Contents

Week 1 Overview: Python & ML Fundamentals

Introduction

Week 1 establishes the foundation for your machine learning journey. You will master essential Python libraries, understand fundamental ML concepts, and complete your first end-to-end machine learning project.

Week Goals

- Master NumPy for numerical computing and array operations
- Learn Matplotlib for data visualization and result presentation
- Understand core ML concepts: supervised learning, train/val/test splits, overfitting
- Implement linear regression and logistic classification from scratch
- Use scikit-learn for practical machine learning workflows
- Complete Titanic survival prediction project

Weekly Structure

- **Day 1**: Python Crash Course - NumPy & Arrays
- **Day 2**: Python Crash Course - Matplotlib & Data Visualization
- **Day 3**: Introduction to Machine Learning Concepts
- **Day 4**: Classification and Logistic Regression
- **Day 5**: Scikit-learn Ecosystem and Mini-Project

Key Learning Outcomes

By the end of Week 1, you will be able to:

- Manipulate multi-dimensional arrays efficiently using NumPy
- Create professional visualizations to communicate insights

- Explain the difference between training, validation, and test sets
- Implement gradient descent and understand how models learn
- Evaluate classification models using appropriate metrics
- Build complete ML pipelines using scikit-learn
- Work independently on structured ML projects

Tips for Success

To get the most out of Week 1:

- Watch all video tutorials completely before starting exercises
- Type out code examples rather than copying and pasting - except for print statements, plotting statements etc I recommend that you should write out every line of code, unless you understand it completely.
- Complete daily reflections - they help consolidate learning
- Ask questions during check-ins if concepts are unclear
- Compare your solutions with peers to learn different approaches

Daily Schedule Format

Each day follows this structure:

Morning Session (4 hours)

- Optional: Daily check-in with peers on Teams (15 min)
- Video Learning (60-120 min)
- Reference Material (30 min)
- Hands-on Coding - Part 1 (remainder)

Afternoon Session (4 hours)

- Video Learning (30 min, if applicable)
- Hands-on Coding - Part 2 (3-3.5 hours)

Reflection & Consolidation (30 min)

- Review key concepts
- Write daily reflection
- List questions for check-in

Assessment

As long as you demonstrate a bona fide effort in completing exercises and projects, you will receive credit for the vac work. Focus on learning rather than perfection.

Communication

- **Microsoft Teams**: Use the group for questions, discussions, and peer support
- **Check-ins**: Monday, Wednesday, Friday (approximately 1 hour each)
- **Email**: For individual questions or concerns

Ready to begin? Start with Day 1: NumPy & Arrays

Week 1, Day 1: Python Crash Course - NumPy & Arrays

Daily Goals

- Set up Python environment (Anaconda/Colab)
- Master NumPy array operations
- Understand vectorization benefits

Morning Session (4 hours)

Optional: Daily Check-in with Peers on Teams (15 min)

Video Learning (90 min)

??? **Watch**: NumPy Tutorial for Beginners by freeCodeCamp (1 hour)

??? **Watch**: Python NumPy Tutorial for Beginners by Keith Galli - focus on 0:00-15:00 for basics and 30:00-45:00 for array operations (30 min)

Reference Material (30 min)

??? **Read**: NumPy Quickstart Tutorial - Sections on "The Basics" and "Array Creation"

??? **Bookmark**: NumPy Reference Documentation for afternoon use

Hands-on Coding - Part 1 (2 hours)

Setup (15 min) ??? Create a Google Colab notebook titled "Day1_NumPy_Practice"

??? Import NumPy: `import numpy as np`

??? Test installation: `print(np.__version__)`

Exercise 1: Array Creation (30 min) Create the following arrays and print their shape, dtype, and contents:

1. A 1D array with integers from 0 to 9
   - *Hint: Use `np.arange()`*
2. A 2D array (3x3) filled with zeros
   - *Hint: Use `np.zeros()`*
3. A 3x3 identity matrix
   - *Hint: Use `np.eye()`*
4. A 1D array with 10 evenly spaced values between 0 and 1
   - *Hint: Use `np.linspace()`*
5. A 2D array (5x5) with random values between 0 and 1
   - *Hint: Use `np.random.rand()`*
6. A 2D array (3x4) with sequential values from 0 to 11
   - *Hint: Use `np.arange()` then `.reshape()`*

Exercise 2: Array Indexing and Slicing (40 min) Given this array:

```
arr = np.arange(0, 100).reshape(10, 10)
```

Complete these tasks:

1. Extract the element at row 3, column 5
   - *Expected output: 35*
2. Extract the entire 5th row

    - *Expected output: array([50, 51, 52, 53, 54, 55, 56, 57, 58, 59])*
3. Extract the top-left 3x3 subarray
    - *Hint: Use slicing `arr[start:end, start:end]`*
4. Extract every other row
    - *Hint: Use step in slicing `arr[::2]`*
5. Extract the bottom-right 2x2 subarray
    - *Hint: Use negative indices*
6. Extract all elements greater than 50 (returns 1D array)
    - *Hint: Use boolean indexing `arr[arr > 50]`*

Exercise 3: Basic Array Operations (35 min) Create two arrays:

```
a = np.array([1, 2, 3, 4, 5])
b = np.array([10, 20, 30, 40, 50])
```

Perform and print results:

1. Element-wise addition (a + b)
    - *Expected output: [11, 22, 33, 44, 55]*
2. Element-wise multiplication (a * b)
    - *Expected output: [10, 40, 90, 160, 250]*
3. Square each element of array a
    - *Hint: Use a `** 2`*
4. Calculate the dot product of a and b
    - *Hint: Use `np.dot()` or `@` operator*
    - *Expected output: 550*
5. Find the sum, mean, and standard deviation of array a
    - *Hint: Use `np.sum()`, `np.mean()`, `np.std()`*
6. Find the maximum value in b and its index
    - *Hint: Use `np.max()` and `np.argmax()`*

Afternoon Session (4 hours)

Video Learning (30 min)

??? **Watch**: NumPy Broadcasting by Algorithmic Simplicity (15 min)

??? **Watch**: Vectorization in Python first half (15 min)

Hands-on Coding - Part 2 (3 hours)

Exercise 4: Statistics Without Loops (50 min) Implement these functions using only NumPy operations (no Python loops):

1. normalize_array(arr): Takes a 1D array and returns it normalized to have mean=0 and std=1

```python
def normalize_array(arr):
    # Your code here
    pass

# Test
arr = np.array([1, 2, 3, 4, 5])
result = normalize_array(arr)
print(f"Mean: {result.mean():.10f}, Std: {result.std():.10f}")
# Expected: Mean ??? 0, Std ??? 1
```

Hint: normalized = (arr - mean) / std

2. moving_average(arr, window_size): Calculate moving average with given window size

```python
def moving_average(arr, window_size):
    # Your code here
    pass

# Test
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
result = moving_average(arr, 3)
print(result)
# Expected output: [2., 3., 4., 5., 6., 7., 8., 9.]
```

Hint: Use `np.convolve()` with `mode='valid'`

3. find_outliers(arr, threshold): Return indices where values deviate from the mean by more than threshold * std

```python
def find_outliers(arr, threshold):
    # Your code here
    pass

# Test
arr = np.array([1, 2, 3, 4, 5, 100, 6, 7, 8])
outliers = find_outliers(arr, 2)
print(f"Outlier indices: {outliers}")
print(f"Outlier values: {arr[outliers]}")
# Expected: Index 5 (value 100) is an outlier
```

Hint: Calculate z-scores: `(arr - mean) / std`, then find where `abs(z_scores) > threshold`

Exercise 5: Broadcasting Practice (40 min) Solve these without loops:

1. Add a 1D array `[1, 2, 3]` to each row of a 4x3 matrix

```python
matrix = np.array([[1, 1, 1],
                   [2, 2, 2],
                   [3, 3, 3],
                   [4, 4, 4]])
to_add = np.array([1, 2, 3])
# Your code here
# Expected output:
# [[2, 3, 4],
#  [3, 4, 5],
#  [4, 5, 6],
#  [5, 6, 7]]
```

Hint: Broadcasting works automatically with `matrix + to_add`

2. Create a 10x10 multiplication table using broadcasting

```python
# Your code here
# Expected: Element (i,j) should be i*j
# Row 5 should be [0, 5, 10, 15, 20, 25, 30, 35, 40, 45]
```

Hint: Create two arrays with shapes (10, 1) and (1, 10), then multiply

3. Normalize each column of a matrix independently to range [0, 1]

```python
matrix = np.random.rand(5, 3) * 100
# Your code here: normalize each column so min=0, max=1
# Verify: Check that each column's min is 0 and max is 1
```

Hint: For each column: `(col - col.min()) / (col.max() - col.min())` Use broadcasting with `axis=0` in min/max functions

Mini-Challenge: Image Manipulation (90 min) You'll work with a grayscale image represented as a 2D NumPy array. First, create a test image:

```
# Create a simple test pattern
image = np.zeros((100, 100))
image[25:75, 25:75] = 1.0  # White square in center
image[40:60, 40:60] = 0.5  # Gray square in middle

# Visualize (you'll need this throughout)
import matplotlib.pyplot as plt
plt.imshow(image, cmap='gray')
plt.colorbar()
plt.show()
```

Implement these image transformations:

1. rotate_90(image): Rotate image 90 degrees clockwise

```
def rotate_90(image):
    # Your code here
    pass

result = rotate_90(image)
plt.imshow(result, cmap='gray')
plt.title('Rotated 90?? clockwise')
plt.show()
```

Hint: Use `np.rot90()` with appropriate k value

2. flip_horizontal(image): Flip image horizontally (mirror left-right)

```
def flip_horizontal(image):
    # Your code here
    pass

result = flip_horizontal(image)
plt.imshow(result, cmap='gray')
plt.title('Flipped Horizontally')
plt.show()
```

Hint: Use `np.fliplr()` or slicing with `::-1`

3. crop_center(image, size): Crop a square of given size from the center

```
def crop_center(image, size):
    # Your code here
    # Example: if image is 100x100 and size=50, extract the central 50x50 region
    pass

result = crop_center(image, 50)
print(f"Cropped shape: {result.shape}")  # Should be (50, 50)
plt.imshow(result, cmap='gray')
plt.title('Center Crop (50x50)')
plt.show()
```

Hint: Calculate center indices, then slice appropriately

4. adjust_brightness(image, factor): Multiply all pixels by factor and clip to [0, 1]

```python
def adjust_brightness(image, factor):
    # Your code here
    pass

# Test with darkening (factor=0.5) and brightening (factor=1.5)
dark = adjust_brightness(image, 0.5)
bright = adjust_brightness(image, 1.5)

fig, axes = plt.subplots(1, 3, figsize=(15, 5))
axes[0].imshow(image, cmap='gray')
axes[0].set_title('Original')
axes[1].imshow(dark, cmap='gray')
axes[1].set_title('Darkened (0.5x)')
axes[2].imshow(bright, cmap='gray')
axes[2].set_title('Brightened (1.5x)')
plt.show()
```

Hint: Use `np.clip(image * factor, 0, 1)`

5. apply_threshold(image, threshold): Convert to binary (0 or 1) based on threshold

```python
def apply_threshold(image, threshold):
    # Your code here
    # Pixels >= threshold become 1, others become 0
    pass

result = apply_threshold(image, 0.5)
plt.imshow(result, cmap='gray')
plt.title('Thresholded at 0.5')
plt.show()
```

Hint: Use boolean comparison and `.astype(float)`

Reflection & Consolidation (30 min)

??? Review all code you wrote today

??? Clean up your notebook with comments and markdown cells

??? Write your daily reflection in a separate document (choose 2-3 prompts below)

??? List any questions for the Monday check-in email

Daily Reflection Prompts (Choose 2-3):

- What was the most important concept you learned today?
- What challenged you the most? How did you approach it?
- What connections did you make between today's content and previous learning?
- What questions do you still have?
- How does today's learning relate to real-world applications?
- What would you do differently if you repeated today?

Next: Day 2 - Matplotlib & Data Visualization

Week 1, Day 2: Python Crash Course - Matplotlib & Data Visualization

Daily Goals

- Master basic plotting with Matplotlib
- Create various plot types (line, scatter, histogram, heatmap)
- Understand when to use different visualizations

Morning Session (4 hours)

Optional: Daily Check-in with Peers on Teams (15 min)

Video Learning (90 min)

??? **Watch**: Matplotlib Tutorial for Beginners by Corey Schafer (1 hour)

??? **Watch**: Matplotlib Crash Course by Tech With Tim - focus on 0:00-30:00 (30 min)

Reference Material (30 min)

??? **Read**: Matplotlib Quick Start Guide

??? **Bookmark**: Matplotlib Gallery for examples

Hands-on Coding - Part 1 (2 hours)

Setup (10 min) ??? Create a new Colab notebook titled "Day2_Matplotlib_Practice"

??? Import libraries:

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

Exercise 1: Basic Line Plots (40 min) **1. Simple line plot**: Plot y = x?? for x values from -10 to 10

```
x = np.linspace(-10, 10, 100)
# Your code here
```

Hint: Use `plt.plot()` then `plt.show()` Expected: A parabola centered at origin

2. Multiple lines: Plot sin(x), cos(x), and tan(x) for x from 0 to 2?? on the same graph

```
x = np.linspace(0, 2*np.pi, 100)
# Your code here
# Add labels, legend, title, and grid
```

Hint: Call `plt.plot()` multiple times before `plt.show()` Use `plt.legend()`, `plt.xlabel()`, `plt.ylabel()`, `plt.title()`, `plt.grid()`

3. Customized plot: Create a line plot with: - Custom colors (use 'red', 'blue', etc.) - Different line styles (solid, dashed, dotted) - Markers at data points - Custom line widths

```
x = np.linspace(0, 10, 20)
y1 = x
y2 = x**2
# Your code here
```

Hint: Use parameters like `color='red'`, `linestyle='--'`, `marker='o'`, `linewidth=2`

Exercise 2: Scatter Plots (35 min) **1. Basic scatter plot**: Create random data and visualize

```
np.random.seed(42)
x = np.random.randn(100)
y = 2*x + np.random.randn(100)*0.5
# Create scatter plot
# Add title and labels
```

Hint: Use `plt.scatter()` Expected: Points roughly following a line with some noise

2. Colored scatter plot: Create a scatter plot where point colors represent a third variable

```
x = np.random.rand(50)
y = np.random.rand(50)
colors = x + y  # Color based on sum of coordinates
sizes = (x * 100) ** 2  # Size based on x value
# Create scatter plot with colors and varying sizes
# Add a colorbar
```

Hint: Use `c=colors`, `s=sizes` parameters in `plt.scatter()`, then `plt.colorbar()`

Exercise 3: Subplots (35 min) Create a 2x2 grid of subplots showing different mathematical functions:

```
x = np.linspace(-5, 5, 100)

# Create 2x2 subplot layout
fig, axes = plt.subplots(2, 2, figsize=(10, 10))

# Top-left: y = x??
# Top-right: y = e^x
# Bottom-left: y = sin(x)
# Bottom-right: y = 1/x (avoid x=0)

# Add titles to each subplot
# Add a main title to the figure
```

Hint: Access subplots using `axes[row, col].plot()` Use `axes[row, col].set_title()` for individual titles Use `fig.suptitle()` for main title Expected: Four distinct plots in a grid layout

---

Afternoon Session (4 hours)

Video Learning (30 min)

??? **Watch**: Matplotlib Histograms and Bar Charts by Corey Schafer (15 min)

??? **Watch**: Seaborn Heatmaps first 15 minutes (15 min)

Hands-on Coding - Part 2 (3 hours)

Exercise 4: Histograms (40 min) **1. Single histogram**: Generate and visualize normal distribution

```
data = np.random.randn(1000)
# Create histogram with 30 bins
# Add labels and title
```

Hint: Use `plt.hist(data, bins=30)` Expected: Bell curve shape

2. Overlapping histograms: Compare two distributions

```
data1 = np.random.randn(1000)
data2 = np.random.randn(1000) + 2  # Shifted distribution
# Plot both histograms with transparency
# Add legend
```

Hint: Use `alpha=0.5` for transparency Use different colors for each histogram

3. Histogram analysis: Generate data and analyze with histogram

```
# Generate data from mixed distributions
data = np.concatenate([
    np.random.randn(500),
    np.random.randn(500) + 5
])
# Create histogram
# What pattern do you observe?
```

Expected: Bimodal distribution (two peaks)

Exercise 5: Heatmaps (45 min) **1. Correlation matrix**: Create and visualize a correlation matrix

```
# Generate correlated data
np.random.seed(42)
data = np.random.randn(100, 5)
# Calculate correlation matrix
corr_matrix = np.corrcoef(data.T)

# Create heatmap
# Add colorbar and labels
```

Hint: Use `plt.imshow()` with cmap='`coolwarm`' Use `plt.colorbar()` to show scale

2. 2D function visualization: Visualize z = sin(x) * cos(y)

```
x = np.linspace(-np.pi, np.pi, 100)
y = np.linspace(-np.pi, np.pi, 100)
X, Y = np.meshgrid(x, y)
Z = np.sin(X) * np.cos(Y)

# Create heatmap
# Add colorbar, labels, and title
```

Hint: Use `plt.imshow()` or `plt.contourf()` Expected: Symmetric pattern with positive and negative regions

Exercise 6: Bar Charts (35 min) **1. Simple bar chart**: Visualize category data

```
categories = ['Category A', 'Category B', 'Category C', 'Category D']
values = [23, 45, 56, 78]
# Create bar chart
# Rotate x-labels if needed
```

Hint: Use `plt.bar()` Use `plt.xticks(rotation=45)` if labels overlap

2. Grouped bar chart: Compare multiple series

```
categories = ['Q1', 'Q2', 'Q3', 'Q4']
product_A = [50, 60, 70, 80]
product_B = [45, 55, 65, 85]
```

```
# Create grouped bar chart showing both products
# Add legend
```

Hint: Use x positions and offset bars with width parameter Example: `plt.bar(x - width/2, product_A)` *and* `plt.bar(x + width/2, product_B)`

Mini-Challenge: Data Visualization Dashboard (60 min) Create a comprehensive visualization dashboard for a synthetic dataset:

```
# Generate synthetic student performance data
np.random.seed(42)
n_students = 200

study_hours = np.random.uniform(0, 10, n_students)
attendance = np.random.uniform(50, 100, n_students)
exam_scores = (study_hours * 5 + attendance * 0.3 +
               np.random.randn(n_students) * 5)
exam_scores = np.clip(exam_scores, 0, 100)
```

Create a figure with 6 subplots (2 rows ?? 3 columns) showing:

1. **Scatter plot**: Study hours vs exam scores (with trendline) *Hint: Use* `np.polyfit()` *and* `np.poly1d()` *for trendline*
2. **Scatter plot**: Attendance vs exam scores (colored by study hours) *Hint: Use* `c=study_hours` *and add colorbar*
3. **Histogram**: Distribution of exam scores
   - Mark mean and median with vertical lines *Hint: Use* `plt.axvline()` *for vertical lines*
4. **Histogram**: Distribution of study hours
5. **Box plot**: Exam scores grouped by study hour ranges (0-3, 3-6, 6-10 hours)

```
# Categorize students
low_study = exam_scores[study_hours < 3]
med_study = exam_scores[(study_hours >= 3) & (study_hours < 6)]
high_study = exam_scores[study_hours >= 6]
```

Hint: Use `plt.boxplot([low_study, med_study, high_study])`

6. **Heatmap**: 2D histogram (study hours vs attendance), colored by average exam score *Hint: Use* `plt.hist2d()` *or bin the data manually*

Requirements: - All subplots should have appropriate titles, labels, and legends where needed - Use a consistent color scheme - Add a main title to the entire figure - Make sure the figure is large enough to see all details clearly

---

Reflection & Consolidation (30 min)

??? Review all visualizations you created today

??? Experiment with different color schemes and styles

??? Write your daily reflection (choose 2-3 prompts below)

??? Note any questions for the check-in email

Daily Reflection Prompts (Choose 2-3):

- What was the most important concept you learned today?
- What challenged you the most? How did you approach it?
- What connections did you make between yesterday’s NumPy work and today’s visualization?
- What questions do you still have?
- How could these visualization techniques be useful in understanding data?
- What would you do differently if you repeated today?

