Lab Session: Simple Neurons and Perceptrons

Master's Course in Deep Learning
Department of Computer Science

Semester 2.2026

Duration: 3 hours
Prerequisites: Basic Python programming, Linear Algebra fundamentals
Tools: Python 3.x, NumPy, Matplotlib, scikit-learn

Abstract

This laboratory session introduces fundamental concepts of artificial neurons, starting from the McCulloch-Pitts model to Rosenblatt's perceptron. Students will implement these models from scratch, visualize decision boundaries, and explore their capabilities and limitations. The session lays the groundwork for understanding modern neural networks.

Contents

1 Introduction and Learning Objectives **3**
- 1.1 Background . . . 3
- 1.2 Learning Objectives . . . 3

2 Part 1: Biological Neuron and McCulloch-Pitts Model **3**
- 2.1 Theoretical Background . . . 3
- 2.2 Practical Implementation . . . 4
- 2.3 Discussion Questions . . . 5

3 Part 2: Perceptron Learning Algorithm **5**
- 3.1 Theoretical Foundation . . . 5
- 3.2 Implementation . . . 6
- 3.3 Visualization Functions . . . 8
- 3.4 Experiments with Parameters . . . 9

4 Part 3: Limitations and Advanced Topics **11**
- 4.1 The XOR Problem . . . 11
- 4.2 Comparison with Logistic Regression . . . 12
- 4.3 Extensions and Modifications . . . 12

5 Assessment and Deliverables **13**
5.1 Lab Report Requirements 13
5.2 Discussion Questions for Report 13

6 Additional Resources **14**
6.1 Recommended Reading 14
6.2 Online Resources 14
6.3 Future Directions 14

1 Introduction and Learning Objectives

1.1 Background

Artificial neural networks draw inspiration from biological neural systems. The journey begins with simple mathematical models that mimic neuron behavior:

- **1943:** McCulloch-Pitts neuron – First mathematical model of a biological neuron
- **1958:** Rosenblatt's perceptron – First learning algorithm for artificial neurons
- **1969:** Minsky & Papert – Identified limitations (XOR problem)

1.2 Learning Objectives

By the end of this session, students should be able to:

1. Explain the biological inspiration for artificial neurons
2. Implement the McCulloch-Pitts neuron model
3. Implement and train a single-layer perceptron
4. Visualize decision boundaries and convergence
5. Understand the limitations of linear classifiers
6. Analyze the effect of different learning parameters

2 Part 1: Biological Neuron and McCulloch-Pitts Model

2.1 Theoretical Background

The biological neuron consists of:

- **Dendrites:** Receive signals from other neurons
- **Soma:** Cell body that processes inputs
- **Axon:** Transmits output signals
- **Synapses:** Connections between neurons

The McCulloch-Pitts (1943) neuron formalizes this as:

$$y = f\left(\sum_{i=1}^{n} w_i x_i - \theta\right)$$

where:

- x_i are binary inputs (0 or 1)
- w_i are binary weights (-1, 0, or 1)
- θ is the threshold
- f is the step function: $f(z) = 1$ if $z \geq 0$, else 0

2.2 Practical Implementation

Exercise 1.1: McCulloch-Pitts Neuron Implementation

Implement a McCulloch-Pitts neuron that can simulate logic gates.

```python
import numpy as np

class MPNeuron:
    """McCulloch-Pitts Neuron Model"""
    def __init__(self, threshold):
        self.threshold = threshold

    def activate(self, inputs, weights):
        """
        Parameters:
        inputs: array-like, binary inputs [0, 1]
        weights: array-like, connection weights

        Returns:
        output: 0 or 1
        """
        # Calculate weighted sum
        weighted_sum = np.dot(inputs, weights)

        # Apply threshold activation
        return 1 if weighted_sum >= self.threshold else 0

# Test with logic gates
def test_logic_gates():
    """Demonstrate logic gate implementation"""

    # AND Gate: Output 1 only if both inputs are 1
    print("=== AND Gate ===")
    neuron = MPNeuron(threshold=2)
    weights = [1, 1]

    truth_table = [(0,0), (0,1), (1,0), (1,1)]
    for inputs in truth_table:
        output = neuron.activate(inputs, weights)
        print(f"Input: {inputs} -> Output: {output}")

    # OR Gate: Output 1 if at least one input is 1
    print("\n=== OR Gate ===")
    neuron.threshold = 1  # Change only threshold

    for inputs in truth_table:
        output = neuron.activate(inputs, weights)
        print(f"Input: {inputs} -> Output: {output}")

    # NOT Gate (single input)
    print("\n=== NOT Gate ===")
    neuron = MPNeuron(threshold=0)
    weights = [-1]  # Negative weight

    for input_val in [0, 1]:
        output = neuron.activate([input_val], weights)
```

```python
        print(f"Input: {input_val} -> Output: {output}")

if __name__ == "__main__":
    test_logic_gates()
```

