**Deep Learning:**

import tensorflow as tf

from tensorflow.keras.datasets import imdb

from tensorflow.keras.preprocessing.sequence import pad\_sequences

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Embedding, LSTM, Dense

# Load top 10,000 words from IMDB dataset

(x\_train, y\_train), (x\_test, y\_test) = imdb.load\_data(num\_words=10000)

# Pad sequences to the same length (200 words)

x\_train = pad\_sequences(x\_train, maxlen=200)

x\_test = pad\_sequences(x\_test, maxlen=200)

model = Sequential([

Embedding(input\_dim=10000, output\_dim=64, input\_length=200),

LSTM(64),

Dense(1, activation='sigmoid') # Output: 0 (neg) or 1 (pos)

])

model.compile(loss='binary\_crossentropy', optimizer='adam', metrics=['accuracy'])

model.fit(x\_train, y\_train, epochs=3, batch\_size=64, validation\_split=0.2)

loss, acc = model.evaluate(x\_test, y\_test)

print(f"Test Accuracy: {acc:.2f}")

CNN:

import numpy as np

from tensorflow import keras

from tensorflow.keras import layers

# Parameters

num\_classes = 10

input\_shape = (28, 28, 1)

# Load and preprocess data

(x\_train, y\_train), (x\_test, y\_test) = keras.datasets.mnist.load\_data()

x\_train = x\_train.astype("float32") / 255

x\_test = x\_test.astype("float32") / 255

# Reshape to (28, 28, 1)

x\_train = np.expand\_dims(x\_train, -1)

x\_test = np.expand\_dims(x\_test, -1)

# One-hot encode labels

y\_train = keras.utils.to\_categorical(y\_train, num\_classes)

y\_test = keras.utils.to\_categorical(y\_test, num\_classes)

# Build CNN model

model = keras.Sequential([

layers.Conv2D(32, kernel\_size=3, activation="relu", input\_shape=input\_shape),

layers.MaxPooling2D(pool\_size=2),

layers.Conv2D(64, kernel\_size=3, activation="relu"),

layers.MaxPooling2D(pool\_size=2),

layers.Flatten(),

layers.Dropout(0.5),

layers.Dense(num\_classes, activation="softmax")

])

model.compile(optimizer="adam", loss="categorical\_crossentropy", metrics=["accuracy"])

model.summary()

# Train

model.fit(x\_train, y\_train, batch\_size=128, epochs=10, validation\_split=0.1)

# Evaluate

loss, acc = model.evaluate(x\_test, y\_test)

print(f"Test loss: {loss:.4f}, Test accuracy: {acc:.4f}")

**Q Learning:**

import numpy as np

# Q-learning settings

alpha = 0.1 # Learning rate

gamma = 0.9 # Discount factor

epsilon = 0.1 # Exploration rate

episodes = 1000 # Training episodes

# Environment: 5x5 grid, goal at (4, 4)

grid\_size = 5

num\_actions = 4 # up, down, left, right

q\_table = np.zeros((grid\_size \* grid\_size, num\_actions))

def state\_to\_index(state):

return state[0] \* grid\_size + state[1]

def choose\_action(state):

if np.random.rand() < epsilon:

return np.random.randint(num\_actions)

return np.argmax(q\_table[state\_to\_index(state)])

def take\_action(state, action):

row, col = state

if action == 0 and row > 0: row -= 1 # Up

elif action == 1 and row < 4: row += 1 # Down

elif action == 2 and col > 0: col -= 1 # Left

elif action == 3 and col < 4: col += 1 # Right

next\_state = (row, col)

reward = 1 if next\_state == (4, 4) else -0.1

done = next\_state == (4, 4)

return next\_state, reward, done

# Q-learning main loop

for ep in range(episodes):

state = (0, 0)

while True:

action = choose\_action(state)

next\_state, reward, done = take\_action(state, action)

i, ni = state\_to\_index(state), state\_to\_index(next\_state)

q\_table[i, action] += alpha \* (reward + gamma \* np.max(q\_table[ni]) - q\_table[i, action])

state = next\_state

if done:

break

if (ep + 1) % 100 == 0:

print(f"Episode {ep+1} complete")

# Test learned policy

def test\_policy():

state = (0, 0)

path = [state]

while state != (4, 4):

action = np.argmax(q\_table[state\_to\_index(state)])

state, \_, \_ = take\_action(state, action)

path.append(state)

return path

print("Learned path to goal:", test\_policy())