Machine Learning Code Explanation

Practice Lab: Advice for Applying Machine Learning

Table of Contents

---

1. Package Imports and Setup

Code:

python

```python
import numpy as np
%matplotlib widget
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression, Ridge
from sklearn.preprocessing import StandardScaler, PolynomialFeatures
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.activations import relu,linear
from tensorflow.keras.losses import SparseCategoricalCrossentropy
from tensorflow.keras.optimizers import Adam

import logging
logging.getLogger("tensorflow").setLevel(logging.ERROR)

from public_tests_a1 import *

tf.keras.backend.set_floatx('float64')
from assigment_utils import *

tf.autograph.set_verbosity(0)
```

Explanation:

This section imports all necessary libraries for machine learning tasks:

- **NumPy**: Fundamental package for numerical computing in Python
- **Matplotlib**: Library for creating visualizations and plots
- **Scikit-learn**: Machine learning library providing tools for:
  - Linear regression models (LinearRegression, Ridge)
  - Data preprocessing (StandardScaler, PolynomialFeatures)
  - Data splitting (train_test_split)
  - Performance metrics (mean_squared_error)
- **TensorFlow/Keras**: Deep learning framework for neural networks
- **Logging**: Controls TensorFlow warning messages

---

2. Data Splitting Techniques

Topic: Data Splitting in Machine Learning

Data splitting is a fundamental technique in machine learning that involves dividing your dataset into separate portions for training, validation, and testing. This is crucial for:

1. **Training the model**: Using one portion to learn patterns
2. **Validating performance**: Using another portion to tune hyperparameters
3. **Testing generalization**: Using a final portion to evaluate real-world performance

Code Example 1: Basic Train-Test Split

python

```python
# Generate some data
X,y,x_ideal,y_ideal = gen_data(18, 2, 0.7)
print("X.shape", X.shape, "y.shape", y.shape)

#split the data using sklearn routine
X_train, X_test, y_train, y_test = train_test_split(X,y,test_size=0.33, random_state=1)
print("X_train.shape", X_train.shape, "y_train.shape", y_train.shape)
print("X_test.shape", X_test.shape, "y_test.shape", y_test.shape)
```

Explanation:

- `gen_data(18, 2, 0.7)`: Generates synthetic data with 18 points, degree 2 polynomial, and 0.7 noise level
- `train_test_split()`: Splits data into training (67%) and testing (33%) sets
- `random_state=1`: Ensures reproducible results by fixing the random seed

Code Example 2: Three-way Split (Train, Cross-validation, Test)

```python
# Generate  data
X,y, x_ideal,y_ideal = gen_data(40, 5, 0.7)
print("X.shape", X.shape, "y.shape", y.shape)

#split the data using sklearn routine
X_train, X_, y_train, y_ = train_test_split(X,y,test_size=0.40, random_state=1)
X_cv, X_test, y_cv, y_test = train_test_split(X_,y_,test_size=0.50, random_state=1)
print("X_train.shape", X_train.shape, "y_train.shape", y_train.shape)
print("X_cv.shape", X_cv.shape, "y_cv.shape", y_cv.shape)
print("X_test.shape", X_test.shape, "y_test.shape", y_test.shape)
```

Explanation: This creates three datasets:

- **Training set (60%)**: Used to train model parameters
- **Cross-validation set (20%)**: Used to tune hyperparameters and select models
- **Test set (20%)**: Used for final performance evaluation

Why Three Sets?

- Training set: Learns patterns in data
- Cross-validation set: Prevents overfitting during model selection
- Test set: Provides unbiased performance estimate

---

3. Error Calculation and Model Evaluation

Topic: Mean Squared Error (MSE)

Mean Squared Error is a common metric for evaluating regression models. It measures the average squared difference between predicted and actual values.

