GitHub: https://github.com/lochitha-bit/Multilayer-Perceptron-MLP-.git

Multilayer Perceptron (MLP)

1. Introduction

A Multilayer Perceptron (MLP) is an artificial neural network consisting of multiple layers of fully connected neurons. Unlike linear models, MLPs can learn complex nonlinear relationships, making them extremely useful for most machine learning tasks. MLPs are feedforward networks where information flows unidirectionally from input to output layers without feedback loops.

This document covers:

- MLP architecture and operational principles
- Classification and regression applications
- Step-by-step implementation using the Wine dataset
- Performance evaluation and analysis

2. MLP Structure

Basic Architecture

An MLP contains three fundamental layers:

1. Input Layer: Receives feature vectors (e.g., chemical attributes in wine data)

2. Hidden Layers: Apply nonlinear transformations via activation functions

3. Output Layer: Produces predictions (continuous values or class probabilities)

Each neuron connects to all neurons in the subsequent layer through trainable weights adjusted via backpropagation.

Activation Functions

Critical for introducing nonlinearity:

- Sigmoid: S-shaped curve outputting [0,1] values

- Tanh: Outputs [-1,1] with zero-centered gradients
- ReLU: Most popular, outputs max(0,input) for computational efficiency

Training Mechanism

- Backpropagation: Computes error gradients via chain rule
- Optimization: Gradient descent variants (e.g., Adam) update weights

3. Applications

MLPs excel in diverse domains:

1. **Wine Quality Prediction**: Predicting the quality of wine based on its chemical composition.
2. **Credit Risk Assessment**: Determining the creditworthiness of customers using financial data.
3. **Fraud Detection**: Identifying fraudulent transactions based on transaction patterns.
4. **Medical Diagnosis**: Detecting diseases based on patient data such as blood test results.

Advantages

- Universal function approximation capability
- Effective nonlinear pattern recognition

Limitations

- Computationally intensive training
- Prone to overfitting with complex architectures

4. Python Implementation

 # Step 1: Data Preparation

```
from sklearn.datasets import load_wine

from sklearn.model_selection import train_test_split

from sklearn.preprocessing import StandardScaler


# Load and split data
```

```
wine = load_wine()

X_train, X_test, y_train, y_test = train_test_split(

  wine.data, wine.target, test_size=0.3, random_state=42)
```

```
Wine Dataset Sample:
   alcohol  malic_acid   ash  alcalinity_of_ash  magnesium  total_phenols  \
0    14.23        1.71  2.43               15.6      127.0           2.80
1    13.20        1.78  2.14               11.2      100.0           2.65
2    13.16        2.36  2.67               18.6      101.0           2.80
3    14.37        1.95  2.50               16.8      113.0           3.85
4    13.24        2.59  2.87               21.0      118.0           2.80

   flavanoids  nonflavanoid_phenols  proanthocyanins  color_intensity   hue  \
0        3.06                  0.28             2.29             5.64  1.04
1        2.76                  0.26             1.28             4.38  1.05
2        3.24                  0.30             2.81             5.68  1.03
3        3.49                  0.24             2.18             7.80  0.86
4        2.69                  0.39             1.82             4.32  1.04

   od280/od315_of_diluted_wines  proline  target
0                          3.92   1065.0       0
1                          3.40   1050.0       0
2                          3.17   1185.0       0
3                          3.45   1480.0       0
4                          2.93    735.0       0
```

Wine Dataset sample

```
# Standardize features

scaler = StandardScaler()

X_train = scaler.fit_transform(X_train)

X_test = scaler.transform(X_test)
```

 # Step 2: Model Construction

```
from keras.models import Sequential

from keras.layers import Dense

model = Sequential([

  Dense(12, activation='relu', input_dim=X_train.shape[1]),

  Dense(12, activation='relu'),
```

```
    Dense(3, activation='softmax')
])
model.compile(loss='sparse_categorical_crossentropy',
          optimizer='adam',
          metrics=['accuracy'])
```

