Neural Networks II: Making Training Work

Introduction

▶ Code

Last time (Neural Networks I) was about how learning works. Today is deliberately practical: **making training work** with scikit-learn.

We'll cover:

- `MLPClassifier` and `MLPRegressor`
- A classification example (MNIST) and a regression example (California housing)
- Hyperparameter tuning with grid search
- Preprocessing, architecture selection, preventing overfitting, choosing a solver
- Common issues and how to diagnose them

Last Time: How Learning Works

Today: The Knobs

Every one of those ideas shows up as an argument you have to choose:

Concept	scikit-learn argument
Layers and neurons	`hidden_layer_sizes`
Activation f	`activation`
Learning rate η	`learning_rate_init`, `learning_rate`
Mini-batch size	`batch_size`
Optimizer	`solver` (`'adam'`, `'sgd'`, `'lbfgs'`)
Epochs / stopping	`max_iter`, `early_stopping`, `n_iter_no_change`
Regularization	`alpha`

The rest of the lecture is about setting these sensibly – and what goes wrong when you don't.

Neural Networks in Scikit-Learn

MLPClassifier and MLPRegressor

Scikit-learn provides simple, high-level interfaces:

- `MLPClassifier`: Multi-layer Perceptron classifier
- `MLPRegressor`: Multi-layer Perceptron regressor

Key features:

- Multiple hidden layers with various activation functions
- Multiple solvers: `'adam'`, `'sgd'`, `'lbfgs'`
- Built-in regularization (L2 penalty)
- Early stopping support
- Easy integration with scikit-learn pipelines

Architecture Specification

Specify architecture as a tuple:

```
# Single hidden layer with 100 neurons
hidden_layer_sizes=(100,)

# Two hidden layers: 100 and 50 neurons
hidden_layer_sizes=(100, 50)

# Three hidden layers
hidden_layer_sizes=(128, 64, 32)
```

Input and output layers are automatically determined from your data!

Scikit-Learn vs PyTorch/TensorFlow

Use Scikit-Learn for:

- Small to medium datasets (< 100K samples)
- Standard feedforward architectures
- Rapid prototyping needed
- Integration with scikit-learn pipelines
- CPU training is sufficient

Use PT/TF for:

- Large datasets (> 100K samples)
- Complex architectures (CNNs, RNNs)
- GPU acceleration required
- Production deployment
- Research and experimentation

Classification Example: MNIST

Load the MNIST Dataset

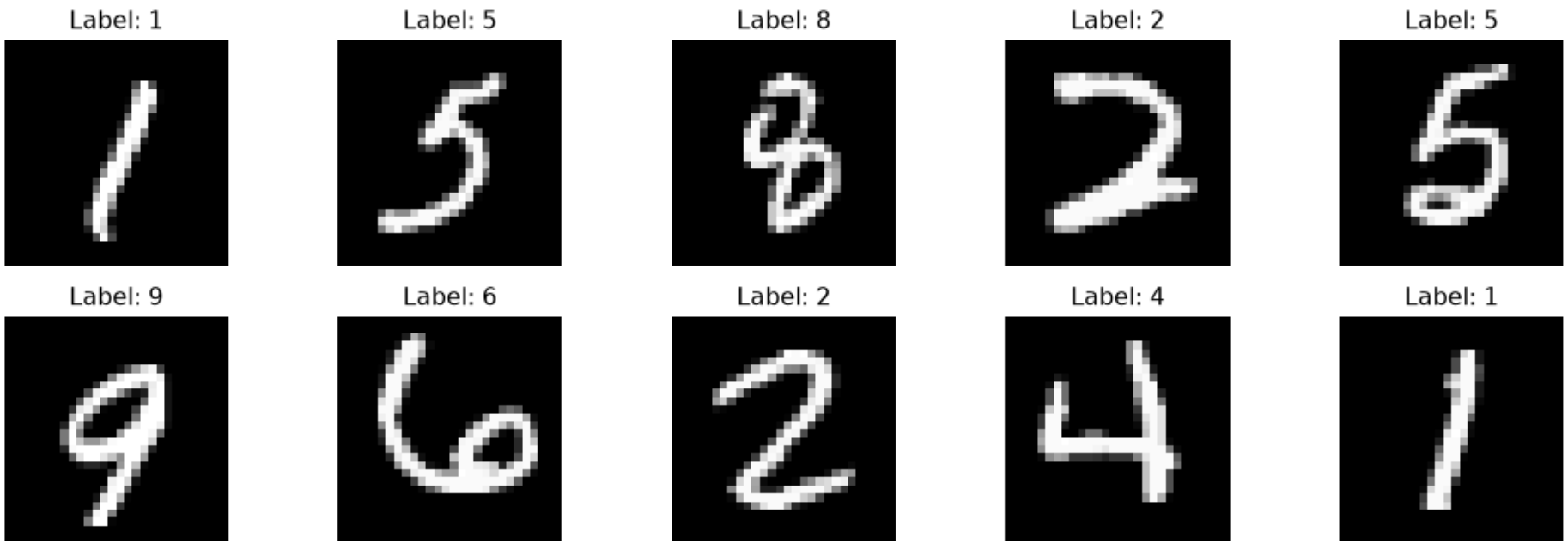
Let's classify handwritten digits (0-9):

```python
from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split

# Load MNIST data
print("Loading MNIST dataset...")
X, y = fetch_openml('mnist_784', version=1, return_X_y=True,
                    as_frame=False, parser='auto')

# Convert labels to integers
y = y.astype(int)

# Use subset for faster demo
X, _, y, _ = train_test_split(X, y, train_size=10000,
                              stratify=y, random_state=42)

print(f"Dataset shape: {X.shape}")
print(f"Number of classes: {len(np.unique(y))}")
```

```
Loading MNIST dataset...
Dataset shape: (10000, 784)
Number of classes: 10
```

