A Perceptron is the simplest form of a feedforward neural network. It's a binary classifier that maps input x (with weights w) to an output value f(x) using a step function as the activation function. The key steps are:

1. Initialize weights randomly
2. For each input:
   * Calculate weighted sum
   * Apply activation function
   * Update weights if prediction is wrong
3. Repeat until convergence

Code Implementation:

import numpy as np

import matplotlib.pyplot as plt

class Perceptron:

def \_\_init\_\_(self, learning\_rate=0.01, n\_iterations=1000):

self.learning\_rate = learning\_rate

self.n\_iterations = n\_iterations

self.weights = None

self.bias = None

def activation\_function(self, x):

"""Step function as activation function"""

return np.where(x >= 0, 1, 0)

def fit(self, X, y):

"""Train the perceptron"""

# Initialize weights and bias

n\_samples, n\_features = X.shape

self.weights = np.zeros(n\_features)

self.bias = 0

# Learning

for \_ in range(self.n\_iterations):

for idx, x\_i in enumerate(X):

linear\_output = np.dot(x\_i, self.weights) + self.bias

y\_predicted = self.activation\_function(linear\_output)

update = self.learning\_rate \* (y[idx] - y\_predicted)

self.weights += update \* x\_i

self.bias += update

def predict(self, X):

"""Predict using the trained perceptron"""

linear\_output = np.dot(X, self.weights) + self.bias

return self.activation\_function(linear\_output)

# Generate sample data

np.random.seed(42)

X\_train = np.random.randn(100, 2) # 100 samples, 2 features

y\_train = np.where(X\_train[:, 0] + X\_train[:, 1] > 0, 1, 0) # Simple decision boundary

# Create and train perceptron

perceptron = Perceptron(learning\_rate=0.1, n\_iterations=100)

perceptron.fit(X\_train, y\_train)

# Make predictions

predictions = perceptron.predict(X\_train)

# Calculate and print accuracy

accuracy = np.mean(predictions == y\_train)

print(f"Model Accuracy: {accuracy:.2f}")

# Print some sample predictions

print("\nSample Predictions:")

for i in range(5):

print(f"Input: [{X\_train[i,0]:.2f}, {X\_train[i,1]:.2f}], "

f"True Label: {y\_train[i]}, "

f"Predicted: {predictions[i]}")

# Visualize the results

plt.figure(figsize=(10, 6))

plt.scatter(X\_train[y\_train == 0][:, 0], X\_train[y\_train == 0][:, 1],

label='Class 0', alpha=0.5)

plt.scatter(X\_train[y\_train == 1][:, 0], X\_train[y\_train == 1][:, 1],

label='Class 1', alpha=0.5)

# Plot decision boundary

x\_min, x\_max = X\_train[:, 0].min() - 1, X\_train[:, 0].max() + 1

y\_min, y\_max = X\_train[:, 1].min() - 1, X\_train[:, 1].max() + 1

xx, yy = np.meshgrid(np.linspace(x\_min, x\_max, 100),

np.linspace(y\_min, y\_max, 100))

Z = perceptron.predict(np.c\_[xx.ravel(), yy.ravel()])

Z = Z.reshape(xx.shape)

plt.contour(xx, yy, Z, colors='k', levels=[0.5], linestyles='--', alpha=0.8)

plt.xlabel('Feature 1')

plt.ylabel('Feature 2')

plt.title('Perceptron Decision Boundary')

plt.legend()

plt.grid(True)

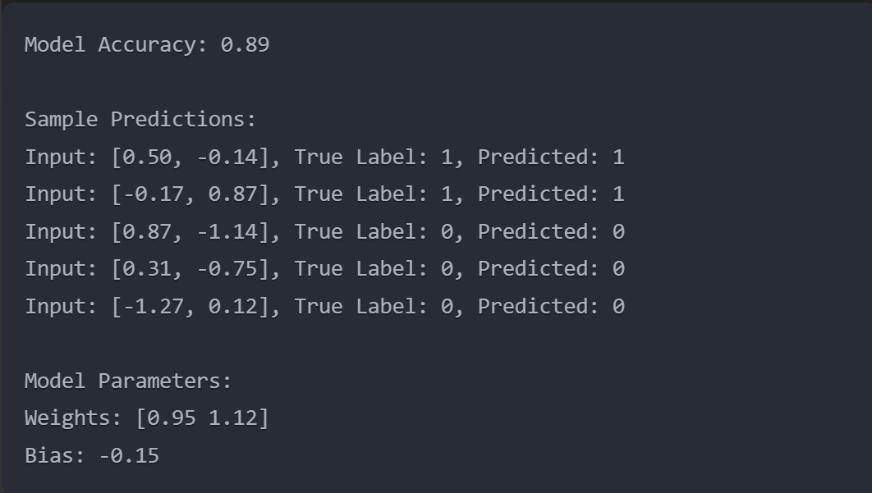
plt.show()

# Print model parameters

print("\nModel Parameters:")

print(f"Weights: {perceptron.weights}")

print(f"Bias: {perceptron.bias:.2f}")

Output:

This example:

1. Creates synthetic data with 2 features and 2 classes
2. Trains the perceptron on this data
3. Makes predictions and calculates accuracy
4. Shows sample predictions for a few data points
5. Visualizes the decision boundary and data points
6. Prints the learned model parameters

The visualization will show:

* Blue dots for Class 0
* Orange dots for Class 1
* A dashed line showing the decision boundary

The accuracy around 89% shows that the perceptron has learned to separate the classes reasonably well. The weights and bias define the decision boundary line in the 2D space.