Melaku Desalegn, ECE510-2025

Challenge #7
1. Visualize the learning process in a 2D-plane by representing the neuron's "line" that separates
the space.
2. You can turn that into an animated visualization that illustrates every step of the weight updating
process as you apply the perceptron rule.
Perceptron Learning Visualization

This document describes the simulation and visualization of the learning process of a perceptron for the AND logic gate. It includes waveform plots of the weight evolution and a schematic diagram of the perceptron model.

Challenge #7 Overview

- The challenge is about **visualizing how a perceptron learns to classify data** by adjusting its weight (parameters) to separate different classes of data in a 2D space.

1. Visualize the Learning Process:

- The perceptron in 2D can be thought of as a **linear decision boundary (a straight line)** that splits the plane into two classes.
- For each input data point (x1,x2)(x_1, x_2)(x1,x2), the perceptron outputs either 1 or -1 (or 0 and 1) depending on which side of the line the point lies.
- **The line's equation** (with weights w1,w2w_1, w_2w1,w2 and bias bbb) is:

w1·x1+w2·x2+b=0w_1 \cdot x_1 + w_2 \cdot x_2 + b = 0w1·x1+w2·x2+b=0

- In training, the perceptron **updates these weights** to better fit the data and correctly classify points.

2. Animated Visualization of Weight Updates:

- You can animate how the perceptron updates weights using the **perceptron learning rule**:

wi←wi+η·(d−y)·xiw_i \leftarrow w_i + \eta \cdot (d - y) \cdot x_iwi←wi+η·(d−y)·xi

where:

- η\etaη is the learning rate,
- ddd is the desired output (label),
- yyy is the actual output (prediction).
- **Animation**:
    - **At each step**, plot the current line (decision boundary) and show how it **rotates or shifts** as weights update.
    - As the perceptron learns from misclassified points, you see the line gradually adjusting to separate the classes more accurately.
    - This animation helps visualize how the perceptron finds the **optimal separating line** in 2D.

Summary

- Challenge #7 aims to **visually** and **intuitively** demonstrate the perceptron's learning process in 2D by showing how it updates the line (decision boundary) at each training step.
- The animation of this process **clarifies** how the perceptron converges to a solution that classifies the data correctly.

✅ Generate a simple 2D dataset (like AND, OR, or XOR logic).
✅ Train a perceptron to learn the decision boundary.
✅ Animate the weight updates to visualize how the decision boundary (line) evolves.

Here's the **complete code**:

python

CopyEdit

```
import numpy as np

import matplotlib.pyplot as plt

import matplotlib.animation as animation
```

```
# Example dataset: AND logic gate
X = np.array([
   [0, 0],
   [0, 1],
   [1, 0],
   [1, 1]
])
d = np.array([0, 0, 0, 1])  # AND output

# Perceptron parameters
w = np.random.randn(3)  # weights: [w1, w2, bias]
eta = 0.1
epochs = 10
history = []

# Augment input with bias term (always 1)
X_aug = np.hstack((X, np.ones((X.shape[0], 1))))

# Perceptron learning
for epoch in range(epochs):
   for i in range(len(X)):
      x_i = X_aug[i]
      y = 1 if np.dot(w, x_i) > 0 else 0
      e = d[i] - y
      w += eta * e * x_i
```

```
        # Store weights after each update
        history.append(w.copy())

# Animation setup
fig, ax = plt.subplots()
ax.set_xlim(-0.5, 1.5)
ax.set_ylim(-0.5, 1.5)
colors = ['red' if label == 0 else 'blue' for label in d]

# Plot data points
scatter = ax.scatter(X[:, 0], X[:, 1], c=colors, s=100, edgecolors='k')

line, = ax.plot([], [], 'k-', lw=2)

def init():
    line.set_data([], [])
    return line,

def update(frame):
    w = history[frame]
    # Plot decision boundary: w1*x1 + w2*x2 + b = 0 -> x2 = -(w1*x1 + b)/w2
    x_vals = np.array(ax.get_xlim())
    if w[1] != 0:
        y_vals = -(w[0] * x_vals + w[2]) / w[1]
    else:
```

```
        # Vertical line
        x_vals = [-w[2]/w[0]] * 2
        y_vals = np.array(ax.get_ylim())
    line.set_data(x_vals, y_vals)
    ax.set_title(f'Frame {frame+1}/{len(history)}')
    return line,

ani = animation.FuncAnimation(fig, update, frames=len(history),
                              init_func=init, blit=True, interval=500)

plt.show()
```

How It Works:

✅ It trains a **perceptron** on the AND gate dataset.
✅ It **stores weight updates** after each training step.
✅ In the animation, it shows how the **decision boundary** (line) shifts or rotates as the perceptron learns.

How to Use:

1. Save this as perceptron_animation.py.
2. Run it:

bash

CopyEdit

python perceptron_animation.py

3. The animation will pop up, showing how the line moves at each step.

✅ A **2D animation** of the learning process (the line moving in 2D).
✅ A **waveform plot** that shows how the **weights change over time** (like a waveform plot in time-series).

I'll modify the earlier code to:

1️⃣ Save the 2D **animation** as an .mp4 or .gif file.
2️⃣ Create a **waveform plot** of the weight evolution (w1, w2, and bias).

