

```
#####  
#                               #  
# CS435 Generative AI: Security, Ethics and Governance    #  
#                               #  
# Instructor: Dr. Adnan Masood                          #  
# Contact:  adnanmasood@gmail.com                      #  
#                               #  
# Notebook is MIT Licensed                             #  
#####
```

✓ Multilayer Perceptron (MLP) - Comprehensive Explanation

Table of Contents

1. [Introduction](#)
 2. [Building an Intuitive Understanding](#)
 3. [Intuition & History](#)
 4. [Mathematics Behind MLP](#)
 5. [Illustrative Example \(With Code\)](#)
 6. [Example Calculations](#)
 7. [Step-by-Step Implementation from Scratch](#)
 8. [Illustrative Problem MLP Solves](#)
 9. [Practical Real-World Problem](#)
 10. [How to Solve a Real-World Problem Using MLP](#)
 11. [Questions to Ponder](#)
 12. [Answers & Code Examples](#)
 13. [A Sample Exercise](#)
 14. [Glossary](#)
-

1. Introduction

Welcome to our deep dive into **Multilayer Perceptrons (MLPs)**. In this Jupyter notebook, we will explore MLPs from the very basics to advanced concepts, walking through intuition, history, mathematics, code examples, and practical applications. Let's begin!

2. Building an Intuitive Understanding

We will explain MLP at five distinct levels of complexity. Feel free to read through them all or skip to your level of comfort.

- **What is an MLP?**
 - An MLP is like a group of friends solving a puzzle together by passing notes. Each friend does a little work, then hands off the result to the next friend.
- **Why is it important?**
 - MLPs help computers learn patterns—like recognizing digits or understanding if an email is spam.
- **A bit more detail:**
 - An MLP has layers of tiny calculators called neurons. These neurons take numbers, do math, and pass results to the next layer.
- **Goal:**
 - Train the MLP so it can make predictions or decisions (like classifying images).
- **Architecture:**
 - Consists of an input layer, hidden layer(s), and an output layer.
- **Learning:**
 - We "teach" the network by showing examples. We measure errors with a loss function, then adjust the network's parameters (weights and biases) to reduce errors.
- **Components:**
 - Activation functions: Sigmoid, ReLU, Tanh, etc.
 - Optimization algorithms: Gradient Descent, Stochastic Gradient Descent, Adam, etc.
- **Training Procedure:**

- Forward pass, compute loss, backward pass (backpropagation), update parameters.

- **Advanced Topics:**

- Convergence properties, theoretical underpinnings (Universal Approximation Theorem), interpretability, and regularization (L2, Dropout).

- **Research Scope:**

- Investigating ways to improve generalization, reduce overfitting, and design deeper architectures.

3. Intuition & Brief History

Intuition: An MLP is like a decision-making pipeline. Each layer refines the information received from the previous layer. Think of it as a multi-step question-and-answer process:

1. Input layer collects data.
2. Each hidden layer processes, extracts patterns, and compresses/expands relevant features.
3. The output layer makes a final decision or prediction.

Brief History:

- **1943:** McCulloch and Pitts propose the first mathematical model of a neuron.
- **1958:** Perceptron invented by Frank Rosenblatt.
- **1970s:** Introduction of multi-layer networks, but training them was difficult.
- **1986:** Backpropagation popularized by Rumelhart, Hinton, and Williams.
- **2006:** Deep learning renaissance begins with efficient training of deeper networks.

4. Mathematics Behind MLP

An MLP is made up of layers of **neurons**. Each neuron does a simple math operation:

1. Takes inputs x_1, x_2, \dots, x_n .
2. Multiplies each x_i by a **weight** w_i .
3. Adds a **bias** b to that sum.
4. Passes the result through an **activation function** (like a squishing function that keeps numbers in a certain range).

Mathematically:

$$z = w_1x_1 + w_2x_2 + \dots + w_nx_n + b$$

Then,

$$\text{output} = \sigma(z)$$

where σ is an activation function (e.g., $\sigma(z) = \text{ReLU}(z) = \max(0, z)$, or a sigmoid, etc.).

Forward Pass

For each layer, we calculate the output of every neuron in that layer and pass it to the next layer.

Backpropagation

To train, we compare the network's prediction to the **target** and compute a **loss** (or error):

$$L = \frac{1}{2}(\hat{y} - y)^2$$

where \hat{y} is the predicted output and y is the true label. Then we adjust weights by taking partial derivatives of L w.r.t. w (chain rule) so that we nudge them in the direction that reduces the loss.

✓ 5. Illustrative Example with Code

We'll build a tiny MLP that learns the XOR function (an exclusive OR truth table). XOR's outputs are tricky for a single neuron, but with a small hidden layer, it works!