Next: Day 3 - Introduction to Machine Learning

Week 1, Day 3: Introduction to Machine Learning Concepts

Daily Goals

- Understand supervised vs unsupervised learning
- Learn about train/validation/test splits
- Understand overfitting and underfitting
- Grasp regression concepts and gradient descent
- Implement linear regression from scratch

Morning Session (4 hours)

Optional: Daily Check-in with Peers on Teams (15 min)

Video Learning (2 hours)

??? **Watch**: Machine Learning Basics by StatQuest (10 min)

??? **Watch**: Supervised Learning by StatQuest (9 min)

??? **Watch**: Train, Validation, and Test Sets by StatQuest (10 min)

??? **Watch**: Overfitting and Underfitting by StatQuest (20 min)

??? **Watch**: Linear Regression by StatQuest (27 min)

??? **Watch**: Gradient Descent by StatQuest (20 min)

Reference Material (30 min)

??? **Read**: Dive into Deep Learning - Chapter 3.1-3.2

Hands-on Coding - Part 1 (1.5 hours)

Exercise 1: Data Splits (30 min) Create train/validation/test splits:

```
from sklearn.model_selection import train_test_split
from sklearn.datasets import make_regression
import numpy as np

# Generate data
X, y = make_regression(n_samples=100, n_features=1, noise=10, random_state=42)

# Split into train (60%), val (20%), test (20%)
X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.4, random_state=42)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5, random_state=4

print(f"Train: {len(X_train)}, Val: {len(X_val)}, Test: {len(X_test)}")
```

Expected: 60, 20, 20 samples

Exercise 2: Linear Regression from Scratch (60 min) Implement gradient descent:

```
def train_linear_regression(X, y, learning_rate=0.01, epochs=100):
    w, b = 0.0, 0.0
    losses = []

    for epoch in range(epochs):
        # Forward pass
```

```python
        y_pred = w * X + b

        # Compute loss (MSE)
        loss = np.mean((y - y_pred) ** 2)
        losses.append(loss)

        # Compute gradients
        dw = -2 * np.mean(X * (y - y_pred))
        db = -2 * np.mean(y - y_pred)

        # Update parameters
        w -= learning_rate * dw
        b -= learning_rate * db

        if (epoch + 1) % 10 == 0:
            print(f"Epoch {epoch+1}: Loss = {loss:.4f}")

    return w, b, losses

w, b, losses = train_linear_regression(X_train, y_train, learning_rate=0.5, epochs=100)
```

Expected: Loss should decrease over epochs

Afternoon Session (4 hours)

Hands-on Coding - Part 2 (3.5 hours)

Exercise 3: Polynomial Regression & Overfitting (60 min)

```python
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error

degrees = [1, 2, 5, 10, 15]
train_errors, val_errors = [], []

for degree in degrees:
    poly = PolynomialFeatures(degree=degree)
    X_train_poly = poly.fit_transform(X_train)
    X_val_poly = poly.transform(X_val)

    model = LinearRegression()
    model.fit(X_train_poly, y_train)

    train_errors.append(mean_squared_error(y_train, model.predict(X_train_poly)))
    val_errors.append(mean_squared_error(y_val, model.predict(X_val_poly)))

# Plot training vs validation error
plt.plot(degrees, train_errors, label='Train')
plt.plot(degrees, val_errors, label='Validation')
plt.xlabel('Polynomial Degree')
plt.ylabel('MSE')
plt.legend()
plt.show()
```

Expected: Validation error increases for high degrees (overfitting)

Exercise 4: Regularization (50 min) Compare Ridge and Lasso:

```
from sklearn.linear_model import Ridge, Lasso

# Using degree 10 polynomial features
poly = PolynomialFeatures(degree=10)
X_train_poly = poly.fit_transform(X_train)
X_val_poly = poly.transform(X_val)

# Compare models
models = {
    'Linear': LinearRegression(),
    'Ridge': Ridge(alpha=1.0),
    'Lasso': Lasso(alpha=0.1)
}

for name, model in models.items():
    model.fit(X_train_poly, y_train)
    val_pred = model.predict(X_val_poly)
    val_error = mean_squared_error(y_val, val_pred)
    print(f"{name}: Val MSE = {val_error:.4f}")
```

Expected: Ridge/Lasso may have better validation performance

Mini-Challenge: California Housing (90 min) Complete end-to-end regression project:

```
from sklearn.datasets import fetch_california_housing
from sklearn.preprocessing import StandardScaler

# Load data
housing = fetch_california_housing()
X, y = housing.data, housing.target

# Split data (60-20-20)
X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.4, random_state=42)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5, random_state=4

# Scale features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_val_scaled = scaler.transform(X_val)
X_test_scaled = scaler.transform(X_test)

# Train and compare models
from sklearn.metrics import r2_score

for alpha in [0.1, 1.0, 10.0]:
    ridge = Ridge(alpha=alpha)
    ridge.fit(X_train_scaled, y_train)

    val_pred = ridge.predict(X_val_scaled)
    val_mse = mean_squared_error(y_val, val_pred)
    val_r2 = r2_score(y_val, val_pred)

    print(f"Ridge (??={alpha}): MSE={val_mse:.4f}, R??={val_r2:.4f}")

# Final evaluation on test set with best model
best_model = Ridge(alpha=1.0)
```

```
best_model.fit(X_train_scaled, y_train)
test_pred = best_model.predict(X_test_scaled)
print(f"\nTest R??: {r2_score(y_test, test_pred):.4f}")

# Visualize predictions vs actual
plt.scatter(y_test, test_pred, alpha=0.5)
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], 'r--')
plt.xlabel('True Values')
plt.ylabel('Predictions')
plt.title('Test Set Predictions')
plt.show()
```

Reflection & Consolidation (30 min)

??? Review gradient descent and loss functions ??? Understand train/val/test splits ??? Write daily reflection (choose 2-3 prompts below) ??? List questions for check-in

Daily Reflection Prompts (Choose 2-3):

- What was the most important concept you learned today?
- How does gradient descent help models learn?
- What is the purpose of validation sets?
- What did you observe about overfitting?
- How does regularization prevent overfitting?
- What questions do you still have?

Next: Day 4 - Classification and Logistic Regression

Week 1, Day 4: Classification and Logistic Regression

Daily Goals

- Understand classification problems and metrics
- Learn logistic regression and sigmoid function
- Explore multi-class classification
- Practice with real classification datasets
- Master evaluation metrics (accuracy, precision, recall, F1, ROC-AUC)

Morning Session (4 hours)

Optional: Daily Check-in with Peers on Teams (15 min)

Video Learning (2 hours)

??? **Watch**: Logistic Regression by StatQuest (15 min)

??? **Watch**: Logistic Regression Details Pt 1 by StatQuest (9 min)

??? **Watch**: Logistic Regression Details Pt 2 by StatQuest (11 min)

??? **Watch**: ROC and AUC by StatQuest (16 min)

??? **Watch**: Confusion Matrix by StatQuest (8 min)

??? **Watch**: Sensitivity and Specificity by StatQuest (12 min)

??? **Watch**: Cross Validation by StatQuest (6 min)

Reference Material (30 min)

??? **Read**: Dive into Deep Learning - Chapter 4.1 - Softmax Regression

??? **Bookmark**: Scikit-learn Classification Metrics

Hands-on Coding - Part 1 (1.5 hours)

Exercise 1: Binary Classification Basics (45 min)

```python
from sklearn.datasets import make_classification
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, confusion_matrix
import numpy as np
import matplotlib.pyplot as plt

# Generate binary classification data
X, y = make_classification(n_samples=200, n_features=2, n_redundant=0,
                          n_informative=2, n_clusters_per_class=1, random_state=42)

# Split data
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Train logistic regression
model = LogisticRegression()
model.fit(X_train, y_train)

# Make predictions
```

```python
y_pred = model.predict(X_test)

# Evaluate
accuracy = accuracy_score(y_test, y_pred)
cm = confusion_matrix(y_test, y_pred)

print(f"Accuracy: {accuracy:.4f}")
print(f"Confusion Matrix:\n{cm}")

# Visualize decision boundary
from matplotlib.colors import ListedColormap

def plot_decision_boundary(X, y, model):
    h = 0.02
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                         np.arange(y_min, y_max, h))

    Z = model.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)

    plt.contourf(xx, yy, Z, alpha=0.4, cmap=ListedColormap(['#FFAAAA', '#AAAAFF']))
    plt.scatter(X[:, 0], X[:, 1], c=y, cmap=ListedColormap(['#FF0000', '#0000FF']),
                edgecolors='black')
    plt.xlabel('Feature 1')
    plt.ylabel('Feature 2')
    plt.title('Decision Boundary')
    plt.show()

plot_decision_boundary(X_train, y_train, model)
```

Expected: Clear separation between classes with decision boundary

Exercise 2: Probability Predictions (45 min)

```python
# Get probability predictions
y_proba = model.predict_proba(X_test)

print("First 10 predictions:")
print("True | Pred | P(class=0) | P(class=1)")
print("-" * 45)
for i in range(10):
    print(f"{y_test[i]:4d} | {y_pred[i]:4d} | {y_proba[i,0]:10.4f} | {y_proba[i,1]:10.4f}")

# Plot probability histogram
plt.figure(figsize=(12, 4))

plt.subplot(1, 2, 1)
plt.hist(y_proba[y_test == 0, 1], bins=20, alpha=0.7, label='Class 0', edgecolor='black')
plt.hist(y_proba[y_test == 1, 1], bins=20, alpha=0.7, label='Class 1', edgecolor='black')
plt.xlabel('Predicted Probability of Class 1')
plt.ylabel('Count')
plt.title('Probability Distribution by True Class')
plt.legend()
plt.grid(True, alpha=0.3)
```

```python
# ROC curve
from sklearn.metrics import roc_curve, auc

fpr, tpr, thresholds = roc_curve(y_test, y_proba[:, 1])
roc_auc = auc(fpr, tpr)

plt.subplot(1, 2, 2)
plt.plot(fpr, tpr, linewidth=2, label=f'ROC (AUC = {roc_auc:.3f})')
plt.plot([0, 1], [0, 1], 'k--', linewidth=1, label='Random')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve')
plt.legend()
plt.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

print(f"\nAUC Score: {roc_auc:.4f}")
```

Expected: AUC should be > 0.9 for this dataset

Afternoon Session (4 hours)

Hands-on Coding - Part 2 (3.5 hours)

Exercise 3: Multi-Class Classification (50 min)

```python
from sklearn.datasets import load_iris
from sklearn.metrics import classification_report, ConfusionMatrixDisplay

# Load Iris dataset
iris = load_iris()
X, y = iris.data, iris.target

# Split data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Train multi-class logistic regression
model = LogisticRegression(max_iter=200, multi_class='multinomial')
model.fit(X_train, y_train)

# Predictions
y_pred = model.predict(X_test)

# Detailed evaluation
print("Classification Report:")
print(classification_report(y_test, y_pred, target_names=iris.target_names))

# Confusion matrix visualization
fig, ax = plt.subplots(figsize=(8, 6))
ConfusionMatrixDisplay.from_predictions(y_test, y_pred,
                                        display_labels=iris.target_names,
                                        cmap='Blues', ax=ax)
plt.title('Confusion Matrix - Iris Dataset')
plt.show()
```

```python
# Feature importance visualization
feature_importance = np.abs(model.coef_).mean(axis=0)
plt.figure(figsize=(10, 6))
plt.barh(iris.feature_names, feature_importance)
plt.xlabel('Average Absolute Coefficient')
plt.title('Feature Importance')
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()
```

Expected: High accuracy (>0.95) on Iris dataset

Exercise 4: Cross-Validation (40 min)

```python
from sklearn.model_selection import cross_val_score, cross_validate

# Simple cross-validation
scores = cross_val_score(model, X, y, cv=5)
print(f"Cross-validation scores: {scores}")
print(f"Mean accuracy: {scores.mean():.4f} (+/- {scores.std() * 2:.4f})")

# Detailed cross-validation with multiple metrics
scoring = ['accuracy', 'precision_macro', 'recall_macro', 'f1_macro']
cv_results = cross_validate(model, X, y, cv=5, scoring=scoring)

print("\nDetailed Cross-Validation Results:")
for metric in scoring:
    scores = cv_results[f'test_{metric}']
    print(f"{metric:20s}: {scores.mean():.4f} (+/- {scores.std() * 2:.4f})")

# Visualize CV results
plt.figure(figsize=(10, 6))
plt.boxplot([cv_results[f'test_{metric}'] for metric in scoring],
            labels=scoring)
plt.ylabel('Score')
plt.title('Cross-Validation Performance')
plt.grid(True, alpha=0.3)
plt.xticks(rotation=15)
plt.tight_layout()
plt.show()
```

Mini-Challenge: Breast Cancer Classification (90 min) Complete end-to-end binary classification project:

```python
from sklearn.datasets import load_breast_cancer
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import GridSearchCV

# Load dataset
cancer = load_breast_cancer()
X, y = cancer.data, cancer.target

print(f"Dataset: {cancer.data.shape[0]} samples, {cancer.data.shape[1]} features")
print(f"Classes: {cancer.target_names}")
print(f"Class distribution: {np.bincount(y)}")
```

```
# Split data
X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.3, random_state=42)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5, random_state=4

# Scale features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_val_scaled = scaler.transform(X_val)
X_test_scaled = scaler.transform(X_test)

# Hyperparameter tuning with GridSearchCV
param_grid = {
    'C': [0.01, 0.1, 1, 10, 100],
    'penalty': ['l1', 'l2'],
    'solver': ['liblinear']
}

grid_search = GridSearchCV(LogisticRegression(max_iter=1000),
                           param_grid, cv=5, scoring='f1')
grid_search.fit(X_train_scaled, y_train)

print(f"\nBest parameters: {grid_search.best_params_}")
print(f"Best cross-validation F1: {grid_search.best_score_:.4f}")

# Train best model
best_model = grid_search.best_estimator_

# Comprehensive evaluation
from sklearn.metrics import precision_score, recall_score, f1_score

y_val_pred = best_model.predict(X_val_scaled)
y_val_proba = best_model.predict_proba(X_val_scaled)[:, 1]

print("\nValidation Set Performance:")
print(f"Accuracy:  {accuracy_score(y_val, y_val_pred):.4f}")
print(f"Precision: {precision_score(y_val, y_val_pred):.4f}")
print(f"Recall:    {recall_score(y_val, y_val_pred):.4f}")
print(f"F1 Score:  {f1_score(y_val, y_val_pred):.4f}")

# Final test set evaluation
y_test_pred = best_model.predict(X_test_scaled)
y_test_proba = best_model.predict_proba(X_test_scaled)[:, 1]

print("\nTest Set Performance:")
print(classification_report(y_test, y_test_pred, target_names=cancer.target_names))

# Visualizations
fig, axes = plt.subplots(2, 2, figsize=(14, 12))

# Confusion matrix
from sklearn.metrics import ConfusionMatrixDisplay
ConfusionMatrixDisplay.from_predictions(y_test, y_test_pred,
                                        display_labels=cancer.target_names,
                                        cmap='Blues', ax=axes[0, 0])
axes[0, 0].set_title('Confusion Matrix')
```

```python
# ROC curve
fpr, tpr, _ = roc_curve(y_test, y_test_proba)
roc_auc = auc(fpr, tpr)
axes[0, 1].plot(fpr, tpr, linewidth=2, label=f'AUC = {roc_auc:.3f}')
axes[0, 1].plot([0, 1], [0, 1], 'k--')
axes[0, 1].set_xlabel('False Positive Rate')
axes[0, 1].set_ylabel('True Positive Rate')
axes[0, 1].set_title('ROC Curve')
axes[0, 1].legend()
axes[0, 1].grid(True, alpha=0.3)

# Feature importance (top 10)
feature_importance = np.abs(best_model.coef_[0])
top_features_idx = np.argsort(feature_importance)[-10:]
axes[1, 0].barh(range(10), feature_importance[top_features_idx])
axes[1, 0].set_yticks(range(10))
axes[1, 0].set_yticklabels([cancer.feature_names[i] for i in top_features_idx])
axes[1, 0].set_xlabel('Absolute Coefficient')
axes[1, 0].set_title('Top 10 Important Features')
axes[1, 0].grid(True, alpha=0.3)

# Probability distribution
axes[1, 1].hist(y_test_proba[y_test == 0], bins=20, alpha=0.7,
                label=cancer.target_names[0], edgecolor='black')
axes[1, 1].hist(y_test_proba[y_test == 1], bins=20, alpha=0.7,
                label=cancer.target_names[1], edgecolor='black')
axes[1, 1].set_xlabel('Predicted Probability of Malignant')
axes[1, 1].set_ylabel('Count')
axes[1, 1].set_title('Prediction Probabilities')
axes[1, 1].legend()
axes[1, 1].grid(True, alpha=0.3)

plt.tight_layout()
plt.show()
```

Reflection & Consolidation (30 min)

??? Review classification metrics ??? Understand ROC curves and AUC ??? Write daily reflection (choose 2-3 prompts) ??? Prepare questions for Friday check-in

Daily Reflection Prompts (Choose 2-3):

- What was the most important concept you learned today?
- How does logistic regression differ from linear regression?
- What is the difference between accuracy, precision, and recall?
- When would you prioritize precision over recall, or vice versa?
- What does the AUC score tell you about a model?
- What surprised you about the breast cancer classification results?