Listing 1: McCulloch-Pitts Neuron Implementation

Expected Output

```
=== AND Gate ===
Input: (0, 0) -> Output: 0
Input: (0, 1) -> Output: 0
Input: (1, 0) -> Output: 0
Input: (1, 1) -> Output: 1

=== OR Gate ===
Input: (0, 0) -> Output: 0
Input: (0, 1) -> Output: 1
Input: (1, 0) -> Output: 1
Input: (1, 1) -> Output: 1

=== NOT Gate ===
Input: 0 -> Output: 1
Input: 1 -> Output: 0
```

2.3 Discussion Questions

1. What logic functions can a single MP neuron implement?
2. Can a single MP neuron implement XOR? Why or why not?
3. How does changing the threshold affect the neuron's behavior?
4. What are the limitations of fixed weights?

3 Part 2: Perceptron Learning Algorithm

3.1 Theoretical Foundation

The perceptron (Rosenblatt, 1958) introduces **learnable weights**:

$$
\begin{aligned}
\text{Output:} \quad & y = f(\mathbf{w} \cdot \mathbf{x} + b) \\
\text{where:} \quad & f(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{otherwise} \end{cases} \\
\text{Update rule:} \quad & \Delta w_i = \eta(t - y)x_i \\
& \Delta b = \eta(t - y)
\end{aligned}
$$

where:

- η: Learning rate ($0 < \eta \leq 1$)
- t: Target output
- y: Predicted output
- x_i: Input feature

Perceptron Convergence Theorem: If the data is linearly separable, the perceptron will converge to a solution in finite time.

3.2 Implementation

Exercise 2.1: Perceptron Class Implementation

Implement a perceptron with the learning algorithm.

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs
from sklearn.model_selection import train_test_split

class Perceptron:
    """Single Layer Perceptron"""

    def __init__(self, learning_rate=0.01, n_iters=100, random_state
=42):
        """
        Parameters:
        learning_rate: float, step size for weight updates
        n_iters: int, maximum number of training iterations
        random_state: int, random seed for reproducibility
        """
        self.lr = learning_rate
        self.n_iters = n_iters
        self.random_state = random_state
        self.weights = None
        self.bias = None
        self.errors = []  # Track errors per epoch
        self.converged = False

    def initialize_weights(self, n_features):
        """Initialize weights with small random values"""
        np.random.seed(self.random_state)
        self.weights = np.random.randn(n_features) * 0.01
        self.bias = np.random.randn() * 0.01

    def activation(self, x):
        """Step activation function"""
        return np.where(x >= 0, 1, 0)

    def fit(self, X, y):
        """
        Train the perceptron

        Parameters:
```

```
        X: array-like, shape (n_samples, n_features)
        y: array-like, shape (n_samples,), binary labels {0, 1}
        """
        n_samples, n_features = X.shape
        self.initialize_weights(n_features)

        # Convert y to numpy array if needed
        y = np.array(y).flatten()

        for epoch in range(self.n_iters):
            epoch_errors = 0

            for idx in range(n_samples):
                # Forward pass
                linear_output = np.dot(X[idx], self.weights) + self.bias
                y_pred = self.activation(linear_output)

                # Compute error
                error = y[idx] - y_pred

                # Update weights if error != 0
                if error != 0:
                    self.weights += self.lr * error * X[idx]
                    self.bias += self.lr * error
                    epoch_errors += 1

