Student Notebook - Machine Learning Theory & Implementation

Student Name: John Doe

Date: January 10, 2026

Course: Introduction to Machine Learning

Question 1: What is Machine Learning? Explain the main types of machine learning with examples.

Answer:

Machine Learning is a subset of artificial intelligence that enables computer systems to learn and improve from experience without being explicitly programmed. It focuses on developing algorithms that can access data and use it to learn for themselves.

The main types of machine learning are:

1. **Supervised Learning**: In this approach, the algorithm learns from labeled training data. The model is trained on input-output pairs and learns to map inputs to correct outputs. Common examples include classification (predicting categories) and regression (predicting continuous values).

Example code for supervised learning:

```python
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split

# Supervised learning example - Linear Regression
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
model = LinearRegression()
model.fit(X_train, y_train)
predictions = model.predict(X_test)
```

2. **Unsupervised Learning**: Here, the algorithm works with unlabeled data and tries to find hidden patterns or structures. Common techniques include clustering and dimensionality reduction.
3. **Reinforcement Learning**: This type involves an agent learning to make decisions by interacting with an environment. The agent receives rewards or penalties based on its actions.

Question 2: Explain the bias-variance tradeoff and demonstrate it with code.

Answer:

The bias-variance tradeoff is a fundamental concept that describes the tradeoff between two sources of error that affect model performance.

Bias refers to errors from overly simplistic assumptions in the learning algorithm. High bias causes underfitting - the model is too simple to capture the underlying pattern.

Variance refers to errors from sensitivity to small fluctuations in the training data. High variance causes overfitting - the model learns noise rather than the actual signal.

Here's code demonstrating the tradeoff:

```python
import numpy as np
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import Ridge
from sklearn.metrics import mean_squared_error

# Generate sample data
X = np.linspace(0, 10, 100).reshape(-1, 1)
y = 2 * X + np.sin(X) + np.random.normal(0, 0.5, X.shape)

# High bias (underfitting) - degree 1
poly_1 = PolynomialFeatures(degree=1)
X_poly_1 = poly_1.fit_transform(X)
model_1 = Ridge(alpha=1.0)
model_1.fit(X_poly_1, y)

# Balanced - degree 3
poly_3 = PolynomialFeatures(degree=3)
X_poly_3 = poly_3.fit_transform(X)
model_3 = Ridge(alpha=1.0)
model_3.fit(X_poly_3, y)

# High variance (overfitting) - degree 15
poly_15 = PolynomialFeatures(degree=15)
X_poly_15 = poly_15.fit_transform(X)
model_15 = Ridge(alpha=0.001)
model_15.fit(X_poly_15, y)
```

The goal is to find the optimal balance where total error is minimized. This is typically achieved through techniques like cross-validation and regularization.

Question 3: What is overfitting and how can it be prevented? Provide code examples.

Answer:

Overfitting occurs when a machine learning model learns the training data too well, including its noise and outliers, rather than learning the general pattern. The model performs excellently on training data but fails to generalize to new, unseen data.

Prevention techniques include regularization, cross-validation, and early stopping.

Example with regularization:

```python
from sklearn.linear_model import Lasso, Ridge
from sklearn.model_selection import cross_val_score

# Without regularization - prone to overfitting
model_no_reg = LinearRegression()
model_no_reg.fit(X_train, y_train)

# L2 Regularization (Ridge)
ridge_model = Ridge(alpha=10.0)
ridge_model.fit(X_train, y_train)

# L1 Regularization (Lasso)
lasso_model = Lasso(alpha=1.0)
lasso_model.fit(X_train, y_train)

# Cross-validation to check generalization
cv_scores = cross_val_score(ridge_model, X, y, cv=5,
                            scoring='neg_mean_squared_error')
print(f"CV Score: {-cv_scores.mean()}")
```

Early stopping example:

```python
```

```python
# Early stopping in training loop
best_val_loss = float('inf')
patience = 10
counter = 0

for epoch in range(100):
    train_loss = train_epoch(model, train_loader)
    val_loss = validate(model, val_loader)

    if val_loss < best_val_loss:
        best_val_loss = val_loss
        counter = 0
    else:
        counter += 1
        if counter >= patience:
            print("Early stopping triggered")
            break
```

Question 4: Describe precision and recall with a code implementation.

Answer:

Precision and recall are evaluation metrics for classification models, particularly important in imbalanced datasets.