Next: Day 5 - Scikit-learn & Titanic Project

Week 1, Day 5: Scikit-learn & Titanic Project

Daily Goals

- Master scikit-learn pipelines and workflows
- Understand various ML algorithms (Decision Trees, Random Forests, SVM, KNN)
- Learn feature engineering and preprocessing
- Complete end-to-end Titanic survival prediction project
- Consolidate Week 1 learning

Morning Session (4 hours)

Optional: Daily Check-in with Peers on Teams (15 min)

Video Learning (90 min)

??? **Watch**: Decision Trees by StatQuest (17 min)

??? **Watch**: Decision Trees Part 2 - Feature Selection by StatQuest (5 min)

??? **Watch**: Random Forests Part 1 by StatQuest (10 min)

??? **Watch**: Random Forests Part 2 by StatQuest (14 min)

??? **Watch**: K-Nearest Neighbors by StatQuest (6 min)

??? **Watch**: Support Vector Machines Part 1 by StatQuest (20 min)

??? **Watch**: Principal Component Analysis (PCA) by StatQuest (20 min)

Reference Material (30 min)

??? **Read**: Scikit-learn User Guide - Sections 1.1-1.4

??? **Bookmark**: Scikit-learn Algorithm Cheat Sheet

Hands-on Coding - Part 1 (2 hours)

Exercise 1: Comparing Multiple Algorithms (60 min)

```python
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
from sklearn.neighbors import KNeighborsClassifier
from sklearn.preprocessing import StandardScaler
import numpy as np
import matplotlib.pyplot as plt

# Generate classification dataset
X, y = make_classification(n_samples=1000, n_features=20, n_informative=15,
                          n_redundant=5, random_state=42)

# Split data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Scale features
```

```python
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Define models
models = {
    'Logistic Regression': LogisticRegression(max_iter=1000),
    'Decision Tree': DecisionTreeClassifier(random_state=42),
    'Random Forest': RandomForestClassifier(n_estimators=100, random_state=42),
    'SVM': SVC(kernel='rbf', random_state=42),
    'KNN': KNeighborsClassifier(n_neighbors=5)
}

# Compare models
results = {}

print("Model Comparison:")
print("-" * 60)

for name, model in models.items():
    # Cross-validation scores
    cv_scores = cross_val_score(model, X_train_scaled, y_train, cv=5)

    # Train and test
    model.fit(X_train_scaled, y_train)
    train_score = model.score(X_train_scaled, y_train)
    test_score = model.score(X_test_scaled, y_test)

    results[name] = {
        'cv_mean': cv_scores.mean(),
        'cv_std': cv_scores.std(),
        'train': train_score,
        'test': test_score
    }

    print(f"{name:20s}: CV={cv_scores.mean():.4f}(??{cv_scores.std():.4f}) "
          f"Train={train_score:.4f} Test={test_score:.4f}")

# Visualize results
fig, axes = plt.subplots(1, 2, figsize=(14, 5))

# CV scores comparison
model_names = list(results.keys())
cv_means = [results[m]['cv_mean'] for m in model_names]
cv_stds = [results[m]['cv_std'] for m in model_names]

axes[0].bar(range(len(model_names)), cv_means, yerr=cv_stds, capsize=5)
axes[0].set_xticks(range(len(model_names)))
axes[0].set_xticklabels(model_names, rotation=45, ha='right')
axes[0].set_ylabel('Cross-Validation Score')
axes[0].set_title('Model Comparison - Cross-Validation')
axes[0].grid(True, alpha=0.3)

# Train vs Test scores
train_scores = [results[m]['train'] for m in model_names]
test_scores = [results[m]['test'] for m in model_names]
```

```
x = np.arange(len(model_names))
width = 0.35

axes[1].bar(x - width/2, train_scores, width, label='Train', alpha=0.8)
axes[1].bar(x + width/2, test_scores, width, label='Test', alpha=0.8)
axes[1].set_xticks(x)
axes[1].set_xticklabels(model_names, rotation=45, ha='right')
axes[1].set_ylabel('Accuracy')
axes[1].set_title('Train vs Test Accuracy')
axes[1].legend()
axes[1].grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

# Identify best model
best_model_name = max(results, key=lambda x: results[x]['test'])
print(f"\nBest performing model: {best_model_name}")
```

Expected: Random Forest and SVM typically perform well

Exercise 2: Scikit-learn Pipelines and GridSearchCV (60 min)

```
from sklearn.pipeline import Pipeline
from sklearn.model_selection import GridSearchCV

# Create pipeline
pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('classifier', RandomForestClassifier(random_state=42))
])

# Define parameter grid
param_grid = {
    'classifier__n_estimators': [50, 100, 200],
    'classifier__max_depth': [5, 10, 15, None],
    'classifier__min_samples_split': [2, 5, 10]
}

# Grid search with cross-validation
grid_search = GridSearchCV(pipeline, param_grid, cv=5,
                          scoring='accuracy', n_jobs=-1, verbose=1)

print("Running Grid Search...")
grid_search.fit(X_train, y_train)

print(f"\nBest parameters: {grid_search.best_params_}")
print(f"Best CV score: {grid_search.best_score_:.4f}")
print(f"Test score: {grid_search.score(X_test, y_test):.4f}")

# Visualize grid search results
results_df = pd.DataFrame(grid_search.cv_results_)

# Plot mean test scores for different n_estimators
import pandas as pd
```

```
for max_depth in [5, 10, 15, None]:
    subset = results_df[results_df['param_classifier__max_depth'] == max_depth]
    if len(subset) > 0:
        estimators = subset['param_classifier__n_estimators']
        scores = subset['mean_test_score']
        plt.plot(estimators, scores, 'o-', label=f'max_depth={max_depth}')

plt.xlabel('Number of Estimators')
plt.ylabel('Mean CV Score')
plt.title('Grid Search Results')
plt.legend()
plt.grid(True, alpha=0.3)
plt.show()
```

Expected: Performance improves with more estimators, but plateaus

---

Afternoon Session (4 hours)

Titanic Survival Prediction Project (3.5 hours)

Complete end-to-end machine learning project with the famous Titanic dataset.

Phase 1: Data Exploration (30 min)

```
import pandas as pd
import seaborn as sns

# Load Titanic dataset
titanic_url = 'https://raw.githubusercontent.com/datasciencedojo/datasets/master/titanic.csv'
df = pd.read_csv(titanic_url)

print("Titanic Dataset Overview:")
print(f"Shape: {df.shape}")
print(f"\nColumn Info:")
print(df.info())

print(f"\nFirst few rows:")
print(df.head())

print(f"\nSurvival rate: {df['Survived'].mean():.2%}")
print(f"\nMissing values:")
print(df.isnull().sum())

# Exploratory visualizations
fig, axes = plt.subplots(2, 3, figsize=(16, 10))

# Survival distribution
df['Survived'].value_counts().plot(kind='bar', ax=axes[0,0])
axes[0,0].set_title('Survival Distribution')
axes[0,0].set_xticklabels(['Died', 'Survived'], rotation=0)
axes[0,0].set_ylabel('Count')

# Survival by class
pd.crosstab(df['Pclass'], df['Survived']).plot(kind='bar', ax=axes[0,1])
axes[0,1].set_title('Survival by Class')
axes[0,1].set_xlabel('Class')
```

```
axes[0,1].set_xticklabels(['1st', '2nd', '3rd'], rotation=0)
axes[0,1].legend(['Died', 'Survived'])

# Survival by sex
pd.crosstab(df['Sex'], df['Survived']).plot(kind='bar', ax=axes[0,2])
axes[0,2].set_title('Survival by Sex')
axes[0,2].set_xticklabels(['Female', 'Male'], rotation=0)
axes[0,2].legend(['Died', 'Survived'])

# Age distribution
df['Age'].hist(bins=30, ax=axes[1,0], edgecolor='black')
axes[1,0].set_title('Age Distribution')
axes[1,0].set_xlabel('Age')

# Fare distribution
df['Fare'].hist(bins=30, ax=axes[1,1], edgecolor='black')
axes[1,1].set_title('Fare Distribution')
axes[1,1].set_xlabel('Fare')

# Correlation heatmap (numeric features only)
numeric_cols = df.select_dtypes(include=[np.number]).columns
correlation = df[numeric_cols].corr()
sns.heatmap(correlation, annot=True, fmt='.2f', cmap='coolwarm', ax=axes[1,2])
axes[1,2].set_title('Feature Correlations')

plt.tight_layout()
plt.show()
```

Phase 2: Data Preprocessing & Feature Engineering (45 min)

```
# Create a copy for preprocessing
df_processed = df.copy()

# Feature engineering
print("Creating new features...")

# 1. Family size
df_processed['FamilySize'] = df_processed['SibSp'] + df_processed['Parch'] + 1

# 2. Is alone
df_processed['IsAlone'] = (df_processed['FamilySize'] == 1).astype(int)

# 3. Title from name
df_processed['Title'] = df_processed['Name'].str.extract(' ([A-Za-z]+)\.', expand=False)

# Group rare titles
title_mapping = {
    'Mr': 'Mr', 'Miss': 'Miss', 'Mrs': 'Mrs', 'Master': 'Master',
    'Dr': 'Rare', 'Rev': 'Rare', 'Col': 'Rare', 'Major': 'Rare',
    'Mlle': 'Miss', 'Mme': 'Mrs', 'Don': 'Rare', 'Dona': 'Rare',
    'Lady': 'Rare', 'Countess': 'Rare', 'Jonkheer': 'Rare', 'Sir': 'Rare',
    'Capt': 'Rare', 'Ms': 'Miss'
}
df_processed['Title'] = df_processed['Title'].map(title_mapping)
df_processed['Title'].fillna('Rare', inplace=True)

# 4. Fare per person
```

```
df_processed['FarePerPerson'] = df_processed['Fare'] / df_processed['FamilySize']

# Handle missing values
print("\nHandling missing values...")

# Age: Fill with median by title and class
for title in df_processed['Title'].unique():
    for pclass in df_processed['Pclass'].unique():
        mask = (df_processed['Title'] == title) & (df_processed['Pclass'] == pclass)
        median_age = df_processed[mask]['Age'].median()
        df_processed.loc[mask & df_processed['Age'].isnull(), 'Age'] = median_age

# Embarked: Fill with mode
df_processed['Embarked'].fillna(df_processed['Embarked'].mode()[0], inplace=True)

# Fare: Fill with median
df_processed['Fare'].fillna(df_processed['Fare'].median(), inplace=True)
df_processed['FarePerPerson'].fillna(df_processed['FarePerPerson'].median(), inplace=True)

# Drop unnecessary columns
columns_to_drop = ['PassengerId', 'Name', 'Ticket', 'Cabin']
df_processed.drop(columns_to_drop, axis=1, inplace=True)

print(f"\nMissing values after preprocessing:")
print(df_processed.isnull().sum())

# Encode categorical variables
from sklearn.preprocessing import LabelEncoder

le_sex = LabelEncoder()
le_embarked = LabelEncoder()
le_title = LabelEncoder()

df_processed['Sex'] = le_sex.fit_transform(df_processed['Sex'])
df_processed['Embarked'] = le_embarked.fit_transform(df_processed['Embarked'])
df_processed['Title'] = le_title.fit_transform(df_processed['Title'])

print(f"\nProcessed dataset shape: {df_processed.shape}")
print(f"Features: {df_processed.columns.tolist()}")
```

Phase 3: Model Training & Comparison (45 min)

```
# Separate features and target
X = df_processed.drop('Survived', axis=1)
y = df_processed['Survived']

# Split data (60-20-20)
X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.4, random_state=42)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5, random_state=4

print(f"Train: {len(X_train)}, Val: {len(X_val)}, Test: {len(X_test)}")

# Scale features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_val_scaled = scaler.transform(X_val)
X_test_scaled = scaler.transform(X_test)
```

```python
# Train multiple models
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, roc_auc_

models_titanic = {
    'Logistic Regression': LogisticRegression(max_iter=1000, random_state=42),
    'Decision Tree': DecisionTreeClassifier(max_depth=5, random_state=42),
    'Random Forest': RandomForestClassifier(n_estimators=100, max_depth=10, random_state=42),
    'SVM': SVC(kernel='rbf', probability=True, random_state=42),
    'KNN': KNeighborsClassifier(n_neighbors=7)
}

results_titanic = []

print("\nModel Performance on Titanic Dataset:")
print("=" * 80)

for name, model in models_titanic.items():
    # Train
    model.fit(X_train_scaled, y_train)

    # Predict on validation set
    y_val_pred = model.predict(X_val_scaled)
    y_val_proba = model.predict_proba(X_val_scaled)[:, 1] if hasattr(model, 'predict_proba')

    # Calculate metrics
    accuracy = accuracy_score(y_val, y_val_pred)
    precision = precision_score(y_val, y_val_pred)
    recall = recall_score(y_val, y_val_pred)
    f1 = f1_score(y_val, y_val_pred)
    roc_auc = roc_auc_score(y_val, y_val_proba) if y_val_proba is not None else None

    results_titanic.append({
        'Model': name,
        'Accuracy': accuracy,
        'Precision': precision,
        'Recall': recall,
        'F1': f1,
        'ROC-AUC': roc_auc
    })

    print(f"{name:20s}: Acc={accuracy:.4f}, Prec={precision:.4f}, "
          f"Rec={recall:.4f}, F1={f1:.4f}, AUC={roc_auc:.4f if roc_auc else 'N/A'}")

# Convert to DataFrame
results_df = pd.DataFrame(results_titanic)
print("\n" + results_df.to_string(index=False))
```

Phase 4: Hyperparameter Optimization (45 min)

```python
# Select Random Forest for optimization
from sklearn.model_selection import RandomizedSearchCV

param_dist = {
    'n_estimators': [50, 100, 200, 300],
    'max_depth': [3, 5, 7, 10, None],
    'min_samples_split': [2, 5, 10, 20],
    'min_samples_leaf': [1, 2, 4, 8],
    'max_features': ['sqrt', 'log2', None]
}

rf = RandomForestClassifier(random_state=42)

random_search = RandomizedSearchCV(
    rf, param_distributions=param_dist,
    n_iter=20, cv=5, scoring='roc_auc',
    random_state=42, n_jobs=-1, verbose=1
)

print("Running Randomized Search...")
random_search.fit(X_train_scaled, y_train)

print(f"\nBest parameters: {random_search.best_params_}")
print(f"Best CV ROC-AUC: {random_search.best_score_:.4f}")

# Evaluate best model on validation set
best_rf = random_search.best_estimator_
y_val_pred_best = best_rf.predict(X_val_scaled)
y_val_proba_best = best_rf.predict_proba(X_val_scaled)[:, 1]

print(f"\nValidation Performance (Optimized Model):")
print(f"Accuracy:  {accuracy_score(y_val, y_val_pred_best):.4f}")
print(f"Precision: {precision_score(y_val, y_val_pred_best):.4f}")
print(f"Recall:    {recall_score(y_val, y_val_pred_best):.4f}")
print(f"F1 Score:  {f1_score(y_val, y_val_pred_best):.4f}")
print(f"ROC-AUC:   {roc_auc_score(y_val, y_val_proba_best):.4f}")
```

Phase 5: Final Evaluation & Analysis (45 min)

```
# Final evaluation on test set
y_test_pred = best_rf.predict(X_test_scaled)
y_test_proba = best_rf.predict_proba(X_test_scaled)[:, 1]

print("\n" + "="*60)
print("FINAL TEST SET PERFORMANCE")
print("="*60)
print(f"Accuracy:  {accuracy_score(y_test, y_test_pred):.4f}")
print(f"Precision: {precision_score(y_test, y_test_pred):.4f}")
print(f"Recall:    {recall_score(y_test, y_test_pred):.4f}")
print(f"F1 Score:  {f1_score(y_test, y_test_pred):.4f}")
print(f"ROC-AUC:   {roc_auc_score(y_test, y_test_proba):.4f}")

# Comprehensive visualizations
fig, axes = plt.subplots(2, 2, figsize=(14, 12))

# Confusion Matrix
from sklearn.metrics import ConfusionMatrixDisplay
ConfusionMatrixDisplay.from_predictions(y_test, y_test_pred,
                                        display_labels=['Died', 'Survived'],
                                        cmap='Blues', ax=axes[0, 0])
axes[0, 0].set_title('Confusion Matrix - Test Set')

# ROC Curve
from sklearn.metrics import roc_curve, auc
```

```python
fpr, tpr, _ = roc_curve(y_test, y_test_proba)
roc_auc_val = auc(fpr, tpr)
axes[0, 1].plot(fpr, tpr, linewidth=2, label=f'AUC = {roc_auc_val:.3f}')
axes[0, 1].plot([0, 1], [0, 1], 'k--', linewidth=1)
axes[0, 1].set_xlabel('False Positive Rate')
axes[0, 1].set_ylabel('True Positive Rate')
axes[0, 1].set_title('ROC Curve')
axes[0, 1].legend()
axes[0, 1].grid(True, alpha=0.3)

# Feature Importance
feature_importance = best_rf.feature_importances_
feature_names = X.columns
sorted_idx = np.argsort(feature_importance)[-10:]  # Top 10

axes[1, 0].barh(range(len(sorted_idx)), feature_importance[sorted_idx])
axes[1, 0].set_yticks(range(len(sorted_idx)))
axes[1, 0].set_yticklabels([feature_names[i] for i in sorted_idx])
axes[1, 0].set_xlabel('Importance')
axes[1, 0].set_title('Top 10 Feature Importance')
axes[1, 0].grid(True, alpha=0.3)

# Prediction Probability Distribution
axes[1, 1].hist(y_test_proba[y_test == 0], bins=20, alpha=0.7,
                label='Died', edgecolor='black')
axes[1, 1].hist(y_test_proba[y_test == 1], bins=20, alpha=0.7,
                label='Survived', edgecolor='black')
axes[1, 1].set_xlabel('Predicted Probability of Survival')
axes[1, 1].set_ylabel('Count')
axes[1, 1].set_title('Prediction Probabilities by True Class')
axes[1, 1].legend()
axes[1, 1].grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

# Detailed feature importance analysis
print("\nFeature Importance Ranking:")
print("="*60)
feature_imp_df = pd.DataFrame({
    'Feature': feature_names,
    'Importance': feature_importance
}).sort_values('Importance', ascending=False)

for idx, row in feature_imp_df.iterrows():
    print(f"{row['Feature']:20s}: {row['Importance']:.4f}")

# Error analysis
print("\n" + "="*60)
print("ERROR ANALYSIS")
print("="*60)

# Find misclassified samples
misclassified_idx = np.where(y_test_pred != y_test)[0]
print(f"Number of misclassified samples: {len(misclassified_idx)}")
```

```
if len(misclassified_idx) > 0:
    # Show a few examples
    print("\nSample Misclassifications:")
    for i in misclassified_idx[:5]:
        true_label = 'Survived' if y_test.iloc[i] == 1 else 'Died'
        pred_label = 'Survived' if y_test_pred[i] == 1 else 'Died'
        confidence = y_test_proba[i] if y_test_pred[i] == 1 else 1 - y_test_proba[i]

        print(f"\nSample {i}:")
        print(f"  True: {true_label}, Predicted: {pred_label}, Confidence: {confidence:.2%}")
        print(f"  Features: {X_test.iloc[i].to_dict()}")

print("\n" + "="*60)
print("PROJECT COMPLETE!")
print("="*60)
```

Reflection & Consolidation (30 min)

??? Review entire Week 1 journey ??? Document your Titanic project results ??? Write comprehensive reflection (address all prompts) ??? Prepare for Week 1 review and Week 2 preview

Daily Reflection Prompts (Address All):

- What was the most important concept you learned this week?
- How did the Titanic project bring together all Week 1 concepts?
- What was your biggest challenge this week? How did you overcome it?
- Which machine learning algorithm performed best on Titanic? Why?
- What surprised you most about feature engineering?
- Review Days 1-5: What are the key takeaways from each day?
- How confident do you feel about starting Week 2?
- What topics from Week 1 need more practice?
- What are you most excited to learn in Week 2?

Week 1 Complete!

Congratulations on completing Week 1! You now understand:

??? NumPy for numerical computing ??? Matplotlib for visualization ??? Core ML concepts (supervised learning, overfitting, regularization) ??? Linear and logistic regression ??? Multiple ML algorithms (Decision Trees, Random Forests, SVM, KNN) ??? Scikit-learn pipelines and workflows ??? End-to-end ML project execution

Weekend Activities:

- Review your Week 1 notes
- Revisit challenging exercises
- Share your Titanic results with peers
- Preview Week 2 topics

Next Week: Neural Networks and Deep Learning with PyTorch!

Next: Week 2 Overview

Week 2 Overview: Neural Networks Foundations

Introduction

Week 2 introduces neural networks - the foundation of modern deep learning. You'll understand how neural networks work from first principles, implement them from scratch, then master PyTorch to build production-ready models.