            # Record errors for this epoch
            self.errors.append(epoch_errors)

            # Check for convergence
            if epoch_errors == 0:
                print(f"Converged at epoch {epoch+1}")
                self.converged = True
                break

        if not self.converged:
            print(f"Did not converge after {self.n_iters} iterations")

    def predict(self, X):
        """Make predictions"""
        linear_output = np.dot(X, self.weights) + self.bias
        return self.activation(linear_output)

    def score(self, X, y):
        """Calculate accuracy"""
        predictions = self.predict(X)
        accuracy = np.mean(predictions == y)
        return accuracy

# Generate synthetic data
def generate_data():
    """Create linearly separable dataset"""
    X, y = make_blobs(
        n_samples=200,
        centers=2,
        n_features=2,
        cluster_std=1.5,
```

```python
        random_state=42
    )
    y = np.where(y == 0, 0, 1)  # Convert to binary labels
    return X, y

# Example usage
X, y = generate_data()
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# Initialize and train perceptron
perceptron = Perceptron(learning_rate=0.1, n_iters=50)
perceptron.fit(X_train, y_train)

# Evaluate
train_acc = perceptron.score(X_train, y_train)
test_acc = perceptron.score(X_test, y_test)
print(f"Training accuracy: {train_acc:.2%}")
print(f"Test accuracy: {test_acc:.2%}")
```

Listing 2: Perceptron Implementation

3.3 Visualization Functions

```python
def plot_decision_boundary(X, y, perceptron, title="Perceptron Decision
    Boundary"):
    """Plot decision boundary and data points"""

    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 5))

    # Plot 1: Decision Boundary
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1

    # Create mesh grid
    xx, yy = np.meshgrid(
        np.arange(x_min, x_max, 0.1),
        np.arange(y_min, y_max, 0.1)
    )

    # Predict for each mesh point
    Z = perceptron.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)

    # Plot contour and scatter
    ax1.contourf(xx, yy, Z, alpha=0.3, cmap='coolwarm')
    scatter = ax1.scatter(X[:, 0], X[:, 1], c=y,
                          edgecolors='k', cmap='coolwarm')
    ax1.set_xlabel('Feature 1')
    ax1.set_ylabel('Feature 2')
    ax1.set_title(title)

    # Add legend
    legend1 = ax1.legend(*scatter.legend_elements(),
                         title="Classes")
    ax1.add_artist(legend1)
```

```python

    # Plot decision boundary line
    if perceptron.weights is not None:
        # For 2D: w1*x1 + w2*x2 + b = 0
        # Solve for x2: x2 = (-w1*x1 - b) / w2
        w1, w2 = perceptron.weights
        b = perceptron.bias

        # Generate line points
        x_line = np.array([x_min, x_max])
        y_line = (-w1 * x_line - b) / w2
        ax1.plot(x_line, y_line, 'k--', linewidth=2,
                label='Decision Boundary')
        ax1.legend()

    # Plot 2: Learning Curve
    ax2.plot(range(1, len(perceptron.errors) + 1),
            perceptron.errors, marker='o', linewidth=2)
    ax2.set_xlabel('Epoch')
    ax2.set_ylabel('Number of Misclassifications')
    ax2.set_title('Perceptron Learning Curve')
    ax2.grid(True, alpha=0.3)

    # Highlight convergence point
    if perceptron.converged:
        conv_epoch = len(perceptron.errors)
        ax2.axvline(x=conv_epoch, color='r', linestyle='--',
                   alpha=0.5, label=f'Convergence (epoch {conv_epoch})')
        ax2.legend()

    plt.tight_layout()
    plt.show()

def plot_weight_evolution(perceptron, X_train, y_train):
    """Plot weight evolution during training (simulated)"""
    # Note: This requires modifying the perceptron to save weight history
    pass

# Generate and plot
X, y = generate_data()
perceptron = Perceptron(learning_rate=0.1, n_iters=30)
perceptron.fit(X, y)
plot_decision_boundary(X, y, perceptron)
```

Listing 3: Visualization Functions

3.4 Experiments with Parameters

Exercise 2.2: Learning Rate Experiment

Investigate the effect of different learning rates.