XOR Truth Table: | x1 | x2 | x1 XOR x2 | |---|-----| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 |

```
# Let's implement a tiny MLP for XOR in Python (using only NumPy):
import numpy as np
```

```
# XOR dataset
X = np.array([[0, 0],
              [0, 1],
```

```

    [1, 0],
    [1, 1]])
y = np.array([[0],
              [1],
              [1],
              [0]])

np.random.seed(0)

# Initialize weights
W1 = np.random.randn(2, 2) # 2 inputs -> 2 hidden
b1 = np.zeros((1, 2))
W2 = np.random.randn(2, 1) # 2 hidden -> 1 output
b2 = np.zeros((1, 1))

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

def sigmoid_derivative(x):
    return sigmoid(x) * (1 - sigmoid(x))

learning_rate = 0.1

for epoch in range(10000):
    # Forward pass
    z1 = np.dot(X, W1) + b1 # shape: (4, 2)
    a1 = sigmoid(z1) # shape: (4, 2)
    z2 = np.dot(a1, W2) + b2 # shape: (4, 1)
    a2 = sigmoid(z2) # shape: (4, 1)

    # Compute loss (Mean Squared Error)
    loss = np.mean((y - a2)**2)

    # Backpropagation
    dLoss_da2 = -(y - a2)
    dLoss_dz2 = dLoss_da2 * sigmoid_derivative(z2)

    dLoss_dW2 = np.dot(a1.T, dLoss_dz2)
    dLoss_db2 = np.sum(dLoss_dz2, axis=0, keepdims=True)

    dLoss_da1 = np.dot(dLoss_dz2, W2.T)
    dLoss_dz1 = dLoss_da1 * sigmoid_derivative(z1)

    dLoss_dW1 = np.dot(X.T, dLoss_dz1)
    dLoss_db1 = np.sum(dLoss_dz1, axis=0, keepdims=True)

    # Gradient Descent update
    W2 -= learning_rate * dLoss_dW2
    b2 -= learning_rate * dLoss_db2
    W1 -= learning_rate * dLoss_dW1
    b1 -= learning_rate * dLoss_db1

    # Print loss every 2000 epochs
    if epoch % 2000 == 0:
        print(f"Epoch {epoch}, Loss: {loss}")

print("\nFinal predictions after training:")
predictions = a2 > 0.5
print(predictions.astype(int))

↔ Epoch 0, Loss: 0.2696367200181884
Epoch 2000, Loss: 0.14613094454674397
Epoch 4000, Loss: 0.13288176632884824
Epoch 6000, Loss: 0.12955145571861026
Epoch 8000, Loss: 0.12814004233797888

Final predictions after training:
[[0]
 [0]
 [1]
 [1]]

```

6. Example Calculations

- **Weights (W):** Numbers used to multiply each input.
- **Bias (b):** A number added to the weighted input before activation.
- **Activation Function (σ):** A function applied to the sum of weighted inputs, e.g. sigmoid.
- **Forward Pass:** Computing the output for each layer.
- **Loss:** The measure of error (e.g., Mean Squared Error).

- **Backward Pass** (Backpropagation): Updating weights to minimize the loss.

Example of a single neuron calculation:

$$z = w_1x_1 + w_2x_2 + b$$
$$\text{output} = \sigma(z) = \frac{1}{1 + e^{-z}}$$

7. Step-by-Step Example from Scratch

1. **Define the Network Architecture:** Decide how many neurons in each layer (input, hidden, output).
2. **Initialize Weights & Biases:** Usually random small values.
3. **Define Activation Function:** Sigmoid, ReLU, etc.
4. **Forward Pass:** Multiply inputs by weights, add bias, apply activation.
5. **Calculate Loss:** Compare prediction with the true label.
6. **Backpropagation:** Compute gradient of the loss w.r.t. weights and biases.
7. **Update Weights:** Adjust parameters to reduce loss.
8. **Repeat** until convergence or max epochs.

8. What Illustrative Problem Does MLP Solve?

We used **XOR** as a classic example because a single perceptron (without hidden layers) *cannot* solve XOR. A small MLP with a hidden layer *can* solve it, showcasing why multiple layers are powerful.

9. Practical, Real-World Problem

MLPs can handle a variety of tasks, for example:

- Handwritten digit recognition (MNIST dataset)
- Spam detection in emails
- Credit score prediction
- Basic image classification

For instance, with digit recognition, each pixel is an input. The MLP learns weights (important pixels) and biases to identify the digit accurately.

10. How to Solve a Real-World Problem Using This Tech

1. Collect or gather a labeled dataset (e.g., images, text, or numeric data).
2. Preprocess and split data into training and testing sets.
3. Define your MLP architecture (number of layers, neurons, activation).
4. Train the MLP using forward pass + backprop.
5. Evaluate on the test set. If performance is low, tune hyperparameters or gather more data.
6. Deploy the trained model to make predictions in the real world.

11. What Other Questions Can You Ask?

1. **How many hidden layers do we need and why?**
2. **What activation function is best for my problem?**
3. **How do we deal with overfitting and underfitting?**
4. **What is the difference between MLP and other deep architectures (CNN, RNN)?**
5. **How does learning rate affect training?**

12. Questions & Answers with Code Examples

Q1: How many hidden layers do we need?