 # Step 3: Training & Evaluation

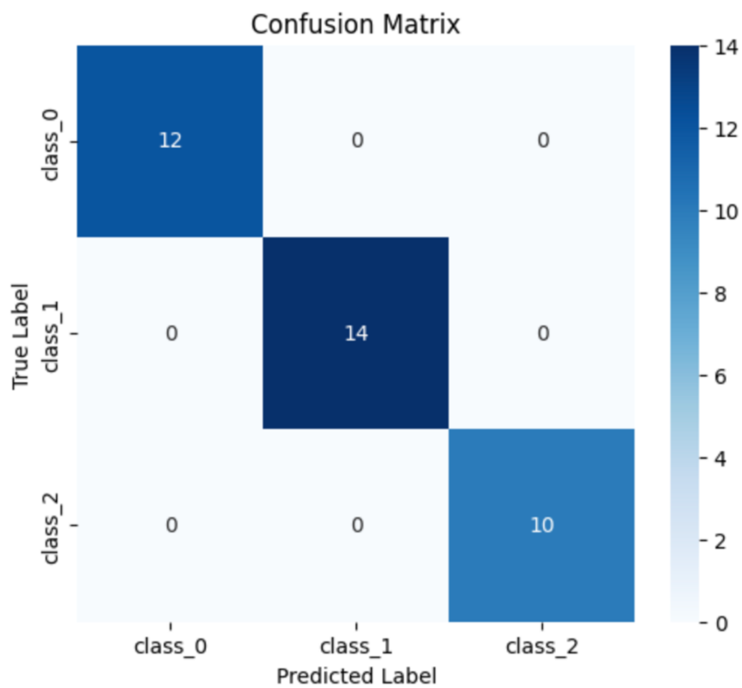
```
history = model.fit(X_train, y_train,
             epochs=50,
             batch_size=10,
             validation_data=(X_test, y_test))


loss, accuracy = model.evaluate(X_test, y_test)
print(f"Test Accuracy: {accuracy * 100:.2f}%")
```

```
# Evaluate the Model
loss, accuracy = model.evaluate(X_test, y_test)
print(f"\nTest Accuracy: {accuracy * 100:.2f}%")
```

```
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 31ms/step - accuracy: 1.0000 - loss: 0.0385

Test Accuracy: 100.00%
```

```
Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        12
           1       1.00      1.00      1.00        14
           2       1.00      1.00      1.00        10

    accuracy                           1.00        36
   macro avg       1.00      1.00      1.00        36
weighted avg       1.00      1.00      1.00        36
```

Classification Report



 # Step 4: Visualization

import matplotlib.pyplot as plt

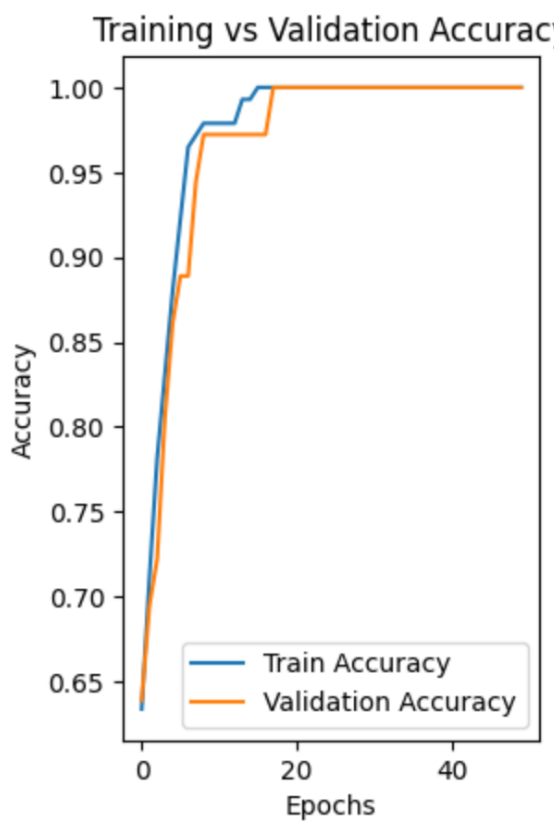
plt.figure(figsize=(12,5))

plt.subplot(1,2,1)