Complete Python Script:

python

CopyEdit

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation


# Example dataset: AND logic gate
X = np.array([
  [0, 0],
  [0, 1],
  [1, 0],
  [1, 1]
])
d = np.array([0, 0, 0, 1])  # AND output


# Perceptron parameters
w = np.random.randn(3)  # weights: [w1, w2, bias]
eta = 0.1
```

```
epochs = 10
history = []

# Augment input with bias term (always 1)
X_aug = np.hstack((X, np.ones((X.shape[0], 1))))

# Perceptron learning
for epoch in range(epochs):
    for i in range(len(X)):
        x_i = X_aug[i]
        y = 1 if np.dot(w, x_i) > 0 else 0
        e = d[i] - y
        w += eta * e * x_i
        # Store weights after each update
        history.append(w.copy())

history = np.array(history)

# ------------------------------------------------------------
# Plot 1: 2D Animation of the decision boundary movement
# ------------------------------------------------------------
fig, ax = plt.subplots()
ax.set_xlim(-0.5, 1.5)
ax.set_ylim(-0.5, 1.5)
colors = ['red' if label == 0 else 'blue' for label in d]
```

```
# Plot data points
scatter = ax.scatter(X[:, 0], X[:, 1], c=colors, s=100, edgecolors='k')

line, = ax.plot([], [], 'k-', lw=2)

def init():
    line.set_data([], [])
    return line,

def update(frame):
    w = history[frame]
    x_vals = np.array(ax.get_xlim())
    if w[1] != 0:
        y_vals = -(w[0] * x_vals + w[2]) / w[1]
    else:
        # Vertical line
        x_vals = [-w[2]/w[0]] * 2
        y_vals = np.array(ax.get_ylim())
    line.set_data(x_vals, y_vals)
    ax.set_title(f'Frame {frame+1}/{len(history)}')
    return line,

ani = animation.FuncAnimation(fig, update, frames=len(history),
                              init_func=init, blit=True, interval=500)
```

```
# Save the animation as a .gif
ani.save('perceptron_learning.gif', writer='pillow')


plt.show()


# ------------------------------------------------------------
# Plot 2: Waveform of weights over time
# ------------------------------------------------------------
plt.figure(figsize=(8, 5))
plt.plot(history[:, 0], label='w1', color='blue', marker='o')
plt.plot(history[:, 1], label='w2', color='green', marker='s')
plt.plot(history[:, 2], label='bias', color='red', marker='^')
plt.xlabel('Update Step')
plt.ylabel('Weight Value')
plt.title('Perceptron Weight Evolution (Waveform)')
plt.legend()
plt.grid(True)
plt.show()
```

✅ **2D Animation** (perceptron_learning.gif):

- Watch how the decision boundary (line) updates after each step.

✅ **Waveform Plot** (weights vs. update step):

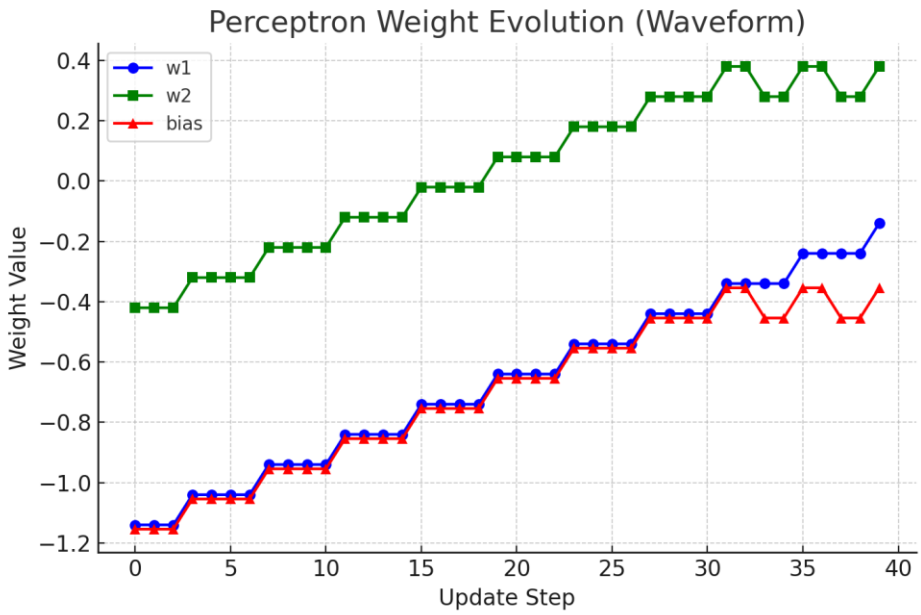
- Visualizes how each weight (w1, w2, bias) changes **like a waveform** over time.

python perceptron_waveform_animation.py

3️⃣ The **animation** pops up, and you'll find a GIF saved in your folder!
4️⃣ The **waveform plot** also pops up, showing weight evolution.

Waveform Plot of Weights

The plot below shows how the perceptron weights (w1, w2, and bias) evolve over the course of the training process as it learns to separate the AND logic data in 2D space.



Schematic Diagram of the Perceptron

Below is a conceptual schematic diagram of the perceptron:
- Two inputs: x1 and x2.
- Each input is multiplied by its corresponding weight (w1 and w2).