Week Goals

- Understand neural network architecture and forward propagation
- Master backpropagation and the chain rule
- Implement neural networks from scratch in NumPy
- Learn PyTorch fundamentals (tensors, autograd, nn.Module)
- Apply regularization techniques (dropout, batch normalization)
- Complete MNIST digit classification project achieving >98% accuracy

Weekly Structure

- **Day 6**: Neural Network Theory and Forward Propagation
- **Day 7**: Backpropagation and Training
- **Day 8**: Introduction to PyTorch
- **Day 9**: Building Neural Networks in PyTorch
- **Day 10**: MNIST Project - Digit Classification

Key Resources

Videos: 3Blue1Brown (primary), StatQuest (supporting), PyTorch tutorials (practical) **Text:** Dive into Deep Learning Chapter 5, PyTorch documentation

Tips for Success

- Watch 3Blue1Brown videos actively - pause and predict
- Implement from scratch before using PyTorch (Days 6-7)
- Type code, don't copy-paste
- Debug systematically - check shapes and gradients
- Complete daily reflections

Ready to start? Begin with Day 6: Neural Network Theory

Week 2, Day 6: Neural Network Theory and Forward Propagation

Daily Goals

- Understand neural network architecture (layers, nodes, connections)
- Learn about activation functions and their purposes
- Implement forward propagation from scratch in NumPy
- Grasp the intuition behind how neural networks transform data
- Solve the XOR problem with a neural network

Morning Session (4 hours)

Optional: Daily Check-in with Peers on Teams (15 min)

Video Learning (90 min)

??? **Watch**: But what is a neural network? by 3Blue1Brown (19 min) *This is THE best introduction to neural networks. Watch it carefully.*

??? **Watch**: Gradient descent, how neural networks learn by 3Blue1Brown (21 min) *Builds intuition for the learning process*

??? **Watch**: Neural Networks Part 1: Setup by StatQuest (20 min) *Alternative perspective - good for reinforcement*

??? **Watch**: Activation Functions by StatQuest (9 min) *Understand why we need non-linearity*

??? **Watch**: Neural Network Architectures by StatQuest (8 min) *Different ways to structure networks*

Reference Material (30 min)

??? **Read**: D2L Chapter 5.1 - Multilayer Perceptrons *Mathematical foundation for neural networks*

??? **Read**: D2L Chapter 5.2 - Implementation from Scratch *See a complete implementation before building your own*

Hands-on Coding - Part 1 (2 hours)

Setup (10 min)

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

np.random.seed(42)
```

Exercise 1: Activation Functions (40 min) Implement and visualize the key activation functions:

1. Sigmoid

```python
def sigmoid(x):
    """
    Sigmoid activation: ??(x) = 1 / (1 + e^(-x))
    Maps input to (0, 1)
    """
    return 1 / (1 + np.exp(-x))

def sigmoid_derivative(x):
    """
    Derivative: ??'(x) = ??(x) * (1 - ??(x))
    """
    s = sigmoid(x)
    return s * (1 - s)

# Test
x = np.linspace(-10, 10, 100)
y = sigmoid(x)
y_prime = sigmoid_derivative(x)

plt.figure(figsize=(10, 4))
plt.subplot(1, 2, 1)
plt.plot(x, y, label='sigmoid(x)', linewidth=2)
plt.grid(True, alpha=0.3)
plt.xlabel('x')
plt.ylabel('sigmoid(x)')
plt.title('Sigmoid Function')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(x, y_prime, label="sigmoid'(x)", color='orange', linewidth=2)
plt.grid(True, alpha=0.3)
plt.xlabel('x')
plt.ylabel("sigmoid'(x)")
plt.title('Sigmoid Derivative')
plt.legend()
plt.tight_layout()
plt.show()
```

Expected: S-curve for sigmoid, bell curve for derivative

2. Tanh

```python
def tanh(x):
    """
    Hyperbolic tangent: tanh(x) = (e^x - e^(-x)) / (e^x + e^(-x))
    Maps input to (-1, 1)
    """
    return np.tanh(x)

def tanh_derivative(x):
    """
    Derivative: tanh'(x) = 1 - tanh??(x)
    """
    t = tanh(x)
    return 1 - t**2

# Visualize
y_tanh = tanh(x)
y_tanh_prime = tanh_derivative(x)

plt.figure(figsize=(10, 4))
plt.subplot(1, 2, 1)
plt.plot(x, y_tanh, label='tanh(x)', color='green', linewidth=2)
plt.axhline(y=0, color='k', linestyle='--', alpha=0.3)
plt.grid(True, alpha=0.3)
plt.xlabel('x')
plt.title('Tanh Function')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(x, y_tanh_prime, label="tanh'(x)", color='red', linewidth=2)
plt.grid(True, alpha=0.3)
plt.xlabel('x')
plt.title('Tanh Derivative')
plt.legend()
plt.tight_layout()
plt.show()
```

Expected: Similar to sigmoid but centered at zero

3. ReLU (Rectified Linear Unit)

```python
def relu(x):
    """
    ReLU: max(0, x)
    Most popular activation for hidden layers
    """
    return np.maximum(0, x)

def relu_derivative(x):
    """
    Derivative: 1 if x > 0, else 0
    """
    return (x > 0).astype(float)

# Visualize
y_relu = relu(x)
y_relu_prime = relu_derivative(x)

plt.figure(figsize=(10, 4))
plt.subplot(1, 2, 1)
plt.plot(x, y_relu, label='ReLU(x)', color='purple', linewidth=2)
plt.grid(True, alpha=0.3)
plt.xlabel('x')
plt.title('ReLU Function')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(x, y_relu_prime, label="ReLU'(x)", color='brown', linewidth=2)
plt.grid(True, alpha=0.3)
plt.xlabel('x')
plt.title('ReLU Derivative')
plt.legend()
plt.tight_layout()
plt.show()
```

Expected: Straight line for x>0, flat for x<0

4. Compare All

```python
plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)
plt.plot(x, sigmoid(x), label='Sigmoid', linewidth=2)
plt.plot(x, tanh(x), label='Tanh', linewidth=2)
plt.plot(x, relu(x), label='ReLU', linewidth=2)
plt.grid(True, alpha=0.3)
plt.xlabel('x')
plt.ylabel('f(x)')
plt.title('Activation Functions Comparison')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(x, sigmoid_derivative(x), label="Sigmoid'", linewidth=2)
plt.plot(x, tanh_derivative(x), label="Tanh'", linewidth=2)
plt.plot(x, relu_derivative(x), label="ReLU'", linewidth=2)
plt.grid(True, alpha=0.3)
plt.xlabel('x')
plt.ylabel("f'(x)")
plt.title('Derivatives Comparison')
plt.legend()

plt.tight_layout()
plt.show()

print("Key observations:")
print("1. Sigmoid: outputs (0,1), derivative peaks at 0")
print("2. Tanh: outputs (-1,1), similar to sigmoid but centered")
print("3. ReLU: simple, efficient, derivative is 0 or 1")
print("4. ReLU avoids vanishing gradient problem for x > 0")
```

Exercise 2: Forward Propagation - Manual Calculation (30 min) Work through forward propagation by hand first:

```
# Simple network: 2 inputs -> 2 hidden -> 1 output
# Let's trace through an example

print("=" * 60)
print("MANUAL FORWARD PROPAGATION EXAMPLE")
print("=" * 60)

# Network architecture
print("\nArchitecture: 2 inputs ??? 2 hidden neurons ??? 1 output")

# Input
X = np.array([0.5, 0.8])
print(f"\nInput: X = {X}")

# Weights and biases (initialized randomly for now)
W1 = np.array([[0.2, 0.5],   # weights from input to hidden
               [0.3, 0.4]])
b1 = np.array([0.1, 0.2])     # biases for hidden layer

W2 = np.array([[0.6],          # weights from hidden to output
               [0.7]])
b2 = np.array([0.3])           # bias for output

print(f"\nW1 (input???hidden):\n{W1}")
print(f"b1 (hidden biases): {b1}")
print(f"\nW2 (hidden???output):\n{W2}")
print(f"b2 (output bias): {b2}")
```

```
# Hidden layer computation
print("\n" + "=" * 60)
print("HIDDEN LAYER")
print("=" * 60)

# Pre-activation (linear combination)
z1 = np.dot(X, W1) + b1
print(f"\nz1 = X??W1 + b1")
print(f"z1 = {X} ?? {W1.T} + {b1}")
print(f"z1 = {z1}")

# Activation
a1 = sigmoid(z1)
print(f"\na1 = sigmoid(z1)")
print(f"a1 = {a1}")

# Output layer computation
print("\n" + "=" * 60)
print("OUTPUT LAYER")
print("=" * 60)

# Pre-activation
z2 = np.dot(a1, W2) + b2
print(f"\nz2 = a1??W2 + b2")
print(f"z2 = {a1} ?? {W2.T} + {b2}")
print(f"z2 = {z2}")

# Activation
a2 = sigmoid(z2)
print(f"\na2 = sigmoid(z2)")
print(f"a2 = {a2}")

print(f"\n{'='*60}")
print(f"FINAL OUTPUT: {a2[0]:.4f}")
print(f"{'='*60}")
```

Expected: Follow the computation step-by-step, verify numbers

Exercise 3: Implement Neural Network Class (40 min) Build a complete neural network from scratch:

```
class NeuralNetwork:
    def __init__(self, input_size, hidden_size, output_size):
        """
        Initialize a 2-layer neural network

        Args:
            input_size: number of input features
            hidden_size: number of neurons in hidden layer
            output_size: number of output neurons
        """
        # Initialize weights with small random values
        # He initialization for better training
        self.W1 = np.random.randn(input_size, hidden_size) * 0.01
        self.b1 = np.zeros((1, hidden_size))

        self.W2 = np.random.randn(hidden_size, output_size) * 0.01
        self.b2 = np.zeros((1, output_size))

        print(f"Network initialized:")
        print(f"  Input size: {input_size}")
        print(f"  Hidden size: {hidden_size}")
        print(f"  Output size: {output_size}")
        print(f"  Total parameters: {self.count_parameters()}")

    def count_parameters(self):
        """Count total number of parameters"""
        return (self.W1.size + self.b1.size +
                self.W2.size + self.b2.size)

    def forward(self, X):
        """
        Forward propagation

        Args:
            X: input data (n_samples, n_features)

        Returns:
            output: predictions (n_samples, n_outputs)
        """
        # Hidden layer
        self.z1 = np.dot(X, self.W1) + self.b1
        self.a1 = sigmoid(self.z1)

        # Output layer
        self.z2 = np.dot(self.a1, self.W2) + self.b2
        self.a2 = sigmoid(self.z2)

        return self.a2

    def __repr__(self):
        return (f"NeuralNetwork(\n"
                f"  W1: {self.W1.shape},\n"
                f"  b1: {self.b1.shape},\n"
                f"  W2: {self.W2.shape},\n"
                f"  b2: {self.b2.shape}\n"
                f")")

# Test the network
print("\n" + "="*60)
print("TESTING NEURAL NETWORK CLASS")
print("="*60)

# Create network for XOR problem (2 inputs, 2 hidden, 1 output)
nn = NeuralNetwork(input_size=2, hidden_size=2, output_size=1)
print(f"\n{nn}")

# Test forward propagation
X_test = np.array([[0, 0],
                   [0, 1],
                   [1, 0],
                   [1, 1]])
```

```
print("\nForward propagation test (random weights):")
predictions = nn.forward(X_test)

for i, (x, pred) in enumerate(zip(X_test, predictions)):
    print(f"Input: {x} ??? Output: {pred[0]:.4f}")

print("\nNote: These are random outputs since we haven't trained yet!")
```

Expected: Network runs without errors, produces random outputs

---

Afternoon Session (4 hours)

Video Learning (30 min)

??? **Watch**: Why Neural Networks Can Learn Almost Anything by 3Blue1Brown-style (15 min)

??? **Review**: Replay key sections from morning videos as needed (15 min)

Hands-on Coding - Part 2 (3.5 hours)

Exercise 4: Understanding Network Capacity (50 min) Explore how hidden layer size affects learning capacity:

```
def visualize_network_capacity():
    """
    Create networks of different sizes and visualize their decision boundaries
    """
    # Generate simple 2D classification data
    np.random.seed(42)
    n_samples = 200

    # Create two circular clusters
    theta = np.random.uniform(0, 2*np.pi, n_samples//2)
    r1 = np.random.normal(1, 0.1, n_samples//2)
    r2 = np.random.normal(2, 0.1, n_samples//2)

    X_class0 = np.column_stack([r1 * np.cos(theta), r1 * np.sin(theta)])
    X_class1 = np.column_stack([r2 * np.cos(theta), r2 * np.sin(theta)])

    X = np.vstack([X_class0, X_class1])
    y = np.hstack([np.zeros(n_samples//2), np.ones(n_samples//2)]).reshape(-1, 1)

    # Try different hidden layer sizes
    hidden_sizes = [2, 5, 10, 20]

    fig, axes = plt.subplots(2, 2, figsize=(12, 12))
    axes = axes.flatten()

    for idx, hidden_size in enumerate(hidden_sizes):
        # Create network
        nn = NeuralNetwork(input_size=2, hidden_size=hidden_size, output_size=1)

        # Create decision boundary plot
        h = 0.1
        x_min, x_max = X[:, 0].min() - 0.5, X[:, 0].max() + 0.5
        y_min, y_max = X[:, 1].min() - 0.5, X[:, 1].max() + 0.5
        xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                             np.arange(y_min, y_max, h))

        # Get predictions for mesh
        mesh_predictions = nn.forward(np.c_[xx.ravel(), yy.ravel()])
        mesh_predictions = mesh_predictions.reshape(xx.shape)

        # Plot
        axes[idx].contourf(xx, yy, mesh_predictions, alpha=0.4, cmap='RdYlBu', levels=20)
        axes[idx].scatter(X[y.flatten()==0, 0], X[y.flatten()==0, 1],
                          c='red', edgecolors='k', label='Class 0', alpha=0.6)
        axes[idx].scatter(X[y.flatten()==1, 0], X[y.flatten()==1, 1],
                          c='blue', edgecolors='k', label='Class 1', alpha=0.6)
        axes[idx].set_title(f'Hidden Size = {hidden_size} ({nn.count_parameters()} params)')
        axes[idx].set_xlabel('Feature 1')
        axes[idx].set_ylabel('Feature 2')
        axes[idx].legend()
        axes[idx].grid(True, alpha=0.3)

    plt.suptitle('Network Capacity: Decision Boundaries with Random Weights', fontsize=14)
    plt.tight_layout()
    plt.show()

    print("\nObservations:")
    print("- Larger networks (more parameters) have more flexible decision boundaries")
    print("- Even untrained, you can see complexity differences")
    print("- With training, larger networks can fit more complex patterns")
    print("- But too large = overfitting risk!")

visualize_network_capacity()
```

Exercise 5: XOR Problem - The Classic Test (60 min) Implement the XOR problem to test your network:

```
print("=" * 60)
print("XOR PROBLEM - The Neural Network Classic")
print("=" * 60)

# XOR truth table
X_xor = np.array([[0, 0],
                  [0, 1],
                  [1, 0],
                  [1, 1]])
y_xor = np.array([[0],
                  [1],
                  [1],
                  [0]])

print("\nXOR Truth Table:")
print("Input  | Target")
print("-------|-------")
for x, y in zip(X_xor, y_xor):
    print(f"{x}  |   {y[0]}")

# Why XOR is important
print("\nWhy XOR matters:")
print("- XOR is NOT linearly separable")
print("- Single perceptron CANNOT solve it")
print("- Need hidden layer (non-linearity) to solve")
print("- Historically significant (ended first AI winter)")

# Visualize XOR
plt.figure(figsize=(8, 6))
colors = ['red' if y[0] == 0 else 'blue' for y in y_xor]
plt.scatter(X_xor[:, 0], X_xor[:, 1], c=colors, s=200, edgecolors='k', linewidth=2)

for i, (x, y) in enumerate(zip(X_xor, y_xor)):
    plt.annotate(f'{x} ??? {y[0]}', xy=x, xytext=(5, 5), textcoords='offset points')

plt.xlabel('Input 1')
plt.ylabel('Input 2')
plt.title('XOR Problem Visualization')
plt.grid(True, alpha=0.3)
plt.xlim(-0.5, 1.5)
plt.ylim(-0.5, 1.5)
plt.show()

# Test with our neural network (untrained)
nn_xor = NeuralNetwork(input_size=2, hidden_size=4, output_size=1)

print("\nUntrained network predictions on XOR:")
predictions = nn_xor.forward(X_xor)

for x, y_true, y_pred in zip(X_xor, y_xor, predictions):
    print(f"Input: {x} | Target: {y_true[0]} | Prediction: {y_pred[0]:.4f}")

print("\nThese are random because we haven't trained yet!")
print("Tomorrow (Day 7) we'll implement backpropagation to train this network!")
```

Mini-Challenge: Network Visualization Tool (90 min) Create comprehensive visualizations for neural networks:

```
def visualize_network_architecture(nn, title="Neural Network Architecture"):
    """
    Visualize network architecture with weights
    """
    fig, axes = plt.subplots(1, 3, figsize=(16, 5))

    # 1. Architecture diagram (simplified - you can elaborate)
    ax = axes[0]
    ax.text(0.1, 0.9, 'Input\nLayer', ha='center', va='center', fontsize=12,
            bbox=dict(boxstyle='round', facecolor='lightblue'))
    ax.text(0.5, 0.9, 'Hidden\nLayer', ha='center', va='center', fontsize=12,
            bbox=dict(boxstyle='round', facecolor='lightgreen'))
    ax.text(0.9, 0.9, 'Output\nLayer', ha='center', va='center', fontsize=12,
            bbox=dict(boxstyle='round', facecolor='lightcoral'))

    ax.annotate('', xy=(0.45, 0.9), xytext=(0.15, 0.9),
                arrowprops=dict(arrowstyle='->', lw=2))
    ax.annotate('', xy=(0.85, 0.9), xytext=(0.55, 0.9),
                arrowprops=dict(arrowstyle='->', lw=2))

    ax.text(0.5, 0.5, f'Parameters: {nn.count_parameters()}', ha='center', fontsize=11)
    ax.set_xlim(0, 1)
    ax.set_ylim(0, 1)
    ax.axis('off')
    ax.set_title('Network Structure')

    # 2. Weight matrix W1 (input to hidden)
    ax = axes[1]
    im = ax.imshow(nn.W1, cmap='RdBu', aspect='auto', vmin=-0.5, vmax=0.5)
    ax.set_title('W1: Input ??? Hidden Weights')
    ax.set_xlabel('Hidden Neurons')
    ax.set_ylabel('Input Features')
    plt.colorbar(im, ax=ax)

    # 3. Weight matrix W2 (hidden to output)
    ax = axes[2]
    im = ax.imshow(nn.W2, cmap='RdBu', aspect='auto', vmin=-0.5, vmax=0.5)
    ax.set_title('W2: Hidden ??? Output Weights')
    ax.set_xlabel('Output Neurons')
    ax.set_ylabel('Hidden Neurons')
    plt.colorbar(im, ax=ax)

    plt.suptitle(title, fontsize=14, fontweight='bold')
    plt.tight_layout()
    plt.show()

