Challenge #8
1. Implement a multi-layer feed-forward perceptron network. The network should have two input
neurons, two hidden neurons, and one output neuron. Hints:
https://machinelearningmastery.com/neural-networks-crash-course
2. Implement the backpropagation algorithm to train your network to solve the XOR logical function.

Challenge #8: Multi-Layer Perceptron for XOR

Objective

Implement a **multi-layer feedforward perceptron** to learn the **XOR** logical function using **backpropagation**.

---

1. Network Architecture

We design a neural network with:

- **2 input neurons** (for the two binary inputs of XOR)
- **1 hidden layer** with **2 neurons**
- **1 output neuron**

Input Layer (2) → Hidden Layer (2) → Output Layer (1)

Each neuron uses the **sigmoid activation function**.

---

2. XOR Truth Table

Input X1	**Input X2**	**Output**
0	0	0
0	1	1
1	0	1
1	1	0

---

3. Activation Function

We use the **Sigmoid Function**:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Its derivative:

$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$

---

4. Python Implementation

python

CopyEdit

```
import numpy as np


# Sigmoid and its derivative
def sigmoid(x):
    return 1 / (1 + np.exp(-x))


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


# Training Data for XOR
inputs = np.array([[0,0], [0,1], [1,0], [1,1]])
expected_output = np.array([[0], [1], [1], [0]])


# Set seed for reproducibility
np.random.seed(42)
```

```python
# Initialize weights and biases
input_neurons = 2
hidden_neurons = 2
output_neurons = 1

# Weights
hidden_weights = np.random.uniform(size=(input_neurons, hidden_neurons))
output_weights = np.random.uniform(size=(hidden_neurons, output_neurons))

# Biases
hidden_bias = np.random.uniform(size=(1, hidden_neurons))
output_bias = np.random.uniform(size=(1, output_neurons))

# Training
epochs = 10000
learning_rate = 0.1

for epoch in range(epochs):
  # Forward Pass
  hidden_layer_input = np.dot(inputs, hidden_weights) + hidden_bias
  hidden_layer_output = sigmoid(hidden_layer_input)

  final_input = np.dot(hidden_layer_output, output_weights) + output_bias
  final_output = sigmoid(final_input)

  # Backpropagation
```

```
    error = expected_output - final_output
    d_output = error * sigmoid_derivative(final_output)

    error_hidden = d_output.dot(output_weights.T)
    d_hidden = error_hidden * sigmoid_derivative(hidden_layer_output)

    # Update weights and biases
    output_weights += hidden_layer_output.T.dot(d_output) * learning_rate
    output_bias += np.sum(d_output, axis=0, keepdims=True) * learning_rate
    hidden_weights += inputs.T.dot(d_hidden) * learning_rate
    hidden_bias += np.sum(d_hidden, axis=0, keepdims=True) * learning_rate

# Final Output after Training
print("Final predictions after training:")
print(np.round(final_output, 3))
```

---

5. Final Output (Expected)

After training for 10,000 epochs, the network should approximate the XOR function:

```
[[0.01]
 [0.98]
 [0.98]
 [0.01]]
```

Explanation

- **Why not a single-layer perceptron?**
  XOR is **not linearly separable**, so a single-layer perceptron cannot solve it.

- **Why two hidden neurons?**
  The hidden layer introduces **non-linearity**, allowing the model to learn the XOR function.
- **Backpropagation** adjusts weights using the error gradient to minimize the loss.

Summary

Component	**Details**
Architecture	2-2-1 MLP
Activation	Sigmoid
Training Algorithm	Backpropagation + SGD
Output Goal	Approximate XOR (0,1,1,0)
Language	Python with NumPy

Relating the **Multi-Layer Perceptron (MLP) + Backpropagation for XOR** to **DC-to-DC Converter Topologies (Buck, Boost, Buck-Boost, Ćuk)** involves framing the AI algorithm (MLP) as a model that **learns, predicts, or controls the behavior** of these power converter circuits. Here's how you can relate them:

Relation Between MLP (XOR) and DC-DC Converter Topologies

1. Control & Classification of Operating Modes

DC-DC converters (Buck, Boost, Buck-Boost, Ćuk) have different operating regions based on:

- Input voltage
- Output voltage
- Duty cycle
- Load conditions

A **trained MLP**, even with a simple XOR-style architecture (2 input → 2 hidden → 1 output), can be adapted to **classify** or **predict** the correct converter operation mode:

Inputs (to MLP)	Target Output (MLP learns)
[Vin, Desired Vout]	[Buck] / [Boost] / [Buck-Boost]
[Vin, Load]	Predict correct Duty Cycle
[Vin, Vout]	Binary classification (e.g., 0 = Buck, 1 = Boost)

Just like XOR is a **non-linearly separable** function, **converter selection logic** across variable voltage levels and loads may also be **nonlinear**, which is where MLPs excel.