Precision = True Positives / (True Positives + False Positives) **Recall** = True Positives / (True Positives + False Negatives)

Implementation:

```python
```

```python
from sklearn.metrics import precision_score, recall_score, f1_score
from sklearn.metrics import confusion_matrix, classification_report

# Make predictions
y_pred = model.predict(X_test)

# Calculate metrics
precision = precision_score(y_test, y_pred, average='binary')
recall = recall_score(y_test, y_pred, average='binary')
f1 = f1_score(y_test, y_pred, average='binary')

print(f"Precision: {precision:.3f}")
print(f"Recall: {recall:.3f}")
print(f"F1-Score: {f1:.3f}")

# Confusion matrix
cm = confusion_matrix(y_test, y_pred)
print("\nConfusion Matrix:")
print(cm)

# Detailed report
print("\nClassification Report:")
print(classification_report(y_test, y_pred))
```

There's often a tradeoff between precision and recall. The F1-score provides a balanced measure combining both metrics. In some applications, you might want to adjust the decision threshold to favor one over the other.

Question 5: Explain gradient descent and implement it from scratch.

Answer:

Gradient descent is an optimization algorithm used to minimize the cost function in machine learning models by iteratively adjusting model parameters.

The algorithm works by:

1. Starting with random initial parameters
2. Computing the gradient of the cost function
3. Updating parameters in the opposite direction of the gradient
4. Repeating until convergence

Update rule: $\theta = \theta - \alpha \times \nabla J(\theta)$

Implementation from scratch:

python

```python
import numpy as np

def gradient_descent(X, y, learning_rate=0.01, iterations=1000):
    """
    Implement gradient descent for linear regression
    """
    m, n = X.shape
    theta = np.zeros(n)  # Initialize parameters
    cost_history = []

    for i in range(iterations):
        # Predictions
        predictions = X.dot(theta)

        # Calculate error
        errors = predictions - y

        # Calculate gradient
        gradient = (1/m) * X.T.dot(errors)

        # Update parameters
        theta = theta - learning_rate * gradient

        # Calculate cost
        cost = (1/(2*m)) * np.sum(errors**2)
        cost_history.append(cost)

        if i % 100 == 0:
            print(f"Iteration {i}: Cost = {cost:.4f}")

    return theta, cost_history

# Usage example
X = np.random.randn(100, 3)
y = 2 * X[:, 0] + 3 * X[:, 1] - X[:, 2] + np.random.randn(100) * 0.1

# Add bias term
X = np.c_[np.ones(X.shape[0]), X]

# Run gradient descent
theta_final, costs = gradient_descent(X, y, learning_rate=0.1, iterations=1000)
print(f"\nFinal parameters: {theta_final}")
```

The learning rate is crucial - too small leads to slow convergence, too large might overshoot the minimum.

Question 6: Explain neural networks and build a simple one using code.

Answer:

A neural network is a computational model inspired by biological neural networks in the brain. It consists of interconnected nodes (neurons) organized in layers that process information.

Basic components:

- **Input Layer**: Receives raw input features
- **Hidden Layers**: Intermediate layers that learn representations
- **Output Layer**: Produces final predictions
- **Weights and Biases**: Learnable parameters
- **Activation Functions**: Non-linear functions (ReLU, sigmoid, tanh)

Simple implementation:

```python
```

```python
import numpy as np
from sklearn.neural_network import MLPClassifier
from sklearn.datasets import make_classification

# Generate sample data
X, y = make_classification(n_samples=1000, n_features=20,
                           n_classes=2, random_state=42)

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

# Create neural network
model = MLPClassifier(hidden_layer_sizes=(64, 32),
                      activation='relu',
                      solver='adam',
                      max_iter=500,
                      random_state=42)

# Train the model
model.fit(X_train, y_train)

# Evaluate
train_acc = model.score(X_train, y_train)
test_acc = model.score(X_test, y_test)

print(f"Training Accuracy: {train_acc:.3f}")
print(f"Test Accuracy: {test_acc:.3f}")
```

Building from scratch with backpropagation:

```python
```

```
class SimpleNeuralNetwork:
    def __init__(self, input_size, hidden_size, output_size):
        # 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))

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

    def forward(self, X):
        # Hidden layer
        self.z1 = X.dot(self.W1) + self.b1
        self.a1 = self.sigmoid(self.z1)

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

        return self.a2

    def backward(self, X, y, learning_rate=0.01):
        m = X.shape[0]

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

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

        # Update weights
        self.W1 -= learning_rate * dW1
        self.b1 -= learning_rate * db1
        self.W2 -= learning_rate * dW2
        self.b2 -= learning_rate * db2

# Create and train
nn = SimpleNeuralNetwork(input_size=20, hidden_size=10, output_size=1)
# Training loop would go here
```

The network learns through backpropagation, which calculates gradients and updates weights using gradient descent.

End of Notebook