Visualize the Data

▶ Code



Each image is 28×28 pixels = 784 features

Prepare the Data

```python
# Split into training and test sets
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# Scale features to [0, 1]
X_train_scaled = X_train / 255.0
X_test_scaled = X_test / 255.0

print(f"Training set: {X_train.shape[0]} samples")
print(f"Test set: {X_test.shape[0]} samples")
print(f"Feature range: [{X_train_scaled.min():.2f}, {X_train_scaled.max():.2f}]")
```

```
Training set: 8000 samples
Test set: 2000 samples
Feature range: [0.00, 1.00]
```

Important

Always scale/normalize your features for neural networks! This helps gradient descent converge faster.

Create the MLP

```
from sklearn.neural_network import MLPClassifier

# Create MLP with 2 hidden layers
mlp = MLPClassifier(
    hidden_layer_sizes=(128, 64),  # Architecture
    activation='relu',              # Activation function
    solver='adam',                  # Optimizer (uses mini-batches)
    alpha=0.0001,                   # L2 regularization
    batch_size=64,                  # Mini-batch size
    learning_rate_init=0.001,       # Initial learning rate
    max_iter=20,                    # Number of epochs
    random_state=42,
    verbose=True                    # Show progress
)
```

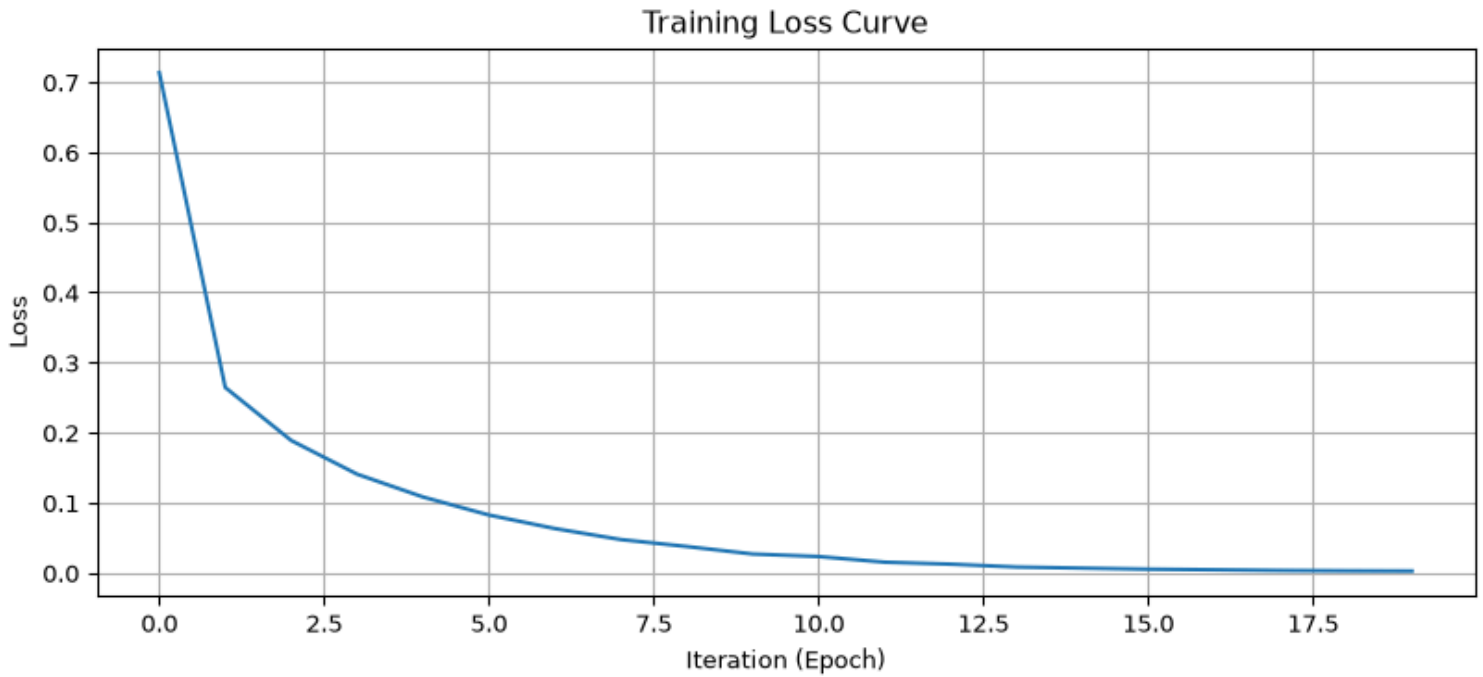
Train the model

```
print("Training MLP...")
mlp.fit(X_train_scaled, y_train)
print(f"Training completed in {mlp.n_iter_} iterations")
```

```
Training MLP...
Iteration 1, loss = 0.71345206
Iteration 2, loss = 0.26439384
Iteration 3, loss = 0.18880844
Iteration 4, loss = 0.14070827
Iteration 5, loss = 0.10813206
Iteration 6, loss = 0.08241134
Iteration 7, loss = 0.06312883
Iteration 8, loss = 0.04742022
Iteration 9, loss = 0.03751731
Iteration 10, loss = 0.02680532
Iteration 11, loss = 0.02321482
Iteration 12, loss = 0.01516949
Iteration 13, loss = 0.01235069
Iteration 14, loss = 0.00823132
Iteration 15, loss = 0.00665425
Iteration 16, loss = 0.00515634
Iteration 17, loss = 0.00431783
Iteration 18, loss = 0.00336489
Iteration 19, loss = 0.00283319
```

Training Loss Curve

▶ Code



The loss decreases smoothly—our model is learning.