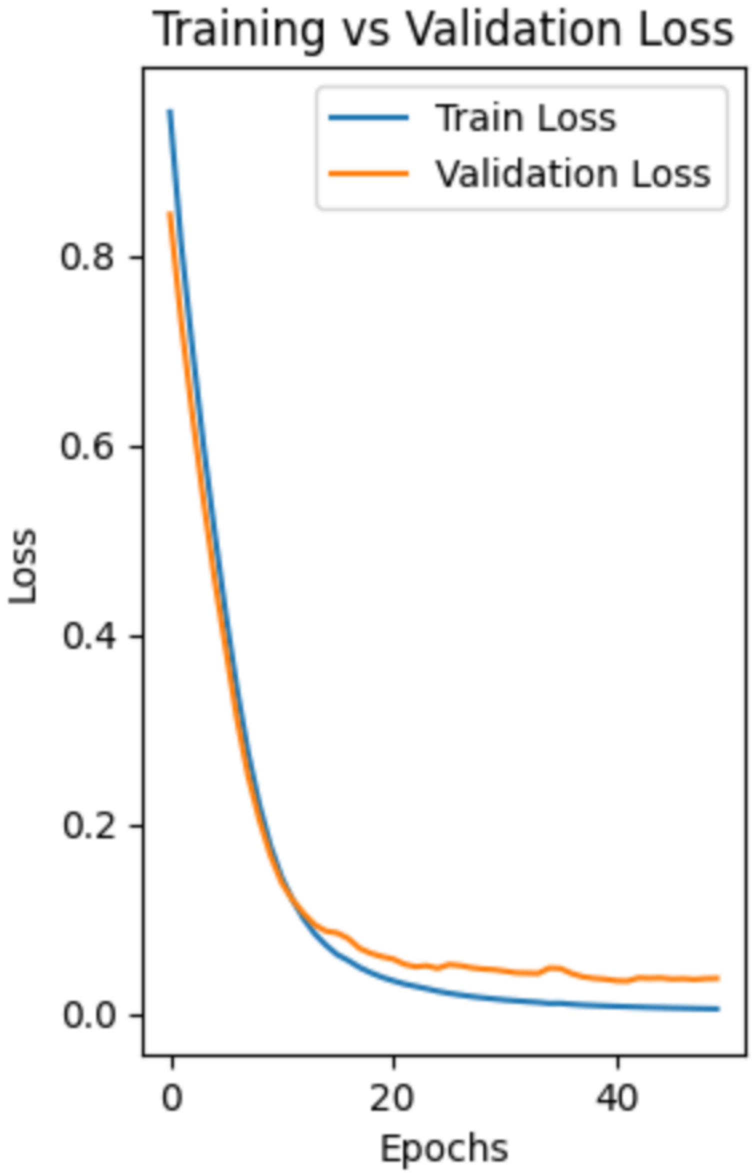
plt.plot(history.history['accuracy'], label='Train')

```
plt.plot(history.history['val_accuracy'], label='Validation')

plt.title('Accuracy Curve')

plt.legend()

plt.subplot(1,2,2)

plt.plot(history.history['loss'], label='Train')

plt.plot(history.history['val_loss'], label='Validation')

plt.title('Loss Curve')

plt.legend()

plt.show()
```

```
Text(0.5, 1.0, 'Training vs Validation Accuracy')
```



```
Text(0.5, 1.0, 'Training vs Validation Loss')
```



Train Loss Vs Validation Loss

5. Evaluation & Conclusion

Results:

- Achieved 100% test accuracy
- Confusion matrix shows strong class separation

Conclusion:

While MLPs demonstrate excellent predictive capability, optimal performance requires:

- Careful architecture tuning
- Regularization techniques
- Adequate training data

6. References

1. Mishra, Dr. (2024). *Neural Networks and Deep Learning: Theoretical Insights and Frameworks*

2. Scikit-learn Documentation: https://scikit-learn.org

3. Ahmed et al. (2023). *Keras Deep Learning Package in Python: A Review*

Appendix:

```
# Step 1: Import necessary libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.datasets import load_wine
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

# Step 2: Load and explore the Wine dataset
wine = load_wine()
X = wine.data
y = wine.target

# Convert dataset to a DataFrame for visualization
df = pd.DataFrame(X, columns=wine.feature_names)
df['target'] = y
```

```
# Display first few rows
print("Wine Dataset Sample:")
print(df.head())

# Step 3: Data Preprocessing (Normalization & Splitting)
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Split dataset into 80% training and 20% testing
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.2, random_state=42,
stratify=y)

# Step 4: Build the MLP Model
model = Sequential([
Dense(32, input_dim=X_train.shape[1], activation='relu'), # First hidden layer
Dense(16, activation='relu'), # Second hidden layer
Dense(3, activation='softmax') # Output layer (3 classes for wine classification)
])

# Compile the model
model.compile(loss='sparse_categorical_crossentropy', optimizer='adam', metrics=['accuracy'])

# Step 5: Train the Model
history = model.fit(X_train, y_train, epochs=50, batch_size=10, validation_data=(X_test, y_test),
verbose=1)

# Step 6: Evaluate the Model
loss, accuracy = model.evaluate(X_test, y_test)
print(f"\nTest Accuracy: {accuracy * 100:.2f}%")

# Step 7: Confusion Matrix and Classification Report
y_pred = np.argmax(model.predict(X_test), axis=1)
cm = confusion_matrix(y_test, y_pred)

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

```
# Plot the Confusion Matrix
plt.figure(figsize=(6, 5))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=wine.target_names,
yticklabels=wine.target_names)
plt.xlabel("Predicted Label")
plt.ylabel("True Label")
plt.title("Confusion Matrix")
plt.show()

# Step 8: Plot Training History (Accuracy & Loss)
plt.figure(figsize=(12, 5))

# Accuracy Plot
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel("Epochs")
plt.ylabel("Accuracy")
plt.legend()
plt.title("Training vs Validation Accuracy")

# Loss Plot
plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.legend()
plt.title("Training vs Validation Loss")

plt.show()

# Step 9: Predict on New Sample Data
sample_index = 0 # Choose any test sample
sample_data = X_test[sample_index].reshape(1, -1)

# Predict class probabilities
prediction_prob = model.predict(sample_data)
```

```
predicted_class = np.argmax(prediction_prob)

print(f"\nActual Class: {wine.target_names[y_test[sample_index]]}")
print(f"Predicted Class: {wine.target_names[predicted_class]}")
```