Mathematical Formula: $J_{test}(w,b) = (1/2m_{test}) * \Sigma(f_{w,b}(x^{(i)}_{test}) - y^{(i)}_{test})^2$

Exercise 1 Code:

python

```python
def eval_mse(y, yhat):
    """
    Calculate the mean squared error on a data set.
    Args:
      y    : (ndarray  Shape (m,) or (m,1))  target value of each example
      yhat : (ndarray  Shape (m,) or (m,1))  predicted value of each example
    Returns:
      err: (scalar)
    """
    m = len(y)
    err = 0.0
    for i in range(m):
        err += (yhat[i] - y[i])**2
    err = err / (2 * m)

    return(err)

y_hat = np.array([2.4, 4.2])
y_tmp = np.array([2.3, 4.1])
eval_mse(y_hat, y_tmp)
```

Explanation:

- Calculates squared differences between predictions and actual values
- Sums all squared errors
- Divides by 2m (where m is number of examples)
- The factor of 1/2 is used for mathematical convenience in calculus

Model Performance Comparison Code:

```python
# create a model in sklearn, train on training data
degree = 10
lmodel = lin_model(degree)
lmodel.fit(X_train, y_train)

# predict on training data, find training error
yhat = lmodel.predict(X_train)
err_train = lmodel.mse(y_train, yhat)

# predict on test data, find error
yhat = lmodel.predict(X_test)
err_test = lmodel.mse(y_test, yhat)

print(f"training err {err_train:0.2f}, test err {err_test:0.2f}")
```

Output Analysis:

```
training err 58.01, test err 171215.01
```

This large difference indicates **overfitting** - the model performs much better on training data than test data.

---

4. Bias and Variance Analysis

Topic: Bias-Variance Tradeoff

Bias refers to the error introduced by approximating a real-world problem with a simplified model. High bias can cause underfitting.

Variance refers to the error introduced by the model's sensitivity to small fluctuations in the training set. High variance can cause overfitting.

Finding Optimal Polynomial Degree Code:

```python
max_degree = 9
err_train = np.zeros(max_degree)
err_cv = np.zeros(max_degree)
x = np.linspace(0,int(X.max()),100)
y_pred = np.zeros((100,max_degree))  #columns are lines to plot

for degree in range(max_degree):
    lmodel = lin_model(degree+1)
    lmodel.fit(X_train, y_train)
    yhat = lmodel.predict(X_train)
    err_train[degree] = lmodel.mse(y_train, yhat)
    yhat = lmodel.predict(X_cv)
    err_cv[degree] = lmodel.mse(y_cv, yhat)
    y_pred[:,degree] = lmodel.predict(x)

optimal_degree = np.argmin(err_cv)+1
```

Explanation:

- Tests polynomial degrees from 1 to 9
- Calculates training and cross-validation errors for each degree
- Optimal degree is where cross-validation error is minimized
- This helps find the sweet spot between underfitting and overfitting

---

5. Regularization Techniques

Topic: Regularization

Regularization is a technique used to prevent overfitting by adding a penalty term to the loss function. It discourages overly complex models by penalizing large parameter values.

Types of Regularization:

- **L1 Regularization (Lasso)**: Adds sum of absolute values of parameters
- **L2 Regularization (Ridge)**: Adds sum of squared values of parameters

Regularization Parameter Tuning Code:

python

```python
lambda_range = np.array([0.0, 1e-6, 1e-5, 1e-4,1e-3,1e-2, 1e-1,1,10,100])
num_steps = len(lambda_range)
degree = 10
err_train = np.zeros(num_steps)
err_cv = np.zeros(num_steps)
x = np.linspace(0,int(X.max()),100)
y_pred = np.zeros((100,num_steps))  #columns are lines to plot

for i in range(num_steps):
    lambda_= lambda_range[i]
    lmodel = lin_model(degree, regularization=True, lambda_=lambda_)
    lmodel.fit(X_train, y_train)
    yhat = lmodel.predict(X_train)
    err_train[i] = lmodel.mse(y_train, yhat)
    yhat = lmodel.predict(X_cv)
    err_cv[i] = lmodel.mse(y_cv, yhat)
    y_pred[:,i] = lmodel.predict(x)

optimal_reg_idx = np.argmin(err_cv)
```

Explanation:

- Tests different regularization strengths (lambda values)
- Higher lambda values increase regularization strength
- Finds optimal lambda that minimizes cross-validation error
- Balances between fitting training data and generalizing to new data

Effect of Training Set Size Code:

python

```python
X_train, y_train, X_cv, y_cv, x, y_pred, err_train, err_cv, m_range,degree = tune_m()
```

Key Insight:

- When a model has high variance (overfitting), adding more training examples improves performance
- When a model has high bias (underfitting), adding more examples doesn't help significantly

---

6. Neural Network Implementation

Topic: Neural Networks for Classification

Neural networks are powerful models that can learn complex patterns through multiple layers of interconnected neurons. They're particularly effective for classification tasks.

