Exercise:13

```
import tensorflow as tf

import numpy as np

# Generate random data for demonstration

num_users = 1000

num_items = 500

num_samples = 10000

user_ids_train = np.random.randint(0, num_users, num_samples)

item_ids_train = np.random.randint(0, num_items, num_samples)

ratings_train = np.random.randint(1, 6, num_samples)  # Assume ratings are integers between
1 and 5

user_ids_val = np.random.randint(0, num_users, num_samples)

item_ids_val = np.random.randint(0, num_items, num_samples)

ratings_val = np.random.randint(1, 6, num_samples)

user_ids_test = np.random.randint(0, num_users, num_samples)

item_ids_test = np.random.randint(0, num_items, num_samples)

ratings_test = np.random.randint(1, 6, num_samples)

# Define the model architecture

class CollaborativeFilteringModel(tf.keras.Model):

    def __init__(self, num_users, num_items, embedding_size):

        super(CollaborativeFilteringModel, self).__init__()

        self.user_embedding = tf.keras.layers.Embedding(num_users, embedding_size)

        self.item_embedding = tf.keras.layers.Embedding(num_items, embedding_size)

        self.dot = tf.keras.layers.Dot(axes=1)

    def call(self, inputs):

        user_id, item_id = inputs

        user_embedding = self.user_embedding(user_id)

        item_embedding = self.item_embedding(item_id)

        return self.dot([user_embedding, item_embedding])
```

```
# Example usage
embedding_size = 50
model = CollaborativeFilteringModel(num_users, num_items, embedding_size)
model.compile(optimizer='adam', loss='mean_squared_error')
# Train the model
history = model.fit([user_ids_train, item_ids_train], ratings_train,
                    validation_data=([user_ids_val, item_ids_val], ratings_val),
                    epochs=10, batch_size=64)
# Evaluate the model
loss = model.evaluate([user_ids_test, item_ids_test], ratings_test)
print("Test Loss:", loss)
```

Exercise:14

```
import numpy as np
import tensorflow as tf
from tensorflow.keras import layers
import cv2
import matplotlib.pyplot as plt
# preapre handwritten digits
(X_num, y_num), _ = tf.keras.datasets.mnist.load_data()
X_num = np.expand_dims(X_num, axis=-1).astype(np.float32) / 255.0
grid_size = 16  # image_size / mask_size
def make_numbers(X, y):
    for _ in range(3):
        # pickup random index
        idx = np.random.randint(len(X_num))
        # make digit colorful
        number = X_num[idx] @ (np.random.rand(1, 3) + 0.1)
        number[number > 0.1] = np.clip(number[number > 0.1], 0.5, 0.8)
        # class of digit
        kls = y_num[idx]
        # random position for digit
        px, py = np.random.randint(0, 100), np.random.randint(0, 100)
        # digit belong which mask position
        mx, my = (px+14) // grid_size, (py+14) // grid_size
        channels = y[my][mx]
        # prevent duplicated problem
        if channels[0] > 0:
            continue
        channels[0] = 1.0
        channels[1] = px - (mx * grid_size)  # x1
```

```
        channels[2] = py - (my * grid_size)  # y1
        channels[3] = 28.0      # x2, in this demo image only 28 px as width
        channels[4] = 28.0      # y2, in this demo image only 28 px as height
        channels[5 + kls] = 1.0
        # put digit in X
        X[py:py+28, px:px+28] += number
def make_data(size=64):
    X = np.zeros((size, 128, 128, 3), dtype=np.float32)
    y = np.zeros((size, 8, 8, 15), dtype=np.float32)
    for i in range(size):
        make_numbers(X[i], y[i])
    X = np.clip(X, 0.0, 1.0)
    return X, y
def get_color_by_probability(p):
    if p < 0.3:
        return (1., 0., 0.)
    if p < 0.7:
        return (1., 1., 0.)
    return (0., 1., 0.)
def show_predict(X, y, threshold=0.1):
    X = X.copy()
    for mx in range(8):
        for my in range(8):
            channels = y[my][mx]
            prob, x1, y1, x2, y2 = channels[:5]
            # if prob < threshold we won't show any thing
            if prob < threshold:
                continue
            color = get_color_by_probability(prob)
            # bounding box
```

```
        px, py = (mx * grid_size) + x1, (my * grid_size) + y1
        cv2.rectangle(X, (int(px), int(py)), (int(px + x2), int(py + y2)), color, 1)
        # label
        cv2.rectangle(X, (int(px), int(py - 10)), (int(px + 12), int(py)), color, -1)
        kls = np.argmax(channels[5:])
        cv2.putText(X, f'{kls}', (int(px + 2), int(py-2)), cv2.FONT_HERSHEY_PLAIN, 0.7, (0.0, 0.0, 0.0))
  plt.imshow(X)
# test
X, y = make_data(size=1)
show_predict(X[0], y[0])
```

Exercise:15

```
import numpy as np
# Define the Q-learning parameters
num_states = 10  # Number of states (simplified for the example)
num_actions = 10  # Number of actions (simplified for the example)
Q = np.zeros((num_states, num_actions))  # Q-table initialization
alpha = 0.1  # Learning rate
gamma = 0.9  # Discount factor
epsilon = 0.1  # Exploration rate
# Simple environment simulation (simplified for the example)
def simulate_environment(state, action):
    reward = 0  # Placeholder reward (simplified for the example)
    next_state = (state + action) % num_states  # Transition to the next state
    return next_state, reward
# Q-learning algorithm
def train_q_learning(num_episodes):
    for episode in range(num_episodes):
        state = np.random.randint(0, num_states)  # Random initial state
        for _ in range(num_states):  # Max number of steps (simplified for the
example)
            if np.random.uniform(0, 1) < epsilon:
                action = np.random.randint(0, num_actions)  # Exploration
            else:
                action = np.argmax(Q[state, :])  # Exploitation
            next_state, reward = simulate_environment(state, action)
            Q[state, action] = (1 - alpha) * Q[state, action] + alpha * (reward +
gamma * np.max(Q[next_state, :]))
```

```
        state = next_state
# Generate response based on learned Q-values
def generate_response(state):
    action = np.argmax(Q[state, :])  # Choose action based on learned Q-values
    return action  # This is a simplified response generation based on the action
# Interactive dialogue interface
def interactive_dialogue():
    print("Welcome to the dialogue system!")
    print("Enter your dialogue context (an integer between 0 and 9):")
    while True:
        try:
            context = int(input())
            if 0 <= context < num_states:
                response_action = generate_response(context)
                print("Generated response action:", response_action)
            else:
                print("Context should be an integer between 0 and 9.")
        except ValueError:
            print("Invalid input. Please enter an integer.")

# Train the Q-learning model
num_episodes = 1000  # Number of training episodes
train_q_learning(num_episodes)
# Start interactive dialogue
interactive_dialogue()
```