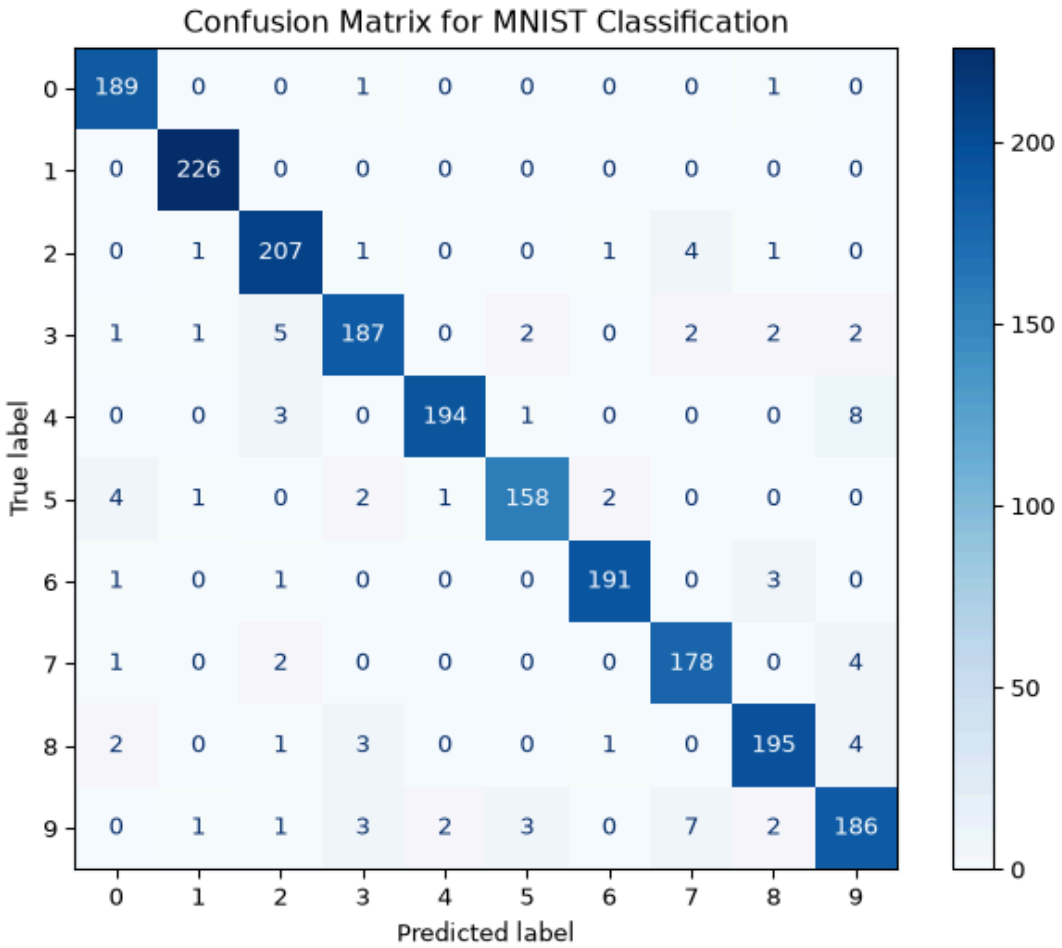
Evaluate Performance

▶ Code

```
Test Accuracy: 0.9555

Classification Report:
              precision    recall  f1-score   support

           0       0.95      0.99      0.97       191
           1       0.98      1.00      0.99       226
           2       0.94      0.96      0.95       215
           3       0.95      0.93      0.94       202
           4       0.98      0.94      0.96       206
           5       0.96      0.94      0.95       168
           6       0.98      0.97      0.98       196
           7       0.93      0.96      0.95       185
           8       0.96      0.95      0.95       206
           9       0.91      0.91      0.91       205

    accuracy                           0.96      2000
   macro avg       0.96      0.96      0.96      2000
weighted avg       0.96      0.96      0.96      2000
```