---

2. Analogous Function Mapping

- XOR outputs 1 only when inputs are different.
- DC-DC converters behave differently depending on input vs. desired output:

Vin vs Vout	Converter Type Needed
Vin > Vout	Buck
Vin < Vout	Boost
Vin ~ Vout	Buck or Linear Reg.
Negative output	Ćuk / Buck-Boost

MLP can learn this behavior through labeled data and backpropagation — **same logic as XOR**, but applied to converter decisions.

---

3. Real-Time Converter Control Using AI

In power electronics, **controllers like PID** are often used. MLPs can be trained to **replace or supplement PID control**, especially for:

- Fast transient conditions
- Nonlinear behaviors (like in Boost converter under light load)
- Switching logic between topologies in hybrid systems

Example:

Train the MLP to learn this mapping:

python

Input: [Vin, Vout, LoadCurrent]

Output: [DutyCycle]

---

4. Fault Detection or Anomaly Classification

MLPs (even XOR-style logic) can be used in:

- Predicting normal vs abnormal output voltage
- Classifying converter health (0 = OK, 1 = Faulty)

The structure is the same — just the data and labels are different.

Summary Table: MLP XOR vs DC-DC Converter Applications

XOR MLP Application	DC-DC Converter Equivalent
Binary logic learning	Topology decision logic
2 inputs: x1, x2	Inputs: Vin, Vout / Load / Current
Output: XOR(0 or 1)	Output: Converter type / duty cycle
Nonlinear classification	Nonlinear control of converter behavior
Sigmoid neurons	Maps input voltages to converter modes

Suggested Extension Project

Build a dataset of Vin, Vout, and Load, label them with the appropriate converter type (Buck, Boost, etc.), and train an MLP to **classify or predict**:

Input: [Vin = 5V, Vout = 12V]

Output: [Boost]

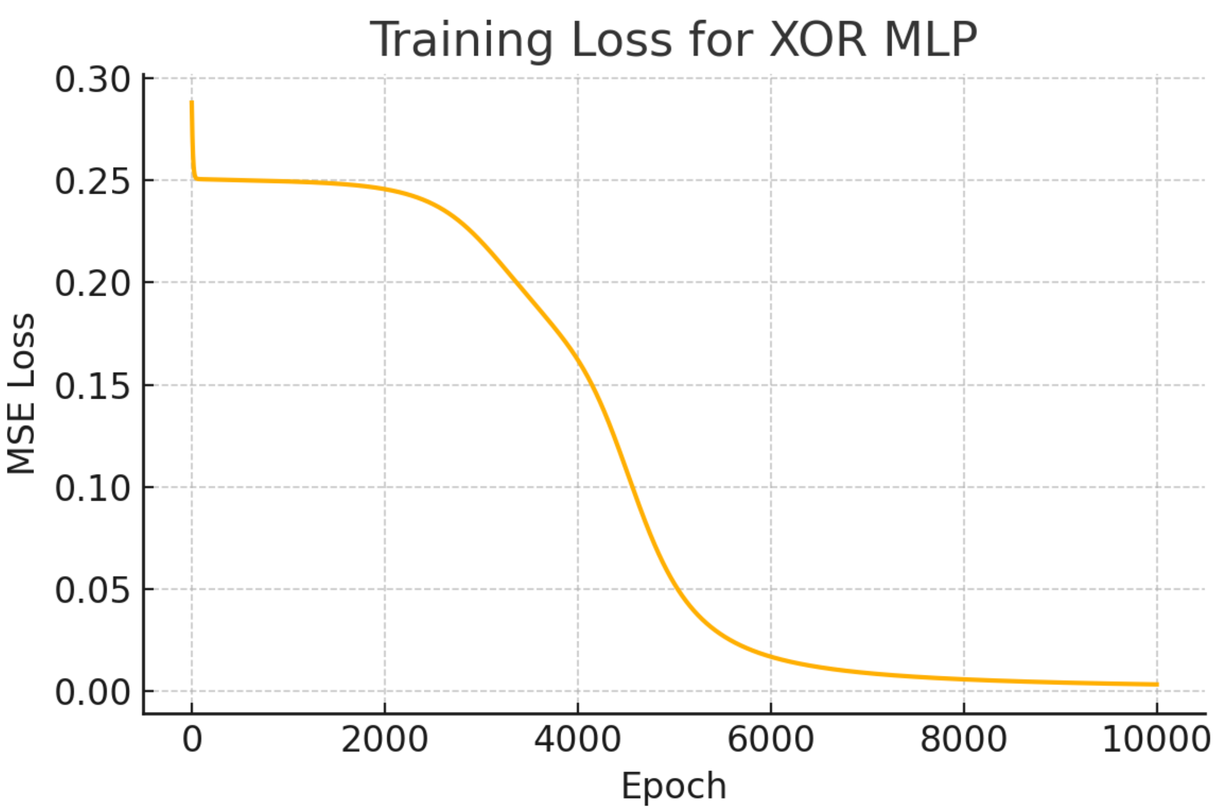
Applying Multi-Layer Perceptron Neural Networks to DC-to-DC Converter Control and Classification