Data Generation for Classification:

```python
# Generate and split data set
X, y, centers, classes, std = gen_blobs()

# split the data. Large CV population for demonstration
X_train, X_, y_train, y_ = train_test_split(X,y,test_size=0.50, random_state=1)
X_cv, X_test, y_cv, y_test = train_test_split(X_,y_,test_size=0.20, random_state=1)
print("X_train.shape:", X_train.shape, "X_cv.shape:", X_cv.shape, "X_test.shape:", X_test.shape
```

Output:

```
X_train.shape: (400, 2) X_cv.shape: (320, 2) X_test.shape: (80, 2)
```

Classification Error Calculation:

Exercise 2 Code:

python

```python
def eval_cat_err(y, yhat):
    """
    Calculate the categorization error
    Args:
      y    : (ndarray  Shape (m,) or (m,1))  target value of each example
      yhat : (ndarray  Shape (m,) or (m,1))  predicted value of each example
    Returns:
      cerr: (scalar)
    """
    m = len(y)
    incorrect = 0
    for i in range(m):
        if yhat[i] != y[i]:
            incorrect += 1

    cerr = incorrect / m
    return(cerr)

y_hat = np.array([1, 2, 0])
y_tmp = np.array([1, 2, 3])
print(f"categorization error {np.squeeze(eval_cat_err(y_hat, y_tmp)):0.3f}, expected:0.333" )
```

Explanation:

- Counts number of incorrect predictions
- Divides by total number of examples
- Returns fraction of misclassified examples

---

7. Model Complexity Analysis

Complex Model Implementation:

Exercise 3 Code:

python

```python
tf.random.set_seed(1234)
model = Sequential(
    [
        Dense(120, activation='relu'),
        Dense(40, activation='relu'),
        Dense(6, activation='linear')
    ], name="Complex"
)
model.compile(
    loss=SparseCategoricalCrossentropy(from_logits=True),
    optimizer=Adam(learning_rate=0.01)
)

model.fit(
    X_train, y_train,
    epochs=1000
)
```

Model Architecture:

- **Layer 1**: 120 neurons with ReLU activation
- **Layer 2**: 40 neurons with ReLU activation
- **Layer 3**: 6 neurons with linear activation (output layer)
- **Total Parameters**: 5,446

Performance Results:

python

```python
training_cerr_complex = eval_cat_err(y_train, model_predict(X_train))
cv_cerr_complex = eval_cat_err(y_cv, model_predict(X_cv))
print(f"categorization error, training, complex model: {training_cerr_complex:0.3f}")
print(f"categorization error, cv,       complex model: {cv_cerr_complex:0.3f}")
```

Output:

```
categorization error, training, complex model: 0.003
categorization error, cv,       complex model: 0.122
```

This shows **overfitting** - very low training error but high cross-validation error.

Simple Model Implementation:

Exercise 4 Code:

```python
tf.random.set_seed(1234)
model_s = Sequential(
    [
        Dense(6, activation='relu'),
        Dense(6, activation='linear')
    ], name = "Simple"
)
model_s.compile(
    loss=SparseCategoricalCrossentropy(from_logits=True),
    optimizer=Adam(learning_rate=0.01)
)

model_s.fit(
    X_train,y_train,
    epochs=1000
)
```

Model Architecture:

- **Layer 1**: 6 neurons with ReLU activation
- **Layer 2**: 6 neurons with linear activation
- **Total Parameters**: 60

Performance Results:

```python
training_cerr_simple = eval_cat_err(y_train, model_predict_s(X_train))
cv_cerr_simple = eval_cat_err(y_cv, model_predict_s(X_cv))
print(f"categorization error, training, simple model, {training_cerr_simple:0.3f}, complex mode
print(f"categorization error, cv,       simple model, {cv_cerr_simple:0.3f}, complex model: {cv
```

Output:

```
categorization error, training, simple model, 0.062, complex model: 0.003
categorization error, cv,       simple model, 0.087, complex model: 0.122
```

The simple model has better generalization (lower cross-validation error).