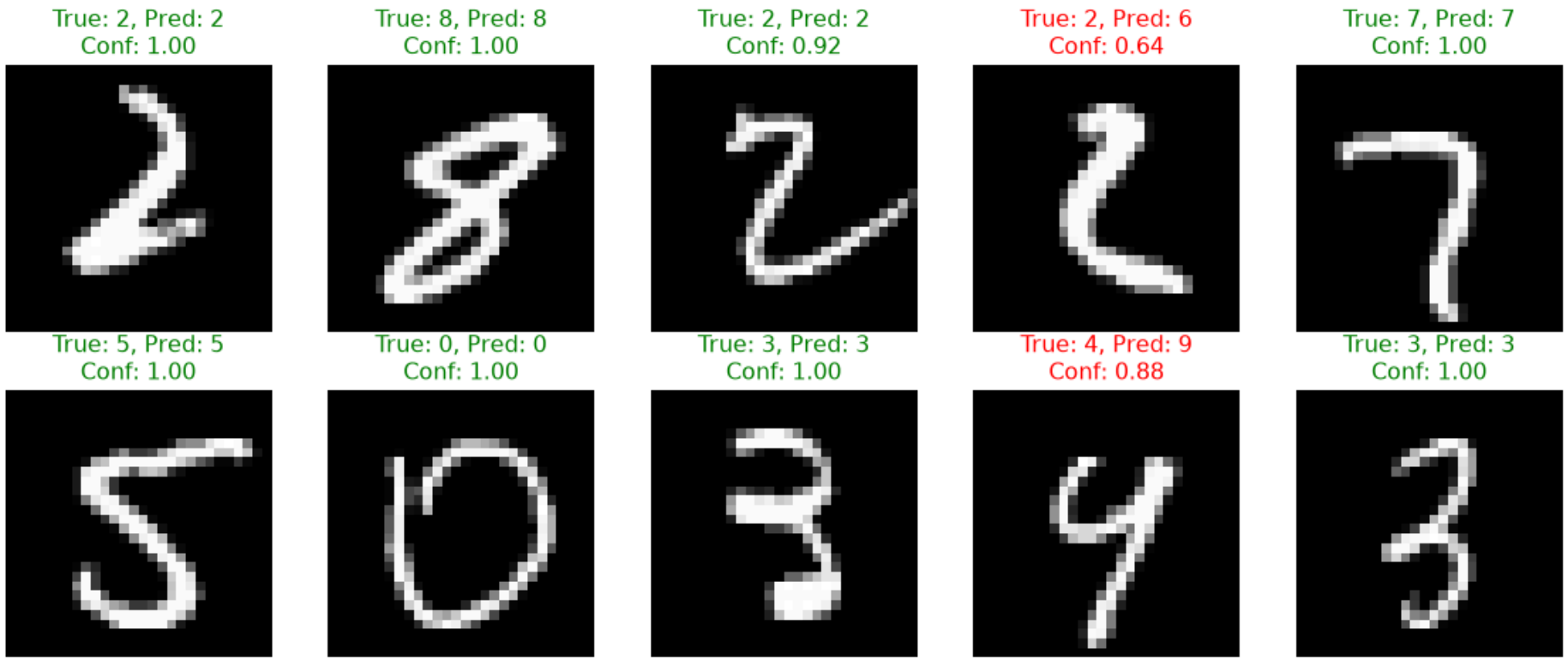
Confusion Matrix

- Code



Visualize Predictions

▶ Code



Regression Example

California Housing Dataset

Let's predict house prices using MLP regression:

```
1  from sklearn.datasets import fetch_california_housing
2  from sklearn.preprocessing import StandardScaler
3  from sklearn.neural_network import MLPRegressor
4
5  # Load housing data
6  housing = fetch_california_housing()
7  X_housing = housing.data
8  y_housing = housing.target
9
10 print(f"Dataset shape: {X_housing.shape}")
11 print(f"Features: {housing.feature_names}")
```

```
Dataset shape: (20640, 8)
Features: ['MedInc', 'HouseAge', 'AveRooms', 'AveBedrms', 'Population', 'AveOccup', 'Latitude', 'Longitude']
```

Target: Median house value (in $100,000s)

Prepare Data and Define Model

```
# Split and scale
X_train_h, X_test_h, y_train_h, y_test_h = train_test_split(
    X_housing, y_housing, test_size=0.2, random_state=42
)

scaler = StandardScaler()
X_train_h_scaled = scaler.fit_transform(X_train_h)
X_test_h_scaled = scaler.transform(X_test_h)

# Train MLP Regressor with early stopping
mlp_reg = MLPRegressor(
    hidden_layer_sizes=(100, 50),
    activation='relu',
    solver='adam',
    alpha=0.001,
    batch_size=32,
    max_iter=100,
    early_stopping=True,
    validation_fraction=0.1,
    n_iter_no_change=10,
    random_state=42,
    verbose=False
)
```

