5, 1.5, 10), np.linspace(-1.5, 1.5, 10)
51 XX, YY = np.meshgrid(xx, yy)
52 ZZ = radius**2 * np.ones_like(XX)
53 ax.plot_surface(XX, YY, ZZ, alpha=0.5, color='green')
54 plt.show()

```

### A.1.3 Why Training with Batches Works

During training, the input  $x$  is reshaped into a column vector  $\mathbf{v} \in \mathbb{N} \times \mathbb{R}$ , 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:

$$\begin{array}{c|c} x & y \\ \hline 3 & -1 \\ 5 & 1 \\ 7 & 0 \end{array}$$

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  $\mathbf{v}$ :

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

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:

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

#### 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  $\mathbf{v} = [3] \in \mathbb{R}^{1 \times 1}$  and compute the output:

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

#### 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  $R \oplus S$ , 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,  $R \oplus S$  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.

**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

## A

acronyms, list of   xiii

## B

backpropagation   10

## E

embedding   23

epoch   11

## G

gradient   7

gradient descent   6, 10

gradient field   8

## H

hidden layer   14

## L

label encoding   23

linear model, univariate   3

loss function   6

## O

one-hot encoding   24

ordinal encoding   23

## R

ring   25

## S

symbols, list of   xiii