# Test the visualization
nn_vis = NeuralNetwork(input_size=4, hidden_size=6, output_size=2)
visualize_network_architecture(nn_vis, "Example Neural Network")

def visualize_activations(nn, X, title="Activation Flow"):
    """
    Visualize how activations flow through the network
    """
    # Get activations
    output = nn.forward(X)

    fig, axes = plt.subplots(1, 3, figsize=(15, 4))

    # Input
    ax = axes[0]
    ax.imshow(X.T, cmap='viridis', aspect='auto')
    ax.set_title(f'Input\n({X.shape[0]} samples, {X.shape[1]} features)')
    ax.set_xlabel('Sample')
    ax.set_ylabel('Feature')
    plt.colorbar(ax.images[0], ax=ax)

    # Hidden activations
    ax = axes[1]
    ax.imshow(nn.a1.T, cmap='viridis', aspect='auto')
    ax.set_title(f'Hidden Layer Activations\n({nn.a1.shape[1]} neurons)')
    ax.set_xlabel('Sample')
    ax.set_ylabel('Neuron')
    plt.colorbar(ax.images[0], ax=ax)

    # Output
    ax = axes[2]
    ax.imshow(output.T, cmap='viridis', aspect='auto')
    ax.set_title(f'Output\n({output.shape[1]} values)')
    ax.set_xlabel('Sample')
    ax.set_ylabel('Output')
    plt.colorbar(ax.images[0], ax=ax)

    plt.suptitle(title, fontsize=14, fontweight='bold')
    plt.tight_layout()
    plt.show()

# Test activation visualization
X_test = np.random.randn(10, 4)
visualize_activations(nn_vis, X_test, "Activation Flow Through Network")
```

Reflection & Consolidation (30 min)

??? Review forward propagation steps thoroughly ??? Ensure you understand activation functions ??? Write daily reflection (choose 2-3 prompts below) ??? List questions for Wednesday check-in

Daily Reflection Prompts (Choose 2-3):

- What was the most important concept you learned today?
- How do activation functions enable neural networks to learn complex patterns?
- What is the purpose of having multiple layers?
- Why can’t a single-layer network solve XOR?
- What surprised you about forward propagation?
- What questions do you still have about neural networks?

Next: Day 7 - Backpropagation and Training

Week 2, Day 7: Backpropagation and Training

Daily Goals

- Understand backpropagation and the chain rule
- Compute gradients manually for simple networks
- Implement backpropagation from scratch
- Build complete training loop with gradient descent
- Successfully train XOR network
- Visualize learning process

Morning Session (4 hours)

Optional: Daily Check-in with Peers on Teams (15 min)

Video Learning (90 min)

??? **Watch**: What is backpropagation really doing? by 3Blue1Brown (14 min) *THE essential video for understanding backpropagation intuitively*

??? **Watch**: Backpropagation calculus by 3Blue1Brown (10 min) *Mathematical details - watch after the intuition video*

??? **Watch**: Backpropagation main ideas by StatQuest (14 min) *Different perspective, reinforces concepts*

??? **Watch**: Chain Rule by StatQuest (18 min) *Foundation for backpropagation mathematics*

??? **Watch**: Backpropagation Details Pt 1 by StatQuest (13 min)

??? **Watch**: Backpropagation Details Pt 2 by StatQuest (11 min)

Reference Material (30 min)

??? **Read**: D2L Chapter 5.3 - Forward and Backward Propagation

??? **Optional**: Michael Nielsen Chapter 2 - Backpropagation details

Hands-on Coding - Part 1 (2 hours)

Exercise 1: Manual Gradient Calculation (45 min) Work through backpropagation by hand to build intuition:

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

# Activation functions from Day 6
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def sigmoid_derivative(x):
    s = sigmoid(x)
    return s * (1 - s)

print("="*70)
print("MANUAL BACKPROPAGATION CALCULATION")
print("="*70)

# Tiny network: 1 input -> 1 hidden -> 1 output
```

```python
print("\nNetwork: 1 input ??? 1 hidden neuron ??? 1 output")

# Example data
x = 0.5
y_true = 0.8

# Weights
w1 = 0.4
b1 = 0.1
w2 = 0.6
b2 = 0.2

print(f"\nInput: x = {x}")
print(f"Target: y = {y_true}")
print(f"\nWeights: w1={w1}, b1={b1}, w2={w2}, b2={b2}")

# Forward pass
print("\n" + "="*70)
print("FORWARD PASS")
print("="*70)

z1 = w1 * x + b1
a1 = sigmoid(z1)
print(f"\nHidden layer:")
print(f"  z1 = w1*x + b1 = {w1}*{x} + {b1} = {z1}")
print(f"  a1 = sigmoid(z1) = {a1:.4f}")

z2 = w2 * a1 + b2
a2 = sigmoid(z2)
print(f"\nOutput layer:")
print(f"  z2 = w2*a1 + b2 = {w2}*{a1:.4f} + {b2} = {z2:.4f}")
print(f"  a2 = sigmoid(z2) = {a2:.4f}")

# Loss (MSE)
loss = 0.5 * (y_true - a2)**2
print(f"\nLoss = 0.5*(y_true - a2)?? = 0.5*({y_true} - {a2:.4f})?? = {loss:.4f}")

# Backward pass
print("\n" + "="*70)
print("BACKWARD PASS (Computing Gradients)")
print("="*70)

# Output layer gradients
print("\nOutput layer:")
dL_da2 = -(y_true - a2)
print(f"  ???L/???a2 = -(y_true - a2) = {dL_da2:.4f}")

da2_dz2 = sigmoid_derivative(z2)
print(f"  ???a2/???z2 = sigmoid'(z2) = {da2_dz2:.4f}")

dL_dz2 = dL_da2 * da2_dz2  # Chain rule!
print(f"  ???L/???z2 = ???L/???a2 * ???a2/???z2 = {dL_dz2:.4f}")

dL_dw2 = dL_dz2 * a1
print(f"  ???L/???w2 = ???L/???z2 * a1 = {dL_dw2:.4f}")
```

```
dL_db2 = dL_dz2
print(f"  ???L/???b2 = ???L/???z2 = {dL_db2:.4f}")

# Hidden layer gradients
print("\nHidden layer:")
dL_da1 = dL_dz2 * w2
print(f"  ???L/???a1 = ???L/???z2 * w2 = {dL_da1:.4f}")

da1_dz1 = sigmoid_derivative(z1)
print(f"  ???a1/???z1 = sigmoid'(z1) = {da1_dz1:.4f}")

dL_dz1 = dL_da1 * da1_dz1  # Chain rule again!
print(f"  ???L/???z1 = ???L/???a1 * ???a1/???z1 = {dL_dz1:.4f}")

dL_dw1 = dL_dz1 * x
print(f"  ???L/???w1 = ???L/???z1 * x = {dL_dw1:.4f}")

dL_db1 = dL_dz1
print(f"  ???L/???b1 = ???L/???z1 = {dL_db1:.4f}")

print("\n" + "="*70)
print("GRADIENT SUMMARY")
print("="*70)
print(f"???L/???w2 = {dL_dw2:.6f}")
print(f"???L/???b2 = {dL_db2:.6f}")
print(f"???L/???w1 = {dL_dw1:.6f}")
print(f"???L/???b1 = {dL_db1:.6f}")

# Update weights with gradient descent
learning_rate = 0.5
print(f"\n" + "="*70)
print(f"WEIGHT UPDATE (learning_rate = {learning_rate})")
print("="*70)

w2_new = w2 - learning_rate * dL_dw2
b2_new = b2 - learning_rate * dL_db2
w1_new = w1 - learning_rate * dL_dw1
b1_new = b1 - learning_rate * dL_db1

print(f"w2: {w2:.4f} ??? {w2_new:.4f} (change: {w2_new - w2:.4f})")
print(f"b2: {b2:.4f} ??? {b2_new:.4f} (change: {b2_new - b2:.4f})")
print(f"w1: {w1:.4f} ??? {w1_new:.4f} (change: {w1_new - w1:.4f})")
print(f"b1: {b1:.4f} ??? {b1_new:.4f} (change: {b1_new - b1:.4f})")

# Verify with new forward pass
z1_new = w1_new * x + b1_new
a1_new = sigmoid(z1_new)
z2_new = w2_new * a1_new + b2_new
a2_new = sigmoid(z2_new)
loss_new = 0.5 * (y_true - a2_new)**2

print(f"\nPrediction: {a2:.4f} ??? {a2_new:.4f} (closer to {y_true})")
print(f"Loss: {loss:.4f} ??? {loss_new:.4f} (decreased by {loss - loss_new:.4f})")
print("\n??? Backpropagation worked! Loss decreased.")
```

Exercise 2: Implement Backpropagation in Neural Network Class (75 min) Add backpropagation to yesterday's NeuralNetwork class:

```python
class NeuralNetwork:
    def __init__(self, input_size, hidden_size, output_size, learning_rate=0.1):
        # Initialize weights
        self.W1 = np.random.randn(input_size, hidden_size) * 0.01
        self.b1 = np.zeros((1, hidden_size))
        self.W2 = np.random.randn(hidden_size, output_size) * 0.01
        self.b2 = np.zeros((1, output_size))

        self.learning_rate = learning_rate

    def forward(self, X):
        """Forward propagation"""
        self.z1 = np.dot(X, self.W1) + self.b1
        self.a1 = sigmoid(self.z1)
        self.z2 = np.dot(self.a1, self.W2) + self.b2
        self.a2 = sigmoid(self.z2)
        return self.a2

    def backward(self, X, y):
        """
        Backpropagation - compute gradients

        Args:
            X: inputs (n_samples, n_features)
            y: targets (n_samples, n_outputs)
        """
        m = X.shape[0]  # number of samples

        # Output layer gradients
        dz2 = self.a2 - y  # derivative of sigmoid + MSE
        dW2 = (1/m) * np.dot(self.a1.T, dz2)
        db2 = (1/m) * np.sum(dz2, axis=0, keepdims=True)

        # Hidden layer gradients
        da1 = np.dot(dz2, self.W2.T)
        dz1 = da1 * sigmoid_derivative(self.z1)
        dW1 = (1/m) * np.dot(X.T, dz1)
        db1 = (1/m) * np.sum(dz1, axis=0, keepdims=True)

        # Store gradients
        self.gradients = {
            'dW2': dW2, 'db2': db2,
            'dW1': dW1, 'db1': db1
        }

        return self.gradients

    def update_weights(self):
        """Update weights using computed gradients"""
        self.W2 -= self.learning_rate * self.gradients['dW2']
        self.b2 -= self.learning_rate * self.gradients['db2']
        self.W1 -= self.learning_rate * self.gradients['dW1']
        self.b1 -= self.learning_rate * self.gradients['db1']
    def compute_loss(self, y_true, y_pred):
        """Mean Squared Error loss"""
        return np.mean((y_true - y_pred) ** 2)

    def train_step(self, X, y):
        """Single training step: forward, backward, update"""
        # Forward
        y_pred = self.forward(X)

        # Compute loss
        loss = self.compute_loss(y, y_pred)

        # Backward
        self.backward(X, y)

        # Update
        self.update_weights()

        return loss

# Test the implementation
print("\n" + "="*70)
print("TESTING BACKPROPAGATION IMPLEMENTATION")
print("="*70)

# Simple test data
X_test = np.array([[0.5]])
y_test = np.array([[0.8]])

nn = NeuralNetwork(input_size=1, hidden_size=2, output_size=1, learning_rate=0.5)

print("\nTraining for 10 steps on single sample:")
for step in range(10):
    loss = nn.train_step(X_test, y_test)
    pred = nn.forward(X_test)[0, 0]
    print(f"Step {step+1}: Loss = {loss:.6f}, Prediction = {pred:.4f}")

print(f"\nTarget: {y_test[0,0]}")
print(f"Final prediction: {pred:.4f}")
print("??? Network is learning!" if loss < 0.01 else "?????? May need more training")
```

Afternoon Session (4 hours)

Hands-on Coding - Part 2 (3.5 hours)

Exercise 3: Train XOR Network (60 min) Finally solve the XOR problem!

```
print("="*70)
print("TRAINING NEURAL NETWORK ON XOR")
print("="*70)

# XOR data
X_xor = np.array([[0, 0],
                  [0, 1],
                  [1, 0],
                  [1, 1]])
```

```python
y_xor = np.array([[0],
                  [1],
                  [1],
                  [0]])

# Create network
nn_xor = NeuralNetwork(input_size=2, hidden_size=4, output_size=1, learning_rate=0.5)

# Training loop
epochs = 5000
losses = []
predictions_history = []

print(f"\nTraining for {epochs} epochs...")
for epoch in range(epochs):
    loss = nn_xor.train_step(X_xor, y_xor)
    losses.append(loss)

    if (epoch + 1) % 500 == 0:
        preds = nn_xor.forward(X_xor)
        print(f"Epoch {epoch+1:5d}: Loss = {loss:.6f}")
        predictions_history.append(preds.copy())

# Final results
print("\n" + "="*70)
print("FINAL RESULTS")
print("="*70)

predictions = nn_xor.forward(X_xor)
print("\nInput | Target | Prediction | Correct?")
print("------|--------|------------|----------")
for x, y_true, y_pred in zip(X_xor, y_xor, predictions):
    correct = "???" if abs(y_true[0] - y_pred[0]) < 0.1 else "???"
    print(f"{x}  |   {y_true[0]}    |   {y_pred[0]:.4f}   |    {correct}")

# Visualizations
fig, axes = plt.subplots(2, 2, figsize=(14, 12))

# Loss curve
axes[0, 0].plot(losses)
axes[0, 0].set_xlabel('Epoch')
axes[0, 0].set_ylabel('Loss')
axes[0, 0].set_title('Training Loss')
axes[0, 0].grid(True, alpha=0.3)
axes[0, 0].set_yscale('log')

# Decision boundary
h = 0.01
x_min, x_max = -0.5, 1.5
y_min, y_max = -0.5, 1.5
xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                     np.arange(y_min, y_max, h))
Z = nn_xor.forward(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)

axes[0, 1].contourf(xx, yy, Z, alpha=0.4, cmap='RdYlBu', levels=20)
```

```
axes[0, 1].scatter(X_xor[y_xor.flatten()==0, 0], X_xor[y_xor.flatten()==0, 1],
                  c='red', s=200, edgecolors='k', linewidth=2, label='Class 0')
axes[0, 1].scatter(X_xor[y_xor.flatten()==1, 0], X_xor[y_xor.flatten()==1, 1],
                  c='blue', s=200, edgecolors='k', linewidth=2, label='Class 1')
axes[0, 1].set_xlabel('Input 1')
axes[0, 1].set_ylabel('Input 2')
axes[0, 1].set_title('Learned Decision Boundary')
axes[0, 1].legend()
axes[0, 1].grid(True, alpha=0.3)

# Weight matrices
im = axes[1, 0].imshow(nn_xor.W1, cmap='RdBu', aspect='auto', vmin=-5, vmax=5)
axes[1, 0].set_title('W1: Input ??? Hidden Weights')
axes[1, 0].set_xlabel('Hidden Neurons')
axes[1, 0].set_ylabel('Input Features')
plt.colorbar(im, ax=axes[1, 0])

im = axes[1, 1].imshow(nn_xor.W2, cmap='RdBu', aspect='auto', vmin=-5, vmax=5)
axes[1, 1].set_title('W2: Hidden ??? Output Weights')
axes[1, 1].set_xlabel('Output')
axes[1, 1].set_ylabel('Hidden Neurons')
plt.colorbar(im, ax=axes[1, 1])

plt.suptitle('XOR Problem: Successfully Learned!', fontsize=14, fontweight='bold')
plt.tight_layout()
plt.show()

print("\n???? Congratulations! You've solved XOR with backpropagation!")
```

Exercise 4: Spiral Dataset Classification (70 min) More challenging non-linear problem:

```
def make_spiral_data(n_samples=300, noise=0.1):
    """Generate spiral dataset"""
    n = n_samples // 2

    # Generate spirals
    theta = np.linspace(0, 4*np.pi, n)

    # Class 0
    r0 = theta / (2*np.pi)
    X0 = np.column_stack([r0 * np.cos(theta) + np.random.randn(n) * noise,
                          r0 * np.sin(theta) + np.random.randn(n) * noise])

    # Class 1
    theta += np.pi
    r1 = theta / (2*np.pi)
    X1 = np.column_stack([r1 * np.cos(theta) + np.random.randn(n) * noise,
                          r1 * np.sin(theta) + np.random.randn(n) * noise])

    X = np.vstack([X0, X1])
    y = np.hstack([np.zeros(n), np.ones(n)]).reshape(-1, 1)

    return X, y

# Generate data
X_spiral, y_spiral = make_spiral_data(n_samples=300, noise=0.2)
```

```python
# Visualize
plt.figure(figsize=(8, 6))
plt.scatter(X_spiral[y_spiral.flatten()==0, 0], X_spiral[y_spiral.flatten()==0, 1],
           c='red', edgecolors='k', alpha=0.6, label='Class 0')
plt.scatter(X_spiral[y_spiral.flatten()==1, 0], X_spiral[y_spiral.flatten()==1, 1],
           c='blue', edgecolors='k', alpha=0.6, label='Class 1')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.title('Spiral Dataset')
plt.legend()
plt.grid(True, alpha=0.3)
plt.show()

# Train network
print("Training on spiral dataset...")
nn_spiral = NeuralNetwork(input_size=2, hidden_size=20, output_size=1, learning_rate=0.3)

epochs = 10000
losses_spiral = []

for epoch in range(epochs):
    loss = nn_spiral.train_step(X_spiral, y_spiral)
    losses_spiral.append(loss)

    if (epoch + 1) % 1000 == 0:
        accuracy = np.mean((nn_spiral.forward(X_spiral) > 0.5) == y_spiral)
        print(f"Epoch {epoch+1:5d}: Loss = {loss:.6f}, Accuracy = {accuracy:.4f}")

# Final visualization
h = 0.02
x_min, x_max = X_spiral[:, 0].min() - 0.5, X_spiral[:, 0].max() + 0.5
y_min, y_max = X_spiral[:, 1].min() - 0.5, X_spiral[:, 1].max() + 0.5
xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                     np.arange(y_min, y_max, h))

Z = nn_spiral.forward(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)

plt.figure(figsize=(14, 5))

plt.subplot(1, 2, 1)
plt.plot(losses_spiral)
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training Loss')
plt.grid(True, alpha=0.3)
plt.yscale('log')

plt.subplot(1, 2, 2)
plt.contourf(xx, yy, Z, alpha=0.4, cmap='RdYlBu', levels=20)
plt.scatter(X_spiral[y_spiral.flatten()==0, 0], X_spiral[y_spiral.flatten()==0, 1],
           c='red', edgecolors='k', alpha=0.7, label='Class 0')
plt.scatter(X_spiral[y_spiral.flatten()==1, 0], X_spiral[y_spiral.flatten()==1, 1],
           c='blue', edgecolors='k', alpha=0.7, label='Class 1')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.title('Learned Decision Boundary')
plt.legend()
plt.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

final_accuracy = np.mean((nn_spiral.forward(X_spiral) > 0.5) == y_spiral)
print(f"\n???? Final Accuracy: {final_accuracy:.2%}")
```

Mini-Challenge: Hyperparameter Exploration (50 min) Explore how different settings affect learning:

```python
# Test different configurations
configs = [
    {'hidden_size': 2, 'learning_rate': 0.1, 'name': 'Small network, slow learning'},
    {'hidden_size': 2, 'learning_rate': 1.0, 'name': 'Small network, fast learning'},
    {'hidden_size': 10, 'learning_rate': 0.1, 'name': 'Large network, slow learning'},
    {'hidden_size': 10, 'learning_rate': 1.0, 'name': 'Large network, fast learning'},
]

fig, axes = plt.subplots(2, 2, figsize=(14, 12))
axes = axes.flatten()

for idx, config in enumerate(configs):
    # Train network
    nn = NeuralNetwork(input_size=2, hidden_size=config['hidden_size'],
                       output_size=1, learning_rate=config['learning_rate'])

    losses = []
    for epoch in range(2000):
        loss = nn.train_step(X_xor, y_xor)
        losses.append(loss)

    # Plot
    axes[idx].plot(losses)
    axes[idx].set_xlabel('Epoch')
    axes[idx].set_ylabel('Loss')
    axes[idx].set_title(config['name'])
    axes[idx].grid(True, alpha=0.3)
    axes[idx].set_yscale('log')

    final_loss = losses[-1]
    axes[idx].text(0.7, 0.9, f'Final: {final_loss:.4f}',
                   transform=axes[idx].transAxes,
                   bbox=dict(boxstyle='round', facecolor='wheat'))

plt.suptitle('Hyperparameter Effects on XOR Learning', fontsize=14, fontweight='bold')
plt.tight_layout()
plt.show()

print("\nKey observations:")
print("- Too small network may not have enough capacity")
print("- Too large learning rate can cause instability")
print("- Balance between network size and learning rate matters")
```

Reflection & Consolidation (30 min)

??? Review backpropagation algorithm thoroughly ??? Ensure you understand the chain rule application ??? Write daily reflection (choose 2-3 prompts below) ??? Prepare questions for Friday check-in

Daily Reflection Prompts (Choose 2-3):

- What was the most important concept you learned today?
- How does backpropagation enable neural networks to learn?
- What is the role of the chain rule in computing gradients?
- What surprised you about training neural networks?
- How did solving XOR feel compared to yesterday's forward propagation?
- What challenges did you face in implementing backpropagation?
- What questions do you still have about gradient descent?

