Samuel J. Cavazos

Algebraic Geometry & Machine Learning

in Python for parseltongues

November 30, 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

1 Linear Regression 3
1.1 Linear Models 3
1.2 Univariate Linear Models 4
1.2.1 Gradient Descent for Univariate Linear Models 5
1.2.2 Univariate Linear Models with Hidden Layers 14

A Python Code 23
A.1 Linear Regression 23

Glossary 27

Index 29

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.

```
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)

# Plot from 0 to 10
plt.xlim(0, 10)
plt.plot(x, y_data, 'o')
plt.plot(x, y_true)

plt.show()
```

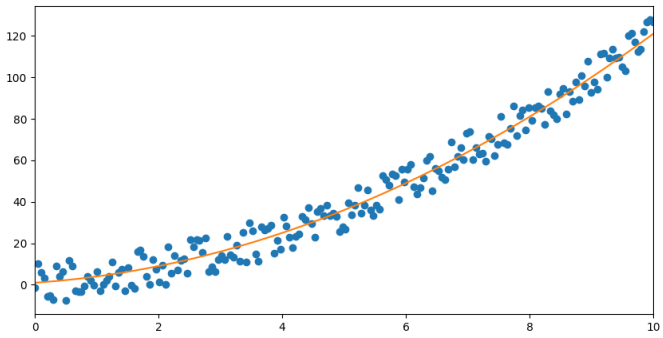

Fig. 1.1 Data generated from the function $f(x) = x^2 + 2 \cdot x + 1$ with added noise.

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 $M : [0, 10] \to \mathbb{R}$ 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 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.

```
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: -0.21299332372819757
Initial bias: -0.7532380921958663
```

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.

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:

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

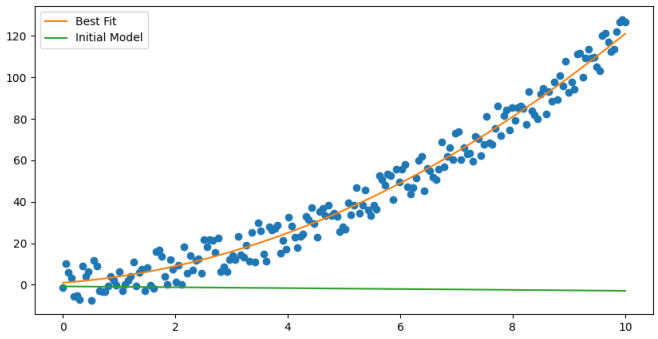

Fig. 1.2 Initial model prediction with random weight and bias.

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

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

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

```
50th sample target: 21.729790078081002
50th prediction: -1.2883971970405839
Loss at 50th sample: 529.8369454325693
Total Loss over all samples: 3485.837693509094
```

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 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:

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

   Thus, we find that

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

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:

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

   Proof of this equation is left as an exercise to 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, 100)
b_vals = np.linspace(-20, 10, 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)

# 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, y_data, 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=(12, 8))

# 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 Point of
    Initial Weight, Bias')

# 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 initial weight, bias
plt.plot(w, b, 'ro', label='Initial (weight, bias)')
plt.legend()
plt.grid(True)
plt.show()
```

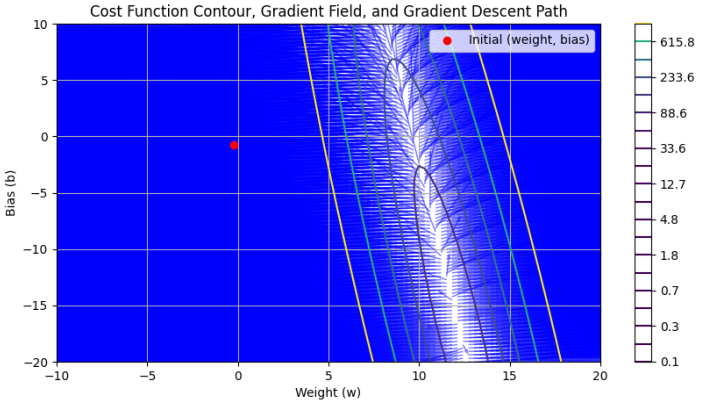

Fig. 1.3 Contour plot of the loss function, gradient field, and gradient descent path.

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**.

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

# Assuming x and y_data are your input features and target values
    respectively

# Define the linear model
def linear_model(x, w, b):
    return w * x + b

# Define the loss function (Mean Squared Error)
def mse_loss(y_pred, y_true):
    return np.mean((y_pred - y_true) ** 2)

# 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

# Gradient Descent parameters
alpha = 0.001  # Learning rate
epochs = 2000  # Number of iterations

# Store parameters for plotting
w_history = [w]
b_history = [b]
loss_history = [mse_loss(linear_model(x, w, b), y_data)]