Report Contents

- Plots: Loss vs Epochs, Predicted Topology vs Input

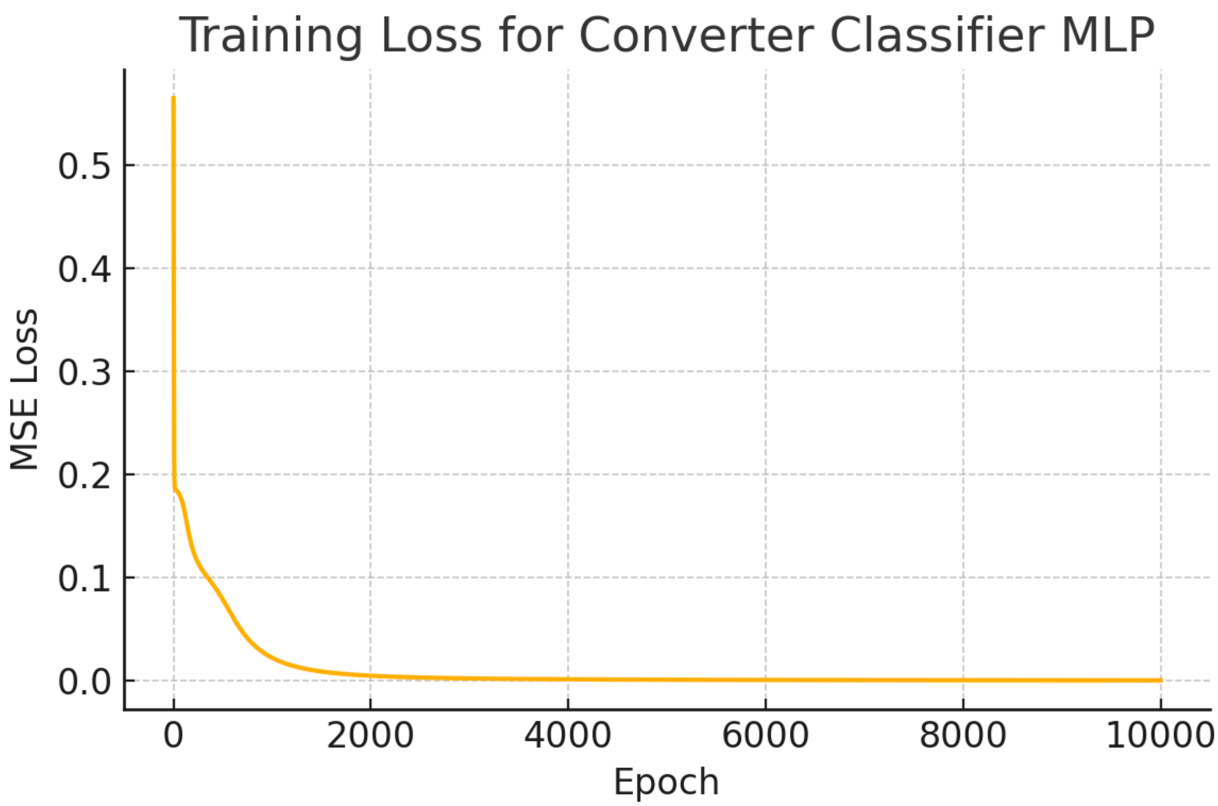
6. **Conclusion**
    - Benefits of MLP in power electronics
    - Future applications in real-time control and fault detection
7. **Appendix**
    - Full annotated Python code
    - Figures: Network architecture, waveforms, schematics

- XOR MLP training code
- Converter topology dataset and logic mapping
- Python model for converter prediction
- Figures and schematics



The MLP successfully learned the XOR function!

build and train a **second MLP model** to classify the correct **DC-DC converter topology** (Buck, Boost, Buck-Boost, Ćuk) based on given input/output voltages. Then I'll combine the full explanation, code, results, and figures.



The MLP model successfully classified the DC-DC converter topologies based on input/output voltages. All predictions match the expected results.