Next: Day 8 - Introduction to PyTorch

Week 2, Day 8: Introduction to PyTorch

Daily Goals

- Understand PyTorch tensors and operations
- Learn automatic differentiation with autograd
- Build neural networks using nn.Module
- Recreate Day 6's network in PyTorch
- Compare NumPy vs PyTorch implementations

Morning Session (4 hours)

Optional: Daily Check-in with Peers on Teams (15 min)

Video Learning (90 min)

??? **Watch**: PyTorch in 100 Seconds by Fireship (3 min) *Quick overview of PyTorch*

??? **Watch**: PyTorch Tutorial - Neural Networks by freeCodeCamp (25 min) *Comprehensive introduction to PyTorch basics*

??? **Watch**: PyTorch Tensors by Aladdin Persson (15 min) *Deep dive into tensor operations*

??? **Watch**: PyTorch Autograd by Aladdin Persson (10 min) *Understanding automatic differentiation*

??? **Watch**: Building Neural Networks in PyTorch by Python Engineer (20 min) *How to use nn.Module*

Reference Material (30 min)

??? **Read**: PyTorch Quickstart

??? **Read**: Tensors Tutorial

??? **Read**: Autograd Tutorial

Hands-on Coding - Part 1 (2 hours)

Setup (10 min)

```python
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
import matplotlib.pyplot as plt

print(f"PyTorch version: {torch.__version__}")
print(f"CUDA available: {torch.cuda.is_available()}")

# Set random seed for reproducibility
torch.manual_seed(42)
np.random.seed(42)
```

Exercise 1: PyTorch Tensors - Matching Day 1's NumPy (45 min) Learn PyTorch tensors by comparing with NumPy:

```python
print("="*70)
print("PYTORCH TENSORS vs NUMPY ARRAYS")
print("="*70)
```

```python
# Creating tensors
print("\n1. Creating tensors")
print("-" * 40)

# NumPy way
np_array = np.array([[1, 2, 3], [4, 5, 6]])
print(f"NumPy array:\n{np_array}")

# PyTorch way
torch_tensor = torch.tensor([[1, 2, 3], [4, 5, 6]])
print(f"\nPyTorch tensor:\n{torch_tensor}")

# From NumPy
torch_from_np = torch.from_numpy(np_array)
print(f"\nFrom NumPy:\n{torch_from_np}")

# To NumPy
np_from_torch = torch_tensor.numpy()
print(f"\nTo NumPy:\n{np_from_torch}")

# Different creation methods
print("\n2. Initialization methods")
print("-" * 40)

zeros_np = np.zeros((2, 3))
zeros_torch = torch.zeros(2, 3)
print(f"Zeros - NumPy:\n{zeros_np}")
print(f"Zeros - PyTorch:\n{zeros_torch}")

ones_torch = torch.ones(2, 3)
random_torch = torch.randn(2, 3)  # Normal distribution
print(f"\nOnes:\n{ones_torch}")
print(f"\nRandom:\n{random_torch}")

# Operations
print("\n3. Basic operations")
print("-" * 40)

a = torch.tensor([[1.0, 2.0], [3.0, 4.0]])
b = torch.tensor([[5.0, 6.0], [7.0, 8.0]])

print(f"a:\n{a}")
print(f"\nb:\n{b}")
print(f"\na + b:\n{a + b}")
print(f"\na * b (element-wise):\n{a * b}")
print(f"\na @ b (matrix multiply):\n{a @ b}")

# More operations
print(f"\nMean: {a.mean()}")
print(f"Sum: {a.sum()}")
print(f"Max: {a.max()}")
print(f"Transpose:\n{a.t()}")

# Reshaping
print("\n4. Reshaping")
print("-" * 40)
```

```
x = torch.arange(12)
print(f"Original: {x}")
print(f"Shape: {x.shape}")

x_reshaped = x.view(3, 4)
print(f"\nReshaped (3, 4):\n{x_reshaped}")

x_reshaped2 = x.view(2, 6)
print(f"\nReshaped (2, 6):\n{x_reshaped2}")

# Indexing (similar to NumPy)
print("\n5. Indexing and slicing")
print("-" * 40)

matrix = torch.arange(20).view(4, 5)
print(f"Matrix:\n{matrix}")
print(f"\nFirst row: {matrix[0]}")
print(f"First column: {matrix[:, 0]}")
print(f"Submatrix:\n{matrix[1:3, 2:4]}")

# Key difference: requires_grad
print("\n6. Gradient tracking")
print("-" * 40)

x = torch.tensor([2.0, 3.0], requires_grad=True)
print(f"Tensor with gradient tracking: {x}")
print(f"requires_grad: {x.requires_grad}")
```

Exercise 2: Automatic Differentiation with Autograd (45 min) Understand PyTorch's autograd system:

```
print("\n" + "="*70)
print("AUTOMATIC DIFFERENTIATION - The Magic of PyTorch")
print("="*70)

# Simple example
print("\n1. Basic autograd example")
print("-" * 40)

x = torch.tensor(2.0, requires_grad=True)
y = x ** 2 + 3 * x + 1

print(f"x = {x.item()}")
print(f"y = x?? + 3x + 1 = {y.item()}")

# Compute gradient
y.backward()  # dy/dx
print(f"\ndy/dx = 2x + 3")
print(f"At x={x.item()}: dy/dx = {x.grad.item()}")
print(f"Expected: 2*{x.item()} + 3 = {2*x.item() + 3}")

# More complex example
print("\n2. Neural network-like computation")
print("-" * 40)

x = torch.tensor([[1.0, 2.0]], requires_grad=True)
```

```
w = torch.tensor([[0.5], [0.3]], requires_grad=True)
b = torch.tensor([[0.1]], requires_grad=True)

# Forward pass
z = x @ w + b  # Linear layer
a = torch.sigmoid(z)  # Activation
loss = (a - 1.0) ** 2  # Simple loss

print(f"Input: {x}")
print(f"Weights: {w.t()}")
print(f"Bias: {b}")
print(f"Output: {a.item():.4f}")
print(f"Loss: {loss.item():.4f}")

# Backward pass
loss.backward()

print(f"\nGradients:")
print(f"???loss/???w:\n{w.grad}")
print(f"???loss/???b: {b.grad}")
print(f"???loss/???x: {x.grad}")

# Manual gradient descent
print("\n3. Manual weight update")
print("-" * 40)

learning_rate = 0.1

with torch.no_grad():  # Don't track these operations
    w -= learning_rate * w.grad
    b -= learning_rate * b.grad

    # Zero gradients for next iteration
    w.grad.zero_()
    b.grad.zero_()

print(f"Updated weights: {w.t()}")
print(f"Updated bias: {b}")

# Visualize gradient flow
print("\n4. Computational graph visualization")
print("-" * 40)

x = torch.tensor(3.0, requires_grad=True)
a = x * 2
b = a * 3
c = b ** 2
c.backward()

print(f"x = {x.item()}")
print(f"a = x * 2 = {a.item()}")
print(f"b = a * 3 = {b.item()}")
print(f"c = b?? = {c.item()}")
print(f"\ndc/dx = {x.grad.item()}")
print(f"Expected: dc/dx = 2b * 3 * 2 = {2*b.item()*3*2}")
```

Exercise 3: Building with nn.Module (30 min) Learn PyTorch's way of building networks:

```python
print("\n" + "="*70)
print("BUILDING NETWORKS WITH nn.Module")
print("="*70)

# Simple network class
class SimpleNetwork(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(SimpleNetwork, self).__init__()

        # Define layers
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.fc2 = nn.Linear(hidden_size, output_size)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        # Define forward pass
        x = self.fc1(x)
        x = self.sigmoid(x)
        x = self.fc2(x)
        x = self.sigmoid(x)
        return x

# Create network
model = SimpleNetwork(input_size=2, hidden_size=4, output_size=1)

print("Network architecture:")
print(model)

print("\nParameters:")
for name, param in model.named_parameters():
    print(f"{name}: {param.shape}")

# Count parameters
total_params = sum(p.numel() for p in model.parameters())
print(f"\nTotal parameters: {total_params}")

# Test forward pass
x_test = torch.tensor([[0.5, 0.8]])
output = model(x_test)
print(f"\nTest input: {x_test}")
print(f"Output: {output.item():.4f}")
```

Afternoon Session (4 hours)

Hands-on Coding - Part 2 (3.5 hours)

Exercise 4: Recreate Day 6's Network in PyTorch (60 min) Build the same network from Day 6, but in PyTorch:

```python
print("="*70)
print("IMPLEMENTING DAY 6's NETWORK IN PYTORCH")
print("="*70)

# XOR data in PyTorch
```

```python
X_xor = torch.tensor([[0, 0],
                      [0, 1],
                      [1, 0],
                      [1, 1]], dtype=torch.float32)
y_xor = torch.tensor([[0],
                      [1],
                      [1],
                      [0]], dtype=torch.float32)

# Network definition
class XORNetwork(nn.Module):
    def __init__(self):
        super(XORNetwork, self).__init__()
        self.fc1 = nn.Linear(2, 4)  # 2 inputs, 4 hidden
        self.fc2 = nn.Linear(4, 1)  # 4 hidden, 1 output

    def forward(self, x):
        x = torch.sigmoid(self.fc1(x))
        x = torch.sigmoid(self.fc2(x))
        return x

# Create model, loss function, optimizer
model = XORNetwork()
criterion = nn.MSELoss()
optimizer = optim.SGD(model.parameters(), lr=0.5)

print("Training XOR network in PyTorch...")
print(f"Model: {model}")

# Training loop
epochs = 5000
losses = []

for epoch in range(epochs):
    # Forward pass
    outputs = model(X_xor)
    loss = criterion(outputs, y_xor)

    # Backward pass
    optimizer.zero_grad()  # Clear gradients
    loss.backward()         # Compute gradients
    optimizer.step()        # Update weights

    losses.append(loss.item())

    if (epoch + 1) % 500 == 0:
        print(f"Epoch {epoch+1:5d}: Loss = {loss.item():.6f}")

# Test the model
print("\n" + "="*70)
print("RESULTS")
print("="*70)

with torch.no_grad():  # Don't track gradients during inference
    predictions = model(X_xor)
```

```
print("\nInput | Target | Prediction | Correct?")
print("------|--------|------------|----------")
for x, y_true, y_pred in zip(X_xor, y_xor, predictions):
    correct = "???" if abs(y_true.item() - y_pred.item()) < 0.1 else "???"
    print(f"{x.numpy()}  |   {y_true.item()}    |   {y_pred.item():.4f}   |    {correct}")

# Visualize
plt.figure(figsize=(12, 4))

plt.subplot(1, 2, 1)
plt.plot(losses)
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('PyTorch Training Loss')
plt.yscale('log')
plt.grid(True, alpha=0.3)

# Decision boundary
h = 0.01
x_min, x_max = -0.5, 1.5
y_min, y_max = -0.5, 1.5
xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                     np.arange(y_min, y_max, h))

grid = torch.tensor(np.c_[xx.ravel(), yy.ravel()], dtype=torch.float32)
with torch.no_grad():
    Z = model(grid).numpy()
Z = Z.reshape(xx.shape)

plt.subplot(1, 2, 2)
plt.contourf(xx, yy, Z, alpha=0.4, cmap='RdYlBu', levels=20)
plt.scatter(X_xor[y_xor.flatten()==0, 0], X_xor[y_xor.flatten()==0, 1],
           c='red', s=200, edgecolors='k', linewidth=2)
plt.scatter(X_xor[y_xor.flatten()==1, 0], X_xor[y_xor.flatten()==1, 1],
           c='blue', s=200, edgecolors='k', linewidth=2)
plt.xlabel('Input 1')
plt.ylabel('Input 2')
plt.title('PyTorch Decision Boundary')
plt.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

print("\n??? PyTorch implementation complete!")
```

Exercise 5: Compare NumPy vs PyTorch (50 min) Side-by-side comparison of implementations:

```
print("="*70)
print("NUMPY vs PYTORCH COMPARISON")
print("="*70)

# Let's compare training time and ease of use
import time

# NumPy implementation (from Day 7)
class NumpyNN:
    def __init__(self):
        self.W1 = np.random.randn(2, 4) * 0.01
        self.b1 = np.zeros((1, 4))
        self.W2 = np.random.randn(4, 1) * 0.01
        self.b2 = np.zeros((1, 1))
        self.lr = 0.5

    def sigmoid(self, x):
        return 1 / (1 + np.exp(-x))

    def forward(self, X):
        self.z1 = np.dot(X, self.W1) + self.b1
        self.a1 = self.sigmoid(self.z1)
        self.z2 = np.dot(self.a1, self.W2) + self.b2
        self.a2 = self.sigmoid(self.z2)
        return self.a2

    def backward(self, X, y):
        m = X.shape[0]
        dz2 = self.a2 - y
        dW2 = (1/m) * np.dot(self.a1.T, dz2)
        db2 = (1/m) * np.sum(dz2, axis=0, keepdims=True)

        da1 = np.dot(dz2, self.W2.T)
        dz1 = da1 * self.a1 * (1 - self.a1)
        dW1 = (1/m) * np.dot(X.T, dz1)
        db1 = (1/m) * np.sum(dz1, axis=0, keepdims=True)

        self.W2 -= self.lr * dW2
        self.b2 -= self.lr * db2
        self.W1 -= self.lr * dW1
        self.b1 -= self.lr * db1

# Training comparison
X_np = X_xor.numpy()
y_np = y_xor.numpy()

# NumPy
print("\nTraining with NumPy...")
np_nn = NumpyNN()
start = time.time()
for _ in range(5000):
    np_nn.forward(X_np)
    np_nn.backward(X_np, y_np)
numpy_time = time.time() - start

# PyTorch
print("Training with PyTorch...")
torch_nn = XORNetwork()
criterion = nn.MSELoss()
optimizer = optim.SGD(torch_nn.parameters(), lr=0.5)

start = time.time()
for _ in range(5000):
    outputs = torch_nn(X_xor)
    loss = criterion(outputs, y_xor)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
pytorch_time = time.time() - start

print("\n" + "="*70)
print("COMPARISON RESULTS")
print("="*70)

print(f"\nTraining Time:")
print(f"NumPy:   {numpy_time:.3f} seconds")
print(f"PyTorch: {pytorch_time:.3f} seconds")
print(f"Speedup: {numpy_time/pytorch_time:.2f}x")

print(f"\nCode Complexity:")
print(f"NumPy:   ~50 lines (manual backprop)")
print(f"PyTorch: ~15 lines (automatic backprop)")

print(f"\nAdvantages:")
print("\nNumPy:")
print("  + Full control over implementation")
print("  + Educational - see every detail")
print("  + No dependencies beyond NumPy")
print("  - Manual gradient computation")
print("  - Error-prone")
print("  - No GPU support")

print("\nPyTorch:")
print("  + Automatic differentiation")
print("  + Less code, fewer bugs")
print("  + GPU acceleration available")
print("  + Production-ready")
print("  + Large ecosystem")
print("  - Abstraction hides details")

print("\n???? Recommendation: Learn with NumPy, build with PyTorch!")
```

Mini-Challenge: Advanced PyTorch Features (70 min) Explore more PyTorch capabilities:

```
print("="*70)
print("ADVANCED PYTORCH FEATURES")
print("="*70)

# 1. Different activation functions
print("\n1. Exploring activation functions")
print("-" * 40)

class FlexibleNetwork(nn.Module):
    def __init__(self, activation='relu'):
        super(FlexibleNetwork, self).__init__()
        self.fc1 = nn.Linear(2, 8)
        self.fc2 = nn.Linear(8, 1)

        # Choose activation
        if activation == 'relu':
            self.activation = nn.ReLU()
        elif activation == 'tanh':
            self.activation = nn.Tanh()
        elif activation == 'sigmoid':
            self.activation = nn.Sigmoid()

    def forward(self, x):
        x = self.fc1(x)
        x = self.activation(x)
        x = self.fc2(x)
        x = torch.sigmoid(x)
        return x

# Compare activations
activations = ['relu', 'tanh', 'sigmoid']
results = {}

for act in activations:
    model = FlexibleNetwork(activation=act)
    criterion = nn.MSELoss()
    optimizer = optim.Adam(model.parameters(), lr=0.01)

    losses = []
    for epoch in range(2000):
        outputs = model(X_xor)
        loss = criterion(outputs, y_xor)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        losses.append(loss.item())

    results[act] = losses
    print(f"{act:10s}: Final loss = {losses[-1]:.6f}")

# Plot comparison
plt.figure(figsize=(10, 6))
for act, losses in results.items():
    plt.plot(losses, label=act.upper())
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Activation Function Comparison')
plt.legend()
plt.yscale('log')
plt.grid(True, alpha=0.3)
plt.show()

# 2. Different optimizers
print("\n2. Comparing optimizers")
print("-" * 40)

optimizers_to_test = {
    'SGD': lambda params: optim.SGD(params, lr=0.1),
    'Adam': lambda params: optim.Adam(params, lr=0.01),
    'RMSprop': lambda params: optim.RMSprop(params, lr=0.01),
}
```

```python
optimizer_results = {}

for opt_name, opt_fn in optimizers_to_test.items():
    model = XORNetwork()
    criterion = nn.MSELoss()
    optimizer = opt_fn(model.parameters())

    losses = []
    for epoch in range(1000):
        outputs = model(X_xor)
        loss = criterion(outputs, y_xor)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        losses.append(loss.item())

    optimizer_results[opt_name] = losses
    print(f"{opt_name:10s}: Final loss = {losses[-1]:.6f}")

# Plot
plt.figure(figsize=(10, 6))
for opt_name, losses in optimizer_results.items():
    plt.plot(losses, label=opt_name)
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Optimizer Comparison')
plt.legend()
plt.yscale('log')
plt.grid(True, alpha=0.3)
plt.show()

print("\n??? PyTorch exploration complete!")
```

Reflection & Consolidation (30 min)

??? Review PyTorch fundamentals ??? Understand autograd mechanism ??? Compare with NumPy implementation from Days 6-7 ??? Write daily reflection (choose 2-3 prompts below)

Daily Reflection Prompts (Choose 2-3):

- What was the most important concept you learned today?
- How does PyTorch's autograd compare to manual backpropagation?
- What are the advantages of using PyTorch over NumPy?
- What surprised you about PyTorch?
- How confident do you feel about building networks in PyTorch?
- What questions do you have about PyTorch?