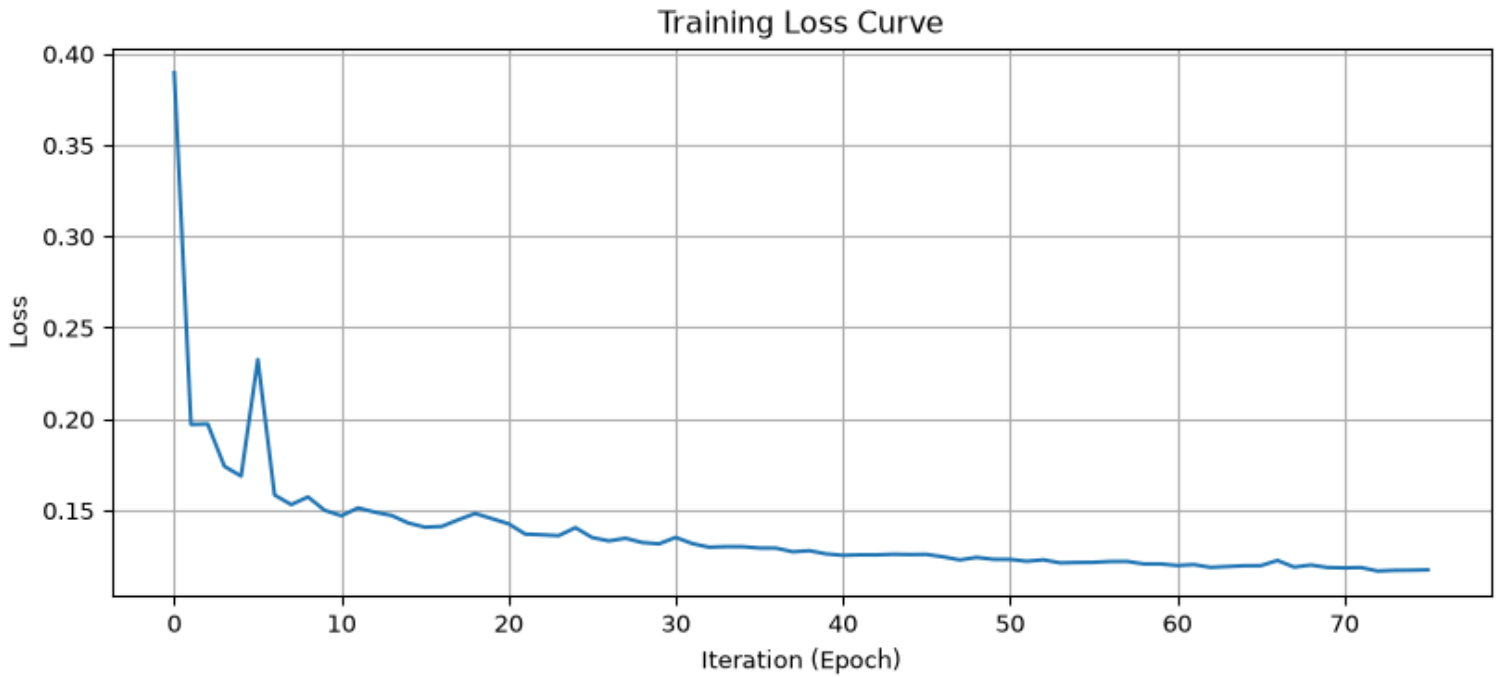
Train the model

```
print("Training MLP Regressor...")
mlp_reg.fit(X_train_h_scaled, y_train_h)
print(f"Training stopped at iteration: {mlp_reg.n_iter_}")
```

```
Training MLP Regressor...
Training stopped at iteration: 76
```

Training Loss Curve

- Code



Evaluate Regression Model

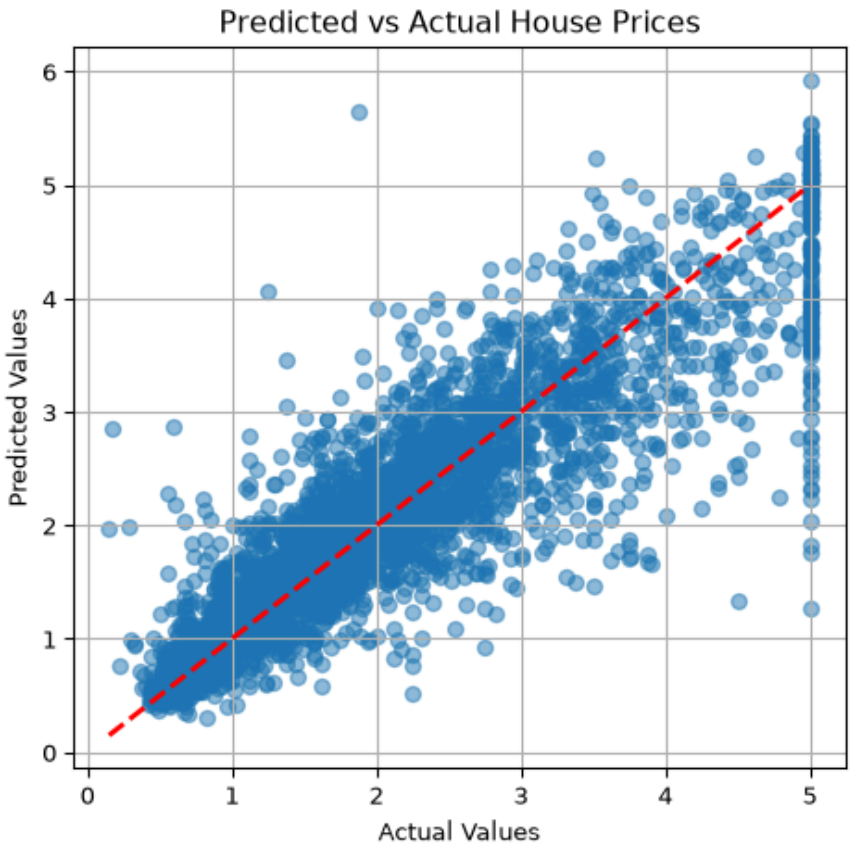
```python
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error

# Make predictions
y_pred_h = mlp_reg.predict(X_test_h_scaled)

# Calculate metrics
mse = mean_squared_error(y_test_h, y_pred_h)
rmse = np.sqrt(mse)
mae = mean_absolute_error(y_test_h, y_pred_h)
r2 = r2_score(y_test_h, y_pred_h)

print(f"Root Mean Squared Error: {rmse:.4f}")
print(f"Mean Absolute Error: {mae:.4f}")
print(f"R² Score: {r2:.4f}")
```