```

```
# Gradient Descent loop
for epoch in range(epochs):
    dw, db = compute_gradients(x, y_data, 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, w, b), y_data))
    # 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)

# Create a grid of w and b values for contour and quiver plotting
w_vals = np.linspace(-10, 20, 100)
b_vals = np.linspace(-20, 10, 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)

# 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, y_data, 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=(12, 8))

# 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.show()
```

```
   Initial (weight, bias): (-0.21299332372819757, -0.7532380921958663)
Final (weight, bias): (11.160092718469242, -10.114682847750428)
```

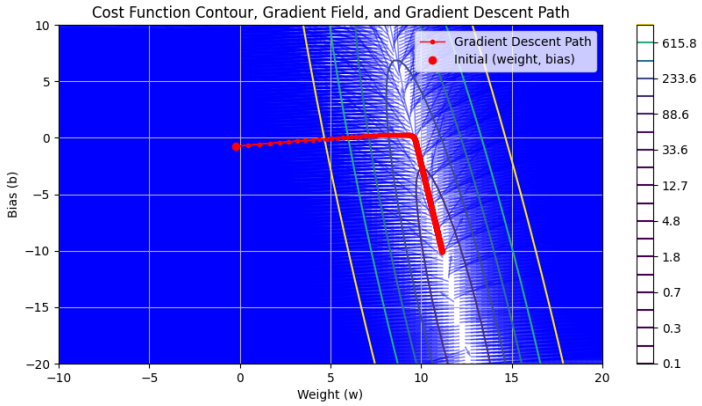

Fig. 1.4 Contour plot of the loss function, gradient field, and gradient descent path.

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.

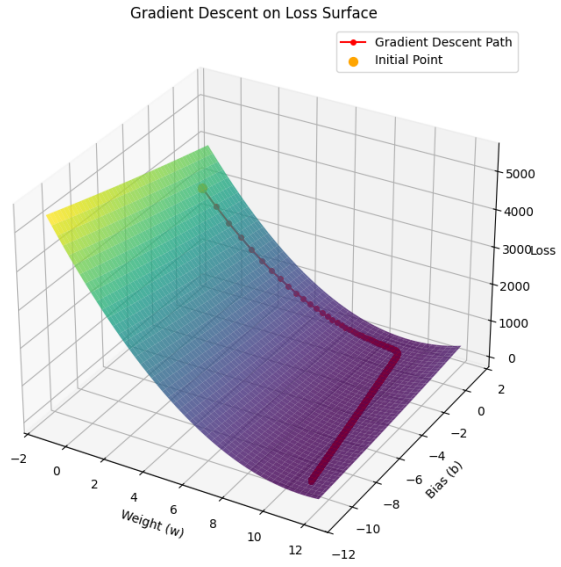

Fig. 1.5 Gradient descent path on the loss function surface. The code used to generate this visualization can be found in A.1.0.1 of the Appendix.

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

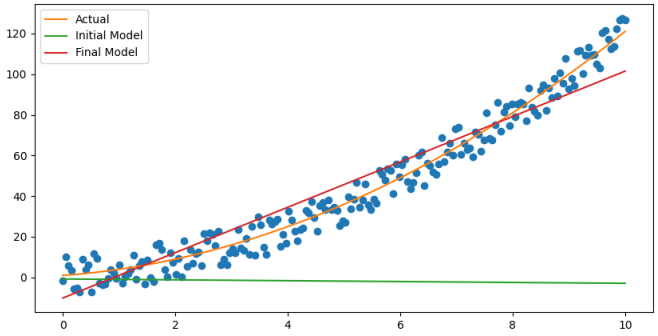

Fig. 1.6 Initial and final linear models compared to the true line of best fit.

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.1.0.2 in the Appendix for the code used to generate the visualizations below.

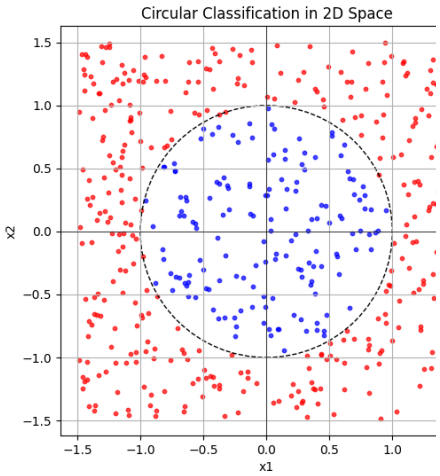

Fig. 1.7 Circle classification problem in 2D.