Next: Day 9 - Building Neural Networks in PyTorch

Week 2, Day 9: Building Neural Networks in PyTorch

Daily Goals

- Master PyTorch Dataset and DataLoader
- Implement dropout and batch normalization
- Build proper training/validation pipelines
- Apply early stopping
- Complete Fashion-MNIST classification
- Compare different architectures and hyperparameters

Morning Session (4 hours)

Optional: Daily Check-in with Peers on Teams (15 min)

Video Learning (90 min)

??? **Watch**: Build the Neural Network by PyTorch (20 min) *Official PyTorch tutorial on building networks*

??? **Watch**: PyTorch Dataset and DataLoader by Aladdin Persson (15 min) *How to handle data efficiently*

??? **Watch**: Batch Normalization by StatQuest (8 min) *Understanding batch norm*

??? **Watch**: Dropout by StatQuest (8 min) *Preventing overfitting with dropout*

??? **Watch**: Training Neural Networks review if needed (20 min)

Reference Material (30 min)

??? **Read**: PyTorch Datasets & DataLoaders

??? **Read**: D2L Chapter 8.4 - Batch Normalization

??? **Read**: D2L Chapter 8.5 - Dropout

Hands-on Coding - Part 1 (2 hours)

Exercise 1: Custom Dataset and DataLoader (50 min) Learn to handle data the PyTorch way:

```python
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader
import numpy as np
import matplotlib.pyplot as plt

print("="*70)
print("PYTORCH DATASET AND DATALOADER")
print("="*70)

# Custom Dataset class
class XORDataset(Dataset):
    def __init__(self, n_samples=1000):
        """
        Generate XOR-like data

        Args:
            n_samples: number of samples to generate
        """
        # Generate random points
        X = np.random.randn(n_samples, 2)

        # XOR logic: y = 1 if (x1 > 0) XOR (x2 > 0)
        y = ((X[:, 0] > 0) != (X[:, 1] > 0)).astype(np.float32)

        self.X = torch.tensor(X, dtype=torch.float32)
        self.y = torch.tensor(y, dtype=torch.float32).unsqueeze(1)

    def __len__(self):
        """Return the number of samples"""
        return len(self.X)

    def __getitem__(self, idx):
        """Get a single sample"""
        return self.X[idx], self.y[idx]

# Create dataset
dataset = XORDataset(n_samples=1000)
print(f"Dataset size: {len(dataset)}")

# Access single sample
x_sample, y_sample = dataset[0]
print(f"\nSample 0:")
print(f"  X: {x_sample}")
print(f"  y: {y_sample.item()}")

# Create DataLoader
batch_size = 32
dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True)

print(f"\nDataLoader:")
print(f"  Batch size: {batch_size}")
print(f"  Number of batches: {len(dataloader)}")

# Iterate through batches
print(f"\nFirst batch:")
for X_batch, y_batch in dataloader:
    print(f"  X_batch shape: {X_batch.shape}")
    print(f"  y_batch shape: {y_batch.shape}")
    print(f"  First sample: X={X_batch[0]}, y={y_batch[0].item()}")
    break  # Just show first batch

# Visualize dataset
X_all = dataset.X.numpy()
y_all = dataset.y.numpy().flatten()

plt.figure(figsize=(8, 6))
plt.scatter(X_all[y_all==0, 0], X_all[y_all==0, 1],
           c='red', alpha=0.5, label='Class 0')
plt.scatter(X_all[y_all==1, 0], X_all[y_all==1, 1],
           c='blue', alpha=0.5, label='Class 1')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.title('XOR Dataset')
plt.legend()
plt.grid(True, alpha=0.3)
plt.show()
```

Exercise 2: Dropout and Batch Normalization (60 min) Add regularization to prevent overfitting:

```
print("\n" + "="*70)
print("REGULARIZATION TECHNIQUES")
print("="*70)

# Network with dropout
class NetworkWithDropout(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, dropout_prob=0.5):
        super(NetworkWithDropout, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.dropout1 = nn.Dropout(p=dropout_prob)
        self.fc2 = nn.Linear(hidden_size, hidden_size)
        self.dropout2 = nn.Dropout(p=dropout_prob)
        self.fc3 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = self.dropout1(x)
        x = torch.relu(self.fc2(x))
        x = self.dropout2(x)
        x = torch.sigmoid(self.fc3(x))
        return x

# Network with batch normalization
class NetworkWithBatchNorm(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(NetworkWithBatchNorm, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.bn1 = nn.BatchNorm1d(hidden_size)
        self.fc2 = nn.Linear(hidden_size, hidden_size)
        self.bn2 = nn.BatchNorm1d(hidden_size)
        self.fc3 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = self.fc1(x)
        x = self.bn1(x)
        x = torch.relu(x)
        x = self.fc2(x)
        x = self.bn2(x)
        x = torch.relu(x)
        x = torch.sigmoid(self.fc3(x))
        return x

# Network with both
class NetworkWithBoth(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, dropout_prob=0.5):
        super(NetworkWithBoth, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.bn1 = nn.BatchNorm1d(hidden_size)
        self.dropout1 = nn.Dropout(p=dropout_prob)
        self.fc2 = nn.Linear(hidden_size, hidden_size)
        self.bn2 = nn.BatchNorm1d(hidden_size)
        self.dropout2 = nn.Dropout(p=dropout_prob)
        self.fc3 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = self.fc1(x)
        x = self.bn1(x)
        x = torch.relu(x)
        x = self.dropout1(x)
        x = self.fc2(x)
        x = self.bn2(x)
        x = torch.relu(x)
        x = self.dropout2(x)
        x = torch.sigmoid(self.fc3(x))
        return x

# Compare them
models = {
    'Baseline': nn.Sequential(
        nn.Linear(2, 64),
        nn.ReLU(),
        nn.Linear(64, 64),
        nn.ReLU(),
        nn.Linear(64, 1),
        nn.Sigmoid()
    ),
    'Dropout': NetworkWithDropout(2, 64, 1, dropout_prob=0.3),
    'BatchNorm': NetworkWithBatchNorm(2, 64, 1),
    'Both': NetworkWithBoth(2, 64, 1, dropout_prob=0.3)
}

# Training function
def train_model(model, train_loader, val_loader, epochs=100):
    criterion = nn.BCELoss()
    optimizer = optim.Adam(model.parameters(), lr=0.001)

    train_losses = []
    val_losses = []

    for epoch in range(epochs):
        # Training
        model.train()
        train_loss = 0
        for X_batch, y_batch in train_loader:
            outputs = model(X_batch)
            loss = criterion(outputs, y_batch)

            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            train_loss += loss.item()

        train_losses.append(train_loss / len(train_loader))
        # Validation
        model.eval()
        val_loss = 0
        with torch.no_grad():
            for X_batch, y_batch in val_loader:
                outputs = model(X_batch)
                loss = criterion(outputs, y_batch)
                val_loss += loss.item()

        val_losses.append(val_loss / len(val_loader))

    return train_losses, val_losses

# Split data
train_dataset = XORDataset(n_samples=800)
val_dataset = XORDataset(n_samples=200)

train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=32, shuffle=False)

# Train all models
results = {}
print("\nTraining models...")
for name, model in models.items():
    print(f"\n{name}...")
    train_losses, val_losses = train_model(model, train_loader, val_loader, epochs=100)
    results[name] = {'train': train_losses, 'val': val_losses}
    print(f"  Final train loss: {train_losses[-1]:.4f}")
    print(f"  Final val loss: {val_losses[-1]:.4f}")

# Plot comparison
fig, axes = plt.subplots(2, 2, figsize=(14, 10))
axes = axes.flatten()

for idx, (name, losses) in enumerate(results.items()):
    axes[idx].plot(losses['train'], label='Train', alpha=0.7)
    axes[idx].plot(losses['val'], label='Validation', alpha=0.7)
    axes[idx].set_xlabel('Epoch')
    axes[idx].set_ylabel('Loss')
    axes[idx].set_title(f'{name}')
    axes[idx].legend()
    axes[idx].grid(True, alpha=0.3)

plt.suptitle('Regularization Techniques Comparison', fontsize=14, fontweight='bold')
plt.tight_layout()
plt.show()

print("\n???? Observations:")
print("- Dropout reduces overfitting during training")
print("- Batch norm stabilizes training")
print("- Combining both often works best")
```

Afternoon Session (4 hours)

Hands-on Coding - Part 2 (3.5 hours)

Exercise 3: Complete Training Pipeline with Validation (60 min) Build a production-ready training system:

```
from torch.utils.data import random_split

print("="*70)
print("COMPLETE TRAINING PIPELINE")
print("="*70)

# Create comprehensive training function
def train_with_validation(model, train_loader, val_loader, epochs=100,
                         patience=10, min_delta=0.001):
    """
    Train model with validation and early stopping

    Args:
        model: PyTorch model
        train_loader: training data loader
        val_loader: validation data loader
        epochs: maximum number of epochs
        patience: epochs to wait for improvement
        min_delta: minimum change to qualify as improvement

    Returns:
        history: dictionary with training history
    """
    criterion = nn.BCELoss()
    optimizer = optim.Adam(model.parameters(), lr=0.001)

    history = {
        'train_loss': [],
        'val_loss': [],
        'train_acc': [],
        'val_acc': []
    }

    best_val_loss = float('inf')
    patience_counter = 0

    for epoch in range(epochs):
        # Training phase
        model.train()
        train_loss = 0
        train_correct = 0
        train_total = 0

        for X_batch, y_batch in train_loader:
            # Forward pass
            outputs = model(X_batch)
            loss = criterion(outputs, y_batch)

            # Backward pass
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            # Metrics
            train_loss += loss.item()
            predictions = (outputs > 0.5).float()
            train_correct += (predictions == y_batch).sum().item()
            train_total += y_batch.size(0)

        # Validation phase
        model.eval()
        val_loss = 0
        val_correct = 0
        val_total = 0

        with torch.no_grad():
            for X_batch, y_batch in val_loader:
                outputs = model(X_batch)
                loss = criterion(outputs, y_batch)

                val_loss += loss.item()
                predictions = (outputs > 0.5).float()
                val_correct += (predictions == y_batch).sum().item()
                val_total += y_batch.size(0)

        # Calculate averages
        avg_train_loss = train_loss / len(train_loader)
        avg_val_loss = val_loss / len(val_loader)
        train_acc = train_correct / train_total
        val_acc = val_correct / val_total

        history['train_loss'].append(avg_train_loss)
        history['val_loss'].append(avg_val_loss)
        history['train_acc'].append(train_acc)
        history['val_acc'].append(val_acc)

        # Print progress
        if (epoch + 1) % 10 == 0:
            print(f"Epoch {epoch+1:3d}/{epochs}: "
                  f"Train Loss: {avg_train_loss:.4f}, Val Loss: {avg_val_loss:.4f}, "
                  f"Train Acc: {train_acc:.4f}, Val Acc: {val_acc:.4f}")

        # Early stopping check
        if avg_val_loss < best_val_loss - min_delta:
            best_val_loss = avg_val_loss
            patience_counter = 0
            # Save best model
            best_model_state = model.state_dict().copy()
        else:
            patience_counter += 1

        if patience_counter >= patience:
            print(f"\nEarly stopping at epoch {epoch+1}")
            # Restore best model
            model.load_state_dict(best_model_state)
            break
```

```python
    return history

# Test the pipeline
model = NetworkWithBoth(2, 64, 1, dropout_prob=0.3)

print("Training with early stopping...")
history = train_with_validation(model, train_loader, val_loader,
                                 epochs=200, patience=15)

# Visualize results
fig, axes = plt.subplots(1, 2, figsize=(14, 5))

# Loss
axes[0].plot(history['train_loss'], label='Train Loss', alpha=0.7)
axes[0].plot(history['val_loss'], label='Validation Loss', alpha=0.7)
axes[0].set_xlabel('Epoch')
axes[0].set_ylabel('Loss')
axes[0].set_title('Training History - Loss')
axes[0].legend()
axes[0].grid(True, alpha=0.3)

# Accuracy
axes[1].plot(history['train_acc'], label='Train Accuracy', alpha=0.7)
axes[1].plot(history['val_acc'], label='Validation Accuracy', alpha=0.7)
axes[1].set_xlabel('Epoch')
axes[1].set_ylabel('Accuracy')
axes[1].set_title('Training History - Accuracy')
axes[1].legend()
axes[1].grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

print(f"\nFinal validation accuracy: {history['val_acc'][-1]:.4f}")
```

Exercise 4: Fashion-MNIST Classification (120 min) Apply everything to a real dataset:

```python
from torchvision import datasets, transforms

print("="*70)
print("FASHION-MNIST CLASSIFICATION")
print("="*70)

# Load Fashion-MNIST
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])

train_dataset = datasets.FashionMNIST(root='./data', train=True,
                                      download=True, transform=transform)
test_dataset = datasets.FashionMNIST(root='./data', train=False,
                                     download=True, transform=transform)

# Split train into train/val
train_size = int(0.8 * len(train_dataset))
val_size = len(train_dataset) - train_size
```

```python
train_dataset, val_dataset = random_split(train_dataset, [train_size, val_size])

# Create data loaders
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=64, shuffle=False)
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)

print(f"Training samples: {len(train_dataset)}")
print(f"Validation samples: {len(val_dataset)}")
print(f"Test samples: {len(test_dataset)}")

# Visualize samples
class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',
               'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']

# Get some samples
dataiter = iter(train_loader)
images, labels = next(dataiter)

fig, axes = plt.subplots(2, 5, figsize=(12, 5))
axes = axes.flatten()

for idx in range(10):
    axes[idx].imshow(images[idx].squeeze(), cmap='gray')
    axes[idx].set_title(class_names[labels[idx]])
    axes[idx].axis('off')

plt.suptitle('Fashion-MNIST Samples')
plt.tight_layout()
plt.show()

# Build model
class FashionMNISTNet(nn.Module):
    def __init__(self):
        super(FashionMNISTNet, self).__init__()
        self.flatten = nn.Flatten()
        self.fc1 = nn.Linear(28*28, 256)
        self.bn1 = nn.BatchNorm1d(256)
        self.dropout1 = nn.Dropout(0.3)
        self.fc2 = nn.Linear(256, 128)
        self.bn2 = nn.BatchNorm1d(128)
        self.dropout2 = nn.Dropout(0.3)
        self.fc3 = nn.Linear(128, 10)

    def forward(self, x):
        x = self.flatten(x)
        x = self.fc1(x)
        x = self.bn1(x)
        x = torch.relu(x)
        x = self.dropout1(x)
        x = self.fc2(x)
        x = self.bn2(x)
        x = torch.relu(x)
        x = self.dropout2(x)
        x = self.fc3(x)
        return x
```

```python
model = FashionMNISTNet()
print(f"\nModel architecture:\n{model}")

total_params = sum(p.numel() for p in model.parameters())
print(f"\nTotal parameters: {total_params:,}")

# Train model
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

epochs = 20
history = {'train_loss': [], 'val_loss': [], 'train_acc': [], 'val_acc': []}

print("\nTraining Fashion-MNIST model...")
for epoch in range(epochs):
    # Training
    model.train()
    train_loss = 0
    train_correct = 0
    train_total = 0

    for images, labels in train_loader:
        outputs = model(images)
        loss = criterion(outputs, labels)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        train_loss += loss.item()
        _, predicted = torch.max(outputs.data, 1)
        train_correct += (predicted == labels).sum().item()
        train_total += labels.size(0)

    # Validation
    model.eval()
    val_loss = 0
    val_correct = 0
    val_total = 0

    with torch.no_grad():
        for images, labels in val_loader:
            outputs = model(images)
            loss = criterion(outputs, labels)

            val_loss += loss.item()
            _, predicted = torch.max(outputs.data, 1)
            val_correct += (predicted == labels).sum().item()
            val_total += labels.size(0)

    # Record metrics
    history['train_loss'].append(train_loss / len(train_loader))
    history['val_loss'].append(val_loss / len(val_loader))
    history['train_acc'].append(train_correct / train_total)
    history['val_acc'].append(val_correct / val_total)
    print(f"Epoch {epoch+1:2d}/{epochs}: "
          f"Train Acc: {train_correct/train_total:.4f}, "
          f"Val Acc: {val_correct/val_total:.4f}")

# Test set evaluation
model.eval()
test_correct = 0
test_total = 0

with torch.no_grad():
    for images, labels in test_loader:
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        test_correct += (predicted == labels).sum().item()
        test_total += labels.size(0)

test_accuracy = test_correct / test_total
print(f"\n???? Test Accuracy: {test_accuracy:.4f}")

# Visualize training
fig, axes = plt.subplots(1, 2, figsize=(14, 5))

axes[0].plot(history['train_loss'], label='Train')
axes[0].plot(history['val_loss'], label='Validation')
axes[0].set_xlabel('Epoch')
axes[0].set_ylabel('Loss')
axes[0].set_title('Training Loss')
axes[0].legend()
axes[0].grid(True, alpha=0.3)

axes[1].plot(history['train_acc'], label='Train')
axes[1].plot(history['val_acc'], label='Validation')
axes[1].axhline(y=test_accuracy, color='r', linestyle='--', label='Test')
axes[1].set_xlabel('Epoch')
axes[1].set_ylabel('Accuracy')
axes[1].set_title('Training Accuracy')
axes[1].legend()
axes[1].grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

print("\n??? Fashion-MNIST training complete!")
```

Reflection & Consolidation (30 min)

??? Review dataset/dataloader patterns ??? Understand regularization techniques ??? Reflect on training pipeline design ??? Write daily reflection (choose 2-3 prompts below)

Daily Reflection Prompts (Choose 2-3):

- What was the most important concept you learned today?
- How do DataLoaders improve training efficiency?
- What is the purpose of dropout and batch normalization?

- How does early stopping prevent overfitting?
- What challenges did you face with Fashion-MNIST?
- How ready do you feel for tomorrow’s MNIST project?