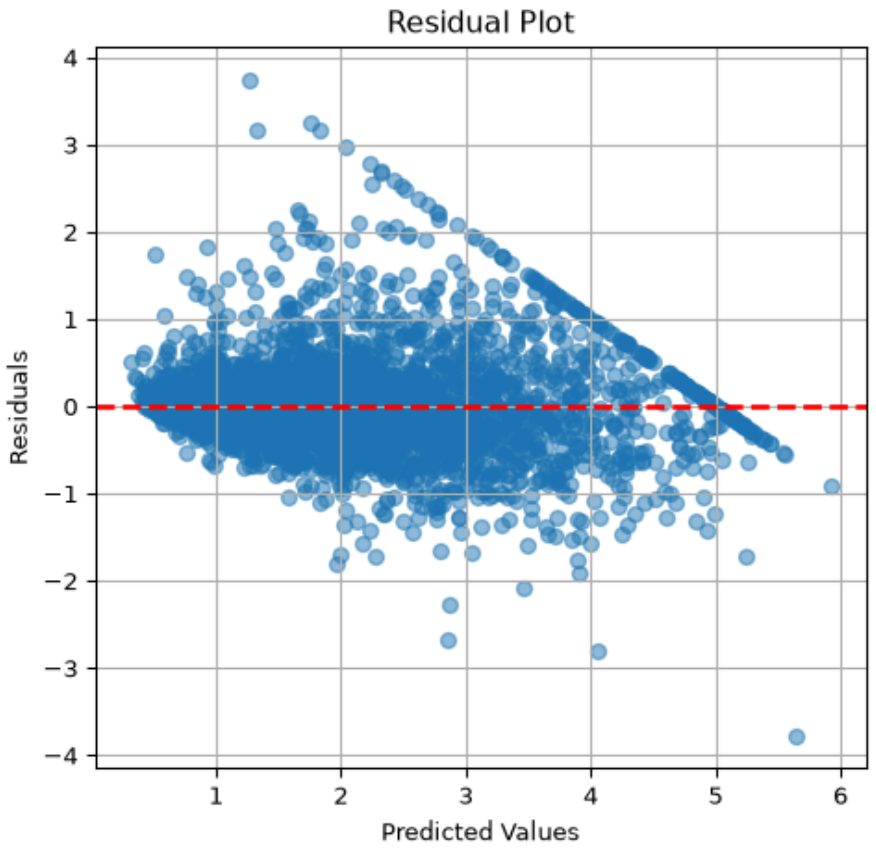
```
Root Mean Squared Error: 0.5138
Mean Absolute Error: 0.3460
R² Score: 0.7986
```

An R^2 of ~0.8 means our model explains 80% of the variance in house prices!

Predictions vs Actual

▶ Code





Hyperparameter Tuning

Grid Search for Optimal Architecture

scikit-learn provides `GridSearchCV` for hyperparameter tuning:

Create a model without certain hyperparameters.

```
# Create MLP
mlp_grid = MLPClassifier(
    max_iter=20,
    random_state=42,
    early_stopping=True,
    validation_fraction=0.1,
    n_iter_no_change=5,
    verbose=False
)

# Use subset for faster demo
X_grid = X_train_scaled[:1500]
y_grid = y_train[:1500]
```

Grid Search, cont.