```python
def experiment_learning_rates(X, y):
    """Compare different learning rates"""
```

```python

    learning_rates = [0.001, 0.01, 0.1, 0.5, 1.0]
    results = []

    fig, axes = plt.subplots(2, 3, figsize=(15, 8))
    axes = axes.flatten()

    for idx, lr in enumerate(learning_rates):
        # Train perceptron
        perceptron = Perceptron(learning_rate=lr, n_iters=50)
        perceptron.fit(X, y)

        # Record results
        results.append({
            'lr': lr,
            'final_errors': perceptron.errors[-1] if perceptron.errors else None,
            'converged': perceptron.converged,
            'epochs': len(perceptron.errors)
        })

        # Plot learning curve
        ax = axes[idx]
        ax.plot(perceptron.errors, marker='o', markersize=4)
        ax.set_title(f'LR = {lr}\nConverged: {perceptron.converged}')
        ax.set_xlabel('Epoch')
        ax.set_ylabel('Errors')
        ax.grid(True, alpha=0.3)

        # Mark convergence
        if perceptron.converged:
            conv_epoch = len(perceptron.errors)
            ax.axvline(x=conv_epoch-1, color='r', linestyle='--', alpha=0.5)

    # Hide unused subplot
    if len(learning_rates) < len(axes):
        axes[-1].axis('off')

    plt.tight_layout()
    plt.show()

    # Print summary table
    print("Summary of Learning Rate Experiments:")
    print("-" * 60)
    print(f"{'Learning Rate':<15} {'Converged':<10} {'Epochs':<10} {'Final Errors':<15}")
    print("-" * 60)
    for res in results:
        print(f"{res['lr']:<15.3f} {str(res['converged']):<10} "
              f"{res['epochs']:<10} {res['final_errors']:<15}")

# Run experiment
X, y = generate_data()
experiment_learning_rates(X, y)
```

Listing 4: Learning Rate Experiment

4 Part 3: Limitations and Advanced Topics

4.1 The XOR Problem

Fundamental Limitation

A single-layer perceptron cannot solve non-linearly separable problems like XOR.

```
def xor_experiment():
    """Demonstrate perceptron's failure on XOR"""

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

    # Try to learn XOR
    print("=== XOR Problem ===")
    perceptron = Perceptron(learning_rate=0.1, n_iters=100)
    perceptron.fit(X_xor, y_xor)

    # Show predictions
    print("\nPredictions:")
    print("-" * 40)
    print(f"{'Input':<10} {'True':<10} {'Predicted':<10} {'Correct':<10}")
    print("-" * 40)

    all_correct = True
    for x, y_true in zip(X_xor, y_xor):
        y_pred = perceptron.predict(x.reshape(1, -1))[0]
        correct = y_true == y_pred
        if not correct:
            all_correct = False
        print(f"{str(x):<10} {y_true:<10} {y_pred:<10} {str(correct):<10}")

    print("-" * 40)
    print(f"All correct: {all_correct}")
    print(f"Final errors per epoch: {perceptron.errors}")

    # Visualize (will show failure)
    plot_decision_boundary(X_xor, y_xor, perceptron,
                           "Perceptron on XOR (Cannot Separate)")

    return all_correct

# Run XOR experiment
xor_success = xor_experiment()
if not xor_success:
    print("\n" + "="*60)
    print("CONCLUSION: Single-layer perceptron CANNOT learn XOR!")
    print("This demonstrates the need for multi-layer networks.")
    print("="*60)
```

Listing 5: XOR Problem Demonstration

4.2 Comparison with Logistic Regression

Perceptron vs. Logistic Regression

- **Perceptron:** Step activation, minimizes misclassifications
- **Logistic Regression:** Sigmoid activation, minimizes log-loss
- **Similarity:** Both are linear classifiers
- **Difference:** Perceptron provides binary outputs, LR provides probabilities

4.3 Extensions and Modifications

Challenge Exercises

Try implementing these extensions:

1. **Pocket Algorithm:** Keep the best weights encountered during training
2. **Voted Perceptron:** Weight predictions by how long weights survived
3. **Kernel Perceptron:** Add kernel trick for non-linear separation
4. **Multi-class Perceptron:** Extend to multiple classes using one-vs-all