Next: Day 10 - MNIST Project

Week 2, Day 10: MNIST Project - Digit Classification

Daily Goals

- Complete end-to-end MNIST digit classification project
- Achieve >95% accuracy (target: >98%)
- Apply all Week 2 concepts in integrated project
- Create professional documentation and visualizations
- Build portfolio-ready project

Morning Session (4 hours)

Optional: Daily Check-in with Peers on Teams (15 min)

Video Learning (45 min)

??? **Watch**: MNIST with PyTorch by Sentdex (20 min) *Complete walkthrough of MNIST project*

??? **Watch**: Saving & Loading Models by Python Engineer (10 min) *How to save and load your trained models*

??? **Optional Review**: Any Week 2 videos as needed (15 min)

Reference Material (30 min)

??? **Read**: PyTorch Save and Load

??? **Review**: D2L Chapter 5 as needed

Project Briefing (30 min)

Read this entire project structure before starting:

```
"""
MNIST DIGIT CLASSIFICATION PROJECT

Goal: Build a neural network that achieves >95% accuracy on MNIST digit classification

Project Structure:
1. Phase 1: Data Exploration (30 min)
2. Phase 2: Baseline Model (45 min)
3. Phase 3: Improved Model (60 min)
4. Phase 4: Analysis & Visualization (45 min)
5. Phase 5: Documentation (30 min)

Total Time: ~3.5 hours

Success Criteria:
- Minimum: >90% test accuracy
- Target: >95% test accuracy
- Stretch: >98% test accuracy
- Code is well-organized and documented
- Visualizations are clear and informative
- Write-up explains methodology and results
"""

import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, random_split
from torchvision import datasets, transforms
import matplotlib.pyplot as plt
import numpy as np
from sklearn.metrics import confusion_matrix, classification_report
import seaborn as sns

# Set random seeds
torch.manual_seed(42)
np.random.seed(42)

print("="*70)
print("MNIST DIGIT CLASSIFICATION PROJECT")
print("Week 2, Day 10 Capstone")
print("="*70)
```

Phase 1: Data Exploration (30 min)

Understand the dataset thoroughly:

```
print("\n" + "="*70)
print("PHASE 1: DATA EXPLORATION")
print("="*70)

# Load MNIST dataset
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))  # MNIST mean and std
])

train_dataset = datasets.MNIST(root='./data', train=True, download=True, transform=transform)
test_dataset = datasets.MNIST(root='./data', train=False, download=True, transform=transform)

print(f"\nDataset Statistics:")
print(f"Training samples: {len(train_dataset)}")
print(f"Test samples: {len(test_dataset)}")
print(f"Image shape: {train_dataset[0][0].shape}")
print(f"Number of classes: {len(train_dataset.classes)}")

# Split training data into train/validation
train_size = int(0.8 * len(train_dataset))
val_size = len(train_dataset) - train_size
train_data, val_data = random_split(train_dataset, [train_size, val_size])

print(f"\nSplit:")
print(f"Training: {len(train_data)}")
print(f"Validation: {len(val_data)}")
print(f"Test: {len(test_dataset)}")

# Visualize samples
fig, axes = plt.subplots(4, 10, figsize=(16, 7))
axes = axes.flatten()
```

```
for i in range(40):
    img, label = train_dataset[i]
    axes[i].imshow(img.squeeze(), cmap='gray')
    axes[i].set_title(f'{label}', fontsize=10)
    axes[i].axis('off')

plt.suptitle('MNIST Dataset Samples', fontsize=14, fontweight='bold')
plt.tight_layout()
plt.show()

# Class distribution
labels = [train_dataset[i][1] for i in range(len(train_dataset))]
unique, counts = np.unique(labels, return_counts=True)

plt.figure(figsize=(10, 6))
plt.bar(unique, counts, color='steelblue', edgecolor='black')
plt.xlabel('Digit Class')
plt.ylabel('Count')
plt.title('Class Distribution in Training Set')
plt.xticks(unique)
plt.grid(True, alpha=0.3, axis='y')
for i, (digit, count) in enumerate(zip(unique, counts)):
    plt.text(digit, count + 50, str(count), ha='center', fontsize=10)
plt.show()

print(f"\nClass distribution:")
for digit, count in zip(unique, counts):
    print(f"  Digit {digit}: {count} samples ({count/len(labels)*100:.1f}%)")

print("\n??? Data exploration complete")
```

Phase 2: Baseline Model (45 min)

Build a simple model to establish baseline:

```
print("\n" + "="*70)
print("PHASE 2: BASELINE MODEL")
print("="*70)

# Simple baseline network
class BaselineNet(nn.Module):
    def __init__(self):
        super(BaselineNet, self).__init__()
        self.flatten = nn.Flatten()
        self.fc1 = nn.Linear(28*28, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = self.flatten(x)
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)
        return x

# Create data loaders
batch_size = 128
```

```
train_loader = DataLoader(train_data, batch_size=batch_size, shuffle=True)
val_loader = DataLoader(val_data, batch_size=batch_size, shuffle=False)
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)

# Initialize model
baseline_model = BaselineNet()
print(f"\nBaseline Model:")
print(baseline_model)

total_params = sum(p.numel() for p in baseline_model.parameters())
print(f"\nTotal parameters: {total_params:,}")

# Training function
def train_epoch(model, train_loader, criterion, optimizer):
    model.train()
    running_loss = 0.0
    correct = 0
    total = 0

    for images, labels in train_loader:
        outputs = model(images)
        loss = criterion(outputs, labels)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        running_loss += loss.item()
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

    return running_loss / len(train_loader), correct / total

def validate(model, val_loader, criterion):
    model.eval()
    running_loss = 0.0
    correct = 0
    total = 0

    with torch.no_grad():
        for images, labels in val_loader:
            outputs = model(images)
            loss = criterion(outputs, labels)

            running_loss += loss.item()
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    return running_loss / len(val_loader), correct / total

# Train baseline model
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(baseline_model.parameters(), lr=0.001)
```

```
print("\nTraining baseline model...")
epochs = 10
baseline_history = {'train_loss': [], 'val_loss': [], 'train_acc': [], 'val_acc': []}

for epoch in range(epochs):
    train_loss, train_acc = train_epoch(baseline_model, train_loader, criterion, optimizer)
    val_loss, val_acc = validate(baseline_model, val_loader, criterion)

    baseline_history['train_loss'].append(train_loss)
    baseline_history['val_loss'].append(val_loss)
    baseline_history['train_acc'].append(train_acc)
    baseline_history['val_acc'].append(val_acc)

    print(f"Epoch {epoch+1:2d}/{epochs}: "
          f"Train Loss: {train_loss:.4f}, Train Acc: {train_acc:.4f}, "
          f"Val Loss: {val_loss:.4f}, Val Acc: {val_acc:.4f}")

# Test baseline
test_loss, test_acc = validate(baseline_model, test_loader, criterion)
print(f"\n???? Baseline Test Accuracy: {test_acc:.4f}")

print("\n??? Baseline model complete")
```

Afternoon Session (4 hours)

Phase 3: Improved Model (90 min)

Build a better model with what you've learned:

```
print("\n" + "="*70)
print("PHASE 3: IMPROVED MODEL")
print("="*70)

class ImprovedNet(nn.Module):
    def __init__(self, dropout_prob=0.3):
        super(ImprovedNet, self).__init__()
        self.flatten = nn.Flatten()

        # Deeper network with batch norm and dropout
        self.fc1 = nn.Linear(28*28, 512)
        self.bn1 = nn.BatchNorm1d(512)
        self.dropout1 = nn.Dropout(dropout_prob)

        self.fc2 = nn.Linear(512, 256)
        self.bn2 = nn.BatchNorm1d(256)
        self.dropout2 = nn.Dropout(dropout_prob)

        self.fc3 = nn.Linear(256, 128)
        self.bn3 = nn.BatchNorm1d(128)
        self.dropout3 = nn.Dropout(dropout_prob)

        self.fc4 = nn.Linear(128, 10)

    def forward(self, x):
        x = self.flatten(x)
        x = self.fc1(x)
        x = self.bn1(x)
        x = torch.relu(x)
        x = self.dropout1(x)

        x = self.fc2(x)
        x = self.bn2(x)
        x = torch.relu(x)
        x = self.dropout2(x)

        x = self.fc3(x)
        x = self.bn3(x)
        x = torch.relu(x)
        x = self.dropout3(x)

        x = self.fc4(x)
        return x

# Create improved model
improved_model = ImprovedNet(dropout_prob=0.25)
print(f"\nImproved Model:")
print(improved_model)

total_params = sum(p.numel() for p in improved_model.parameters())
print(f"\nTotal parameters: {total_params:,}")

# Train improved model
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(improved_model.parameters(), lr=0.001, weight_decay=1e-4)

print("\nTraining improved model...")
epochs = 15
improved_history = {'train_loss': [], 'val_loss': [], 'train_acc': [], 'val_acc': []}

best_val_acc = 0
patience = 5
patience_counter = 0

for epoch in range(epochs):
    train_loss, train_acc = train_epoch(improved_model, train_loader, criterion, optimizer)
    val_loss, val_acc = validate(improved_model, val_loader, criterion)

    improved_history['train_loss'].append(train_loss)
    improved_history['val_loss'].append(val_loss)
    improved_history['train_acc'].append(train_acc)
    improved_history['val_acc'].append(val_acc)

    print(f"Epoch {epoch+1:2d}/{epochs}: "
          f"Train Loss: {train_loss:.4f}, Train Acc: {train_acc:.4f}, "
          f"Val Loss: {val_loss:.4f}, Val Acc: {val_acc:.4f}")

    # Early stopping
    if val_acc > best_val_acc:
        best_val_acc = val_acc
        patience_counter = 0
```

```
            # Save best model
            torch.save(improved_model.state_dict(), 'best_mnist_model.pth')
        else:
            patience_counter += 1

        if patience_counter >= patience:
            print(f"Early stopping at epoch {epoch+1}")
            break

# Load best model
improved_model.load_state_dict(torch.load('best_mnist_model.pth'))

# Test improved model
test_loss, test_acc = validate(improved_model, test_loader, criterion)
print(f"\n???? Improved Test Accuracy: {test_acc:.4f}")

# Compare models
print("\n" + "="*70)
print("MODEL COMPARISON")
print("="*70)
print(f"Baseline Test Accuracy:  {baseline_history['val_acc'][-1]:.4f}")
print(f"Improved Test Accuracy:  {test_acc:.4f}")
print(f"Improvement: +{(test_acc - baseline_history['val_acc'][-1]):.4f}")

print("\n??? Improved model complete")
```

Phase 4: Analysis & Visualization (60 min)

Comprehensive evaluation of the best model:

```
print("\n" + "="*70)
print("PHASE 4: ANALYSIS & VISUALIZATION")
print("="*70)

# 1. Training curves comparison
fig, axes = plt.subplots(2, 2, figsize=(14, 10))

axes[0, 0].plot(baseline_history['train_loss'], label='Train', alpha=0.7)
axes[0, 0].plot(baseline_history['val_loss'], label='Val', alpha=0.7)
axes[0, 0].set_title('Baseline - Loss')
axes[0, 0].set_xlabel('Epoch')
axes[0, 0].set_ylabel('Loss')
axes[0, 0].legend()
axes[0, 0].grid(True, alpha=0.3)

axes[0, 1].plot(improved_history['train_loss'], label='Train', alpha=0.7)
axes[0, 1].plot(improved_history['val_loss'], label='Val', alpha=0.7)
axes[0, 1].set_title('Improved - Loss')
axes[0, 1].set_xlabel('Epoch')
axes[0, 1].set_ylabel('Loss')
axes[0, 1].legend()
axes[0, 1].grid(True, alpha=0.3)

axes[1, 0].plot(baseline_history['train_acc'], label='Train', alpha=0.7)
axes[1, 0].plot(baseline_history['val_acc'], label='Val', alpha=0.7)
```

```
axes[1, 0].set_title('Baseline - Accuracy')
axes[1, 0].set_xlabel('Epoch')
axes[1, 0].set_ylabel('Accuracy')
axes[1, 0].legend()
axes[1, 0].grid(True, alpha=0.3)

axes[1, 1].plot(improved_history['train_acc'], label='Train', alpha=0.7)
axes[1, 1].plot(improved_history['val_acc'], label='Val', alpha=0.7)
axes[1, 1].set_title('Improved - Accuracy')
axes[1, 1].set_xlabel('Epoch')
axes[1, 1].set_ylabel('Accuracy')
axes[1, 1].legend()
axes[1, 1].grid(True, alpha=0.3)

plt.suptitle('Training Comparison', fontsize=14, fontweight='bold')
plt.tight_layout()
plt.show()

# 2. Confusion matrix
print("\nGenerating confusion matrix...")
improved_model.eval()
all_preds = []
all_labels = []

with torch.no_grad():
    for images, labels in test_loader:
        outputs = improved_model(images)
        _, predicted = torch.max(outputs, 1)
        all_preds.extend(predicted.cpu().numpy())
        all_labels.extend(labels.cpu().numpy())

cm = confusion_matrix(all_labels, all_preds)

plt.figure(figsize=(10, 8))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', cbar=True)
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.title(f'Confusion Matrix - Test Accuracy: {test_acc:.4f}')
plt.show()

# 3. Per-class accuracy
print("\nPer-class Performance:")
print("-" * 50)
for i in range(10):
    class_correct = cm[i, i]
    class_total = cm[i, :].sum()
    class_acc = class_correct / class_total
    print(f"Digit {i}: {class_acc:.4f} ({class_correct}/{class_total})")

# 4. Visualize mistakes
print("\nFinding misclassified examples...")
improved_model.eval()
mistakes = {'images': [], 'true': [], 'pred': [], 'conf': []}

with torch.no_grad():
    for images, labels in test_loader:
        outputs = improved_model(images)
        probs = torch.softmax(outputs, dim=1)
        confidences, predicted = torch.max(probs, 1)

        # Find mistakes
        mask = predicted != labels
        if mask.sum() > 0:
            mistakes['images'].extend(images[mask])
            mistakes['true'].extend(labels[mask])
            mistakes['pred'].extend(predicted[mask])
            mistakes['conf'].extend(confidences[mask])

        if len(mistakes['images']) >= 20:
            break

# Plot mistakes
fig, axes = plt.subplots(4, 5, figsize=(12, 10))
axes = axes.flatten()

for i in range(min(20, len(mistakes['images']))):
    img = mistakes['images'][i].squeeze()
    true_label = mistakes['true'][i].item()
    pred_label = mistakes['pred'][i].item()
    conf = mistakes['conf'][i].item()

    axes[i].imshow(img, cmap='gray')
    axes[i].set_title(f'True: {true_label}\nPred: {pred_label} ({conf:.2f})', fontsize=9)
    axes[i].axis('off')

plt.suptitle('Misclassified Examples', fontsize=14, fontweight='bold')
plt.tight_layout()
plt.show()

print("\n??? Analysis complete")
```

Phase 5: Documentation (30 min)

Create professional project documentation:

```
print("\n" + "="*70)
print("PHASE 5: DOCUMENTATION")
print("="*70)

# Generate project report
report = f"""
{'='*70}
MNIST DIGIT CLASSIFICATION - PROJECT REPORT
{'='*70}

PROJECT OVERVIEW
----------------
Goal: Build a neural network to classify handwritten digits (MNIST dataset)
Target Accuracy: >95%
Achieved Accuracy: {test_acc:.4f}
Status: {'??? SUCCESS' if test_acc > 0.95 else '??? TARGET NOT MET'}
```

```
DATASET
-------
Training samples: {len(train_data):,}
Validation samples: {len(val_data):,}
Test samples: {len(test_dataset):,}
Image size: 28x28 pixels
Classes: 10 (digits 0-9)
Preprocessing: Normalized to mean=0.1307, std=0.3081

MODELS DEVELOPED
----------------

1. Baseline Model
   - Architecture: Simple 2-layer network (784 ??? 128 ??? 10)
   - Parameters: {sum(p.numel() for p in baseline_model.parameters()):,}
   - Test Accuracy: {baseline_history['val_acc'][-1]:.4f}

2. Improved Model
   - Architecture: Deep network with regularization
     * 4 fully connected layers (784 ??? 512 ??? 256 ??? 128 ??? 10)
     * Batch normalization after each hidden layer
     * Dropout (p=0.25) for regularization
     * ReLU activation functions
   - Parameters: {sum(p.numel() for p in improved_model.parameters()):,}
   - Test Accuracy: {test_acc:.4f}
   - Improvement: +{(test_acc - baseline_history['val_acc'][-1]):.4f}

TRAINING DETAILS
----------------
Optimizer: Adam
Learning Rate: 0.001
Weight Decay: 1e-4
Batch Size: 128
Epochs: {len(improved_history['train_acc'])} (with early stopping)
Loss Function: Cross Entropy Loss

RESULTS SUMMARY
---------------
Best Validation Accuracy: {best_val_acc:.4f}
Final Test Accuracy: {test_acc:.4f}
Average per-class accuracy: {np.diag(cm).sum() / cm.sum():.4f}

Most confused pairs:
"""

# Find most confused digit pairs
cm_normalized = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
np.fill_diagonal(cm_normalized, 0)
confused_pairs = []

for i in range(10):
    for j in range(10):
        if i != j and cm_normalized[i, j] > 0.05:
            confused_pairs.append((i, j, cm_normalized[i, j]))
confused_pairs.sort(key=lambda x: x[2], reverse=True)

for i, (true_digit, pred_digit, conf_rate) in enumerate(confused_pairs[:5]):
    report += f"  {i+1}. Digit {true_digit} classified as {pred_digit}: {conf_rate:.2%}\n"

report += f"""

KEY INSIGHTS
------------
1. Batch normalization stabilized training
2. Dropout helped prevent overfitting
3. Deeper network improved accuracy significantly
4. Early stopping prevented overtraining

FUTURE IMPROVEMENTS
-------------------
1. Implement Convolutional Neural Networks (Week 3!)
2. Data augmentation (rotation, scaling)
3. Ensemble methods
4. Hyperparameter tuning with grid search

CONCLUSION
----------
Successfully built a neural network achieving {test_acc:.4f} test accuracy
on MNIST digit classification. This demonstrates understanding of:
- Neural network architecture design
- PyTorch implementation
- Training with regularization
- Model evaluation and analysis

{'='*70}
Week 2 Capstone Project Complete! ????
{'='*70}
"""

print(report)

# Save report to file
with open('mnist_project_report.txt', 'w') as f:
    f.write(report)

print("\n???? Report saved to: mnist_project_report.txt")
print("???? Model saved to: best_mnist_model.pth")

print("\n??? Documentation complete")
print("\n" + "="*70)
print("???? CONGRATULATIONS! PROJECT COMPLETE!")
print("="*70)
```

Reflection & Week Review (30 min)

??? Review entire Week 2 journey ??? Document key learnings ??? Celebrate achievements ??? Prepare for Week 3

Week 2 Reflection Prompts (Address All):

- What was the most valuable thing you learned this week?
- How has your understanding of neural networks evolved?
- What was your biggest challenge? How did you overcome it?
- How does your final MNIST accuracy compare to your expectations?
- What connections did you make between Days 6-10?
- How confident do you feel about neural networks now?
- What are you most excited to learn in Week 3 (CNNs)?
- What from Week 2 needs more practice?

Week 2 Achievement Checklist:

??? Understood neural network architecture ??? Implemented forward propagation from scratch ??? Implemented backpropagation from scratch ??? Solved XOR problem ??? Learned PyTorch fundamentals ??? Built models with nn.Module ??? Used Dataset and DataLoader ??? Applied dropout and batch normalization ??? Completed MNIST project with >95% accuracy ??? Created professional documentation

---

???? Week 2 Complete!

Achievements Unlocked: - ??? Neural networks from first principles - ??? Backpropagation mastery - ??? PyTorch proficiency - ??? Production-ready training pipelines - ??? Portfolio project complete

Next Week Preview: Week 3 introduces Convolutional Neural Networks (CNNs) - the backbone of computer vision. You'll learn about convolutions, pooling, modern architectures (ResNet, VGG), and complete CIFAR-10 classification!

Weekend Recommendations: - Review your Week 2 notes - Share your MNIST results with classmates - Optional: Try improving your MNIST model further - Rest and prepare for CNNs!

Next: Week 3 Overview