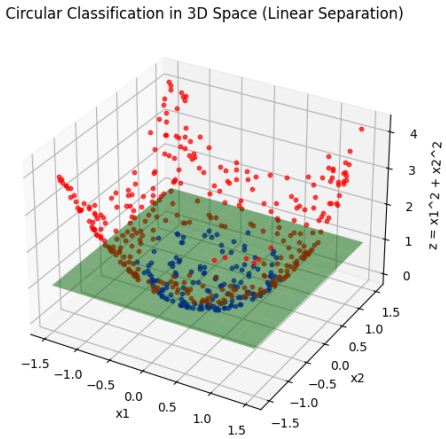

Fig. 1.8 Circle classification problem in 3D.

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.1.0.3 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_{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 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.

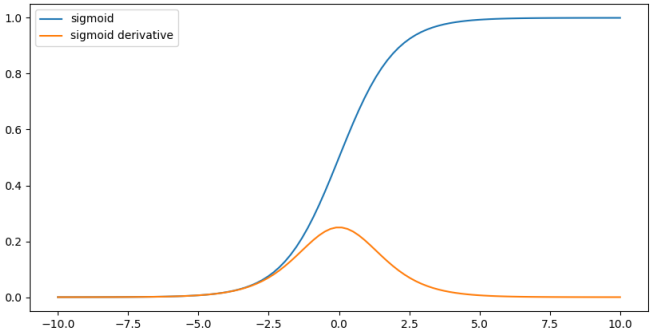

Fig. 1.9 The sigmoid activation function and its derivative.

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.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()
```

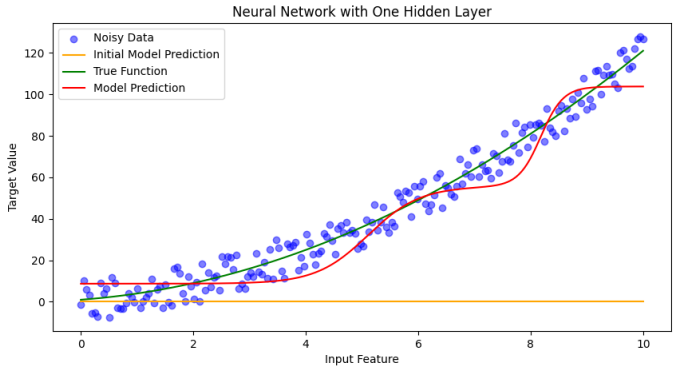

Fig. 1.10 Univariate linear model with a single hidden layer of dimension $d = 2$. The initial model's state is shown, as well as its final state after training.

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()
```

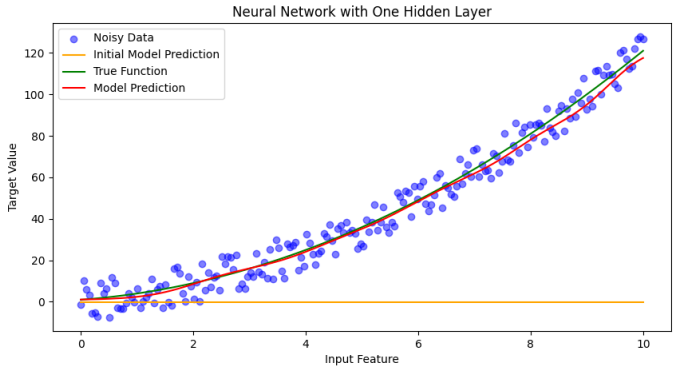

Fig. 1.11 Univariate linear model with a single hidden layer of dimension $d = 10$. The initial model's state is shown, as well as its final state after training.

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 $d = 2$.

Appendix A
Python Code

A.1 Linear Regression

A.1.0.1 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.1.0.2 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.1.0.3 Why Training with Batches Works

During training, the input x is reshaped into a column vector $\mathbf{v} \in \mathbb{N} \times \not{\mathbb{F}}$, 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 $\mathbf{v}$:

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

   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

$\mathbf{v} = \begin{bmatrix} 3 \end{bmatrix} \in \mathbb{R}^{1\times 1}$ and compute the output:

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

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 9

E

epoch 10

G

gradient 6
gradient descent 5, 9
gradient field 8

H

hidden layer 14

L

linear model, univariate 3
loss function 5

S

symbols, list of xiii