A1: There's no one-size-fits-all. Simple problems might need 1 or 2 hidden layers, but more complex tasks might need deeper networks. Experimentation is key.

Q2: Which activation function is best?

A2: Sigmoid is good for small networks, ReLU often works best for deeper networks because it helps avoid vanishing gradients.

Q3: How do we handle overfitting?

A3: Use regularization (e.g., L2, dropout), gather more data, or reduce network complexity.

Q4: MLP vs CNN vs RNN?

A4: MLP is fully connected, CNN is specialized for images (exploits spatial structure), RNN is for sequential data.

Q5: How does learning rate affect training?

A5: A high learning rate can overshoot minima; a low learning rate can be very slow or get stuck.

✓ 13. A Sample Exercise

Below is a simple code example using **Keras** (TensorFlow) to classify the Iris dataset. Some sections are marked with **TODO** for you to fill in.

```
# PLEASE COMPLETE THE TODO SECTIONS
import numpy as np
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import OneHotEncoder, StandardScaler
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

# Load data
iris = load_iris()
X = iris.data # shape (150, 4)
y = iris.target.reshape(-1, 1) # shape (150, 1)

# One-hot encode target
encoder = OneHotEncoder(sparse_output=False)
y = encoder.fit_transform(y) # shape (150, 3)

# Split data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Scale features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Check if GPU is available
device = 'GPU' if tf.config.list_physical_devices('GPU') else 'CPU'

print(f"Training on {device}")

with tf.device(device):
    model = Sequential()
    model.add(Dense(8, activation='relu', input_shape=(4,)))
    model.add(Dense(8, activation='relu'))
    model.add(Dense(3, activation='softmax'))

    model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

    model.fit(X_train, y_train, epochs=50, batch_size=16, verbose=0)

loss, accuracy = model.evaluate(X_test, y_test, verbose=0)
print(f"Test Accuracy: {accuracy*100:.2f}%")

## TODO: Create a Sequential model
# model = Sequential()

## TODO: Add a Dense hidden layer with, say, 8 neurons and activation='relu'
# model.add(Dense(8, activation='relu', input_shape=(4,)))


## TODO: Add another Dense hidden layer with, say, 8 neurons and activation='relu'
# model.add(Dense(8, activation='relu'))

## TODO: Add the output layer with 3 neurons (because 3 classes) and activation='softmax'
# model.add(Dense(3, activation='softmax'))

## Compile the model
## TODO: Choose an optimizer, e.g., 'adam', and a loss, e.g., 'categorical_crossentropy'
# model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
```

```
## Train the model
# model.fit(X_train, y_train, epochs=50, batch_size=16, verbose=0)
```

```
## Evaluate
# loss, accuracy = model.evaluate(X_test, y_test, verbose=0)
# print(f"Test Accuracy: {accuracy*100:.2f}%")
```

 Training on GPU
/usr/local/lib/python3.11/dist-packages/keras/src/layers/core/dense.py:87: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using `super().__init__(activity_regularizer=activity_regularizer, **kwargs)`
Test Accuracy: 93.33%


14. Glossary

- **Activation Function:** A function (Sigmoid, ReLU) that adds non-linearity to the neuron's output.
- **Backpropagation:** Algorithm to compute gradients for all weights in a network, used to train MLP.
- **Bias:** A constant term added before activation.
- **Epoch:** One complete pass through the entire training dataset.
- **Gradient Descent:** An optimization method that updates parameters in the direction opposite the gradient of the loss.
- **Hidden Layer:** The layers between input and output.
- **Loss Function:** Measures how far the network's predictions are from the true labels.
- **Neuron:** The basic unit in a network that computes weighted sums and applies an activation.
- **Weight:** A parameter that scales input data in a neuron.
- **MLP:** A type of neural network with multiple layers of perceptrons.

```
import os, sys, platform, datetime, uuid, socket
```

```
def signoff(name="Anonymous"):
    colab_check = "Yes" if 'google.colab' in sys.modules else "No"
    mac_addr = ':'.join(format((uuid.getnode() >> i) & 0xff, '02x') for i in reversed(range(0, 48, 8)))
    print("+++ Acknowledgement +++")
    print(f"Executed on: {datetime.datetime.now()}")
    print(f"In Google Colab: {colab_check}")
    print(f"System info: {platform.system()} {platform.release()}")
    print(f"Node name: {platform.node()}")
    print(f"MAC address: {mac_addr}")
    try:
        print(f"IP address: {socket.gethostbyname(socket.gethostname())}")
    except:
        print("IP address: Unknown")
    print(f"Signing off, {name}")
```

```
signoff("Ali Muhammad Asad")
```

 +++ Acknowledgement +++
Executed on: 2025-01-28 12:46:42.148449
In Google Colab: Yes
System info: Linux 6.1.85+
Node name: 6ecffb779a71
MAC address: 02:42:ac:1c:00:0c
IP address: 172.28.0.12
Signing off, Ali Muhammad Asad

Start coding or generate with AI.