```
class PocketPerceptron(Perceptron):
    """Perceptron with Pocket Algorithm"""

    def __init__(self, learning_rate=0.01, n_iters=100):
        super().__init__(learning_rate, n_iters)
        self.best_weights = None
        self.best_bias = None
        self.best_score = -np.inf

    def fit(self, X, y):
        n_samples, n_features = X.shape
        self.initialize_weights(n_features)

        # Initialize best weights with initial weights
        self.best_weights = self.weights.copy()
        self.best_bias = self.bias
        self.best_score = self.score(X, y)

        for epoch in range(self.n_iters):
            epoch_errors = 0

            for idx in range(n_samples):
                linear_output = np.dot(X[idx], self.weights) + self.
    bias
                y_pred = self.activation(linear_output)

                error = y[idx] - y_pred
                if error != 0:
                    self.weights += self.lr * error * X[idx]
```

```
                    self.bias += self.lr * error
                    epoch_errors += 1

                    # Check if current weights are better
                    current_score = self.score(X, y)
                    if current_score > self.best_score:
                        self.best_weights = self.weights.copy()
                        self.best_bias = self.bias
                        self.best_score = current_score

            self.errors.append(epoch_errors)

            if epoch_errors == 0:
                break

    def predict(self, X):
        """Use best weights for prediction"""
        linear_output = np.dot(X, self.best_weights) + self.best_bias
        return self.activation(linear_output)
```

Listing 6: Pocket Perceptron Extension

5 Assessment and Deliverables

5.1 Lab Report Requirements

Submit a lab report containing:

1. **Introduction:** Brief background and objectives
2. **Implementation:** Your complete code with comments
3. **Results:**
    - Screenshots of decision boundaries
    - Learning curves for different parameters
    - XOR problem demonstration
4. **Discussion:** Answers to all discussion questions
5. **Conclusion:** Summary of findings and insights

5.2 Discussion Questions for Report

1. Explain why the perceptron cannot learn the XOR function. What architectural change would solve this?
2. How does the learning rate affect convergence? What happens if it's too high or too low?
3. Compare the McCulloch-Pitts neuron with Rosenblatt's perceptron. What key innovation made learning possible?

4. The perceptron convergence theorem guarantees convergence for linearly separable data. Why is this both a strength and a limitation?

5. How would you extend this perceptron to handle:
    - Multi-class classification (more than 2 classes)?
    - Non-linear decision boundaries without adding layers?

6 Additional Resources

6.1 Recommended Reading

- McCulloch, W. S., & Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics.*
- Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review.*
- Minsky, M., & Papert, S. (1969). *Perceptrons.* MIT Press.
- Chapter 1 of Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning.* MIT Press.

6.2 Online Resources

- TensorFlow Playground – Interactive neural network visualization
- scikit-learn Documentation – Machine learning library
- Python Machine Learning Book – Code examples

6.3 Future Directions

This lab serves as foundation for:

- Multi-layer perceptrons (MLPs)
- Backpropagation algorithm
- Deep neural networks
- Convolutional neural networks (CNNs)
- Recurrent neural networks (RNNs)

Appendix: Quick Reference

Key Formulas

- **Neuron output:** $y = f(\sum w_i x_i + b)$
- **Step function:** $f(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{otherwise} \end{cases}$
- **Weight update:** $\Delta w_i = \eta(t - y)x_i$
- **Bias update:** $\Delta b = \eta(t - y)$

Common Issues and Solutions

Issue	**Probable Cause**	**Solution**
No convergence	Data not linearly separable	Check data or use kernel/multi-layer
Oscillating weights	Learning rate too high	Decrease learning rate
Slow convergence	Learning rate too low	Increase learning rate
Poor accuracy	Initialization	Use random initialization

Table 1: Troubleshooting Guide

Python Package Requirements

```
numpy>=1.19.0
matplotlib>=3.3.0
scikit-learn>=0.24.0
jupyter>=1.0.0  # Optional, for notebook
```

End of Lab Session