Regularized Model Implementation:

Exercise 5 Code:

```python
tf.random.set_seed(1234)
model_r = Sequential(
    [
        Dense(120, activation='relu', kernel_regularizer=tf.keras.regularizers.l2(0.1)),
        Dense(40, activation='relu', kernel_regularizer=tf.keras.regularizers.l2(0.1)),
        Dense(6, activation='linear')
    ], name= "Complex-Regularized"
)
model_r.compile(
    loss=SparseCategoricalCrossentropy(from_logits=True),
    optimizer=Adam(learning_rate=0.01)
)

model_r.fit(
    X_train, y_train,
    epochs=1000
)
```

Key Features:

- Same architecture as complex model
- L2 regularization with strength 0.1 on first two layers
- Prevents overfitting by penalizing large weights

Performance Results:

```python
training_cerr_reg = eval_cat_err(y_train, model_predict_r(X_train))
cv_cerr_reg = eval_cat_err(y_cv, model_predict_r(X_cv))
test_cerr_reg = eval_cat_err(y_test, model_predict_r(X_test))
print(f"categorization error, training, regularized: {training_cerr_reg:0.3f}, simple model, {t
print(f"categorization error, cv,       regularized: {cv_cerr_reg:0.3f}, simple model, {cv_cerr
```

Output:

```
categorization error, training, regularized: 0.072, simple model, 0.062, complex model:
0.003
categorization error, cv,      regularized: 0.066, simple model, 0.087, complex model:
0.122
```

The regularized model achieves the best balance between training and cross-validation performance.

Optimal Regularization Search:

python

```python
tf.random.set_seed(1234)
lambdas = [0.0, 0.001, 0.01, 0.05, 0.1, 0.2, 0.3]
models=[None] * len(lambdas)

for i in range(len(lambdas)):
    lambda_ = lambdas[i]
    models[i] =  Sequential(
        [
            Dense(120, activation = 'relu', kernel_regularizer=tf.keras.regularizers.l2(lambda_
            Dense(40, activation = 'relu', kernel_regularizer=tf.keras.regularizers.l2(lambda_)
            Dense(classes, activation = 'linear')
        ]
    )
    models[i].compile(
        loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
        optimizer=tf.keras.optimizers.Adam(0.01),
    )

    models[i].fit(
        X_train,y_train,
        epochs=1000
    )
    print(f"Finished lambda = {lambda_}")
```

Explanation:

- Tests multiple regularization strengths
- Finds optimal lambda where training and cross-validation performance converge
- Lambda > 0.01 appears to be reasonable for this dataset

Key Concepts Summary

1. **Data Splitting**

- Essential for unbiased model evaluation
- Three-way split prevents overfitting during model selection
- Typical distribution: 60% training, 20% cross-validation, 20% test

2. **Overfitting vs Underfitting**

- **Overfitting**: Low training error, high validation error (high variance)
- **Underfitting**: High training error, high validation error (high bias)
- **Good fit**: Balanced training and validation errors

3. **Regularization**

- Prevents overfitting by penalizing model complexity
- L2 regularization adds squared parameter penalty
- Strength controlled by lambda hyperparameter

4. **Model Selection**

- Use cross-validation error to select optimal hyperparameters
- Test set provides final, unbiased performance estimate
- Balance between model complexity and generalization

5. **Neural Network Design**

- More parameters increase model capacity but risk overfitting
- Regularization helps complex models generalize better
- Simple models may generalize better with limited data

This comprehensive analysis demonstrates the iterative process of machine learning model development, from data preparation through model evaluation and optimization.