Define the parameter grid and run the grid search.

```
from sklearn.model_selection import GridSearchCV

# Define parameter grid
param_grid = {
    'hidden_layer_sizes': [(50,), (100,), (50, 50)],
    'activation': ['relu'],
    'alpha': [0.0001, 0.001]
}

print("Running Grid Search...")
grid_search = GridSearchCV(mlp_grid, param_grid, cv=3, n_jobs=2, verbose=0)
grid_search.fit(X_grid, y_grid)

print(f"\nBest parameters: {grid_search.best_params_}")
print(f"Best CV score: {grid_search.best_score_:.4f}")
```

```
Running Grid Search...

Best parameters: {'activation': 'relu', 'alpha': 0.001, 'hidden_layer_sizes': (50,)}
Best CV score: 0.8740
```

Grid Search Results

▶ Code

```
Top configurations:

Configuration 4:
  activation: relu
  alpha: 0.001
  hidden_layer_sizes: (50,)

Configuration 1:
  activation: relu
  alpha: 0.0001
  hidden_layer_sizes: (50,)

Configuration 2:
  activation: relu
  alpha: 0.0001
  hidden_layer_sizes: (100,)

Configuration 5:
  activation: relu
```

Best Practices

Data Preprocessing

Always preprocess your data!

1. **Scale features:** Use `StandardScaler` or normalize to [0, 1]
2. **Handle missing values:** Impute or remove
3. **Encode categorical variables:** One-hot encoding
4. **Use pipelines:** Ensures consistent preprocessing

```
from sklearn.pipeline import Pipeline

pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('mlp', MLPClassifier(hidden_layer_sizes=(100,)))
])

pipeline.fit(X_train, y_train)
```

Architecture Selection

Rules of thumb for architecture:

1. **Start simple:** Try single hidden layer first
2. **Layer sizes:** Between input and output dimensions
3. **Depth vs width:**
    - More layers → learn complex patterns
    - But risk overfitting on small data
4. **Typical architectures:**
    - Small data: `(100,)` or `(50, 50)`
    - Medium data: `(100, 50)` or `(128, 64, 32)`
    - Large data: Consider PyTorch

Preventing Overfitting

Three key techniques:

1. Regularization: Add L2 penalty (`alpha` parameter)

```
mlp = MLPClassifier(alpha=0.01)  # Stronger regularization
```

2. Early Stopping: Stop when validation performance plateaus

```
mlp = MLPClassifier(early_stopping=True,
                    validation_fraction=0.2,
                    n_iter_no_change=10)
```

3. Cross-Validation: Get robust performance estimates

```
from sklearn.model_selection import cross_val_score
scores = cross_val_score(mlp, X_train, y_train, cv=5)
```

Solver Selection

Different solvers for different scenarios:

Solver	Best For	Notes
`'adam'`	Most cases	Good default, adaptive learning rate
`'sgd'`	Large datasets	Classic mini-batch SGD
`'lbfgs'`	Small datasets	Faster for small data, more memory

▶ Code

```
adam      : 0.9000
sgd       : 0.8330
lbfgs     : 0.8800
```

Common Issues

⚠ **Convergence Warnings**

If you see `ConvergenceWarning`:

1. **Increase** `max_iter`
2. **Decrease** `learning_rate_init`
3. **Enable** `early_stopping=True`
4. **Check** if data is properly scaled

⚠ **Poor Performance**

If accuracy is low:

1. Is data scaled/normalized?
2. Is architecture appropriate?
3. Is learning rate too high/low?
4. Do you need more iterations?
5. Is regularization too strong?

Summary

Summary

Theory (from Neural Networks I):

- Neural networks extend linear/logistic regression with multiple layers and non-linearity
- Gradient descent optimizes the loss; backpropagation supplies the gradients
- Mini-batch GD balances speed and stability

Practice (today):

- Scikit-learn's `MLPClassifier` and `MLPRegressor` for easy implementation
- Always preprocess/scale your data
- Use early stopping and regularization to prevent overfitting
- Grid search helps find optimal hyperparameters

To Dig Deeper

Other modules in the course notes:

- Neural Networks I: How Learning Works
- NN I – Gradient Descent
- NN II – Compute Graph and Backpropagation
- NN III – SGD and CNNs
- NN IV – NNs with Scikit-Learn

Additional resources:

- Understanding Deep Learning, Simon J.D. Prince, MIT Press, 2023
- DS542, Deep Learning for Data Science