1. Neural Network Implementation

program

```
import numpy as np
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import OneHotEncoder, StandardScaler
from sklearn.metrics import accuracy_score
iris = load_iris()
X = iris.data
y = iris.target.reshape(-1, 1)
encoder = OneHotEncoder(sparse=False)
y = encoder.fit_transform(y)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)
```

Department of Artificial Intelligence and Data Science

```
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
input_size = X_train.shape[1]
hidden_size = 10
output_size = y_train.shape[1]
learning_rate = 0.01
epochs = 1000
W1 = np.random.randn(input_size, hidden_size)
b1 = np.zeros((1, hidden_size))
W2 = np.random.randn(hidden_size, output_size)
b2 = np.zeros((1, output_size))
def sigmoid(x):
 return 1 / (1 + np.exp(-x))
def sigmoid_derivative(x):
 return x * (1 - x)
for epoch in range(epochs):
 z1 = np.dot(X_train, W1) + b1
 a1 = sigmoid(z1)
 z2 = np.dot(a1, W2) + b2
 a2 = sigmoid(z2)
 loss = np.mean((a2 - y_train) ** 2)
 d_loss_a2 = 2 * (a2 - y_train) / y_train.shape[0]
 d_a2_z2 = sigmoid_derivative(a2)
 d_z2_W2 = a1
 d_z2_a1 = W2
 d_z2 = d_loss_a2 * d_a2_z2
 d_W2 = np.dot(d_z2_W2.T, d_z2)
 d_b2 = np.sum(d_z2, axis=0, keepdims=True)
 d_a1_z1 = sigmoid_derivative(a1)
 d_z1_W1 = X_train
 d_z1 = np.dot(d_z2, d_z2_a1.T) * d_a1_z1
 d_W1 = np.dot(d_z1_W1.T, d_z1)
 d_b1 = np.sum(d_z1, axis=0, keepdims=True)
 W2 -= learning_rate * d_W2
```

```
 b2 -= learning_rate * d_b2
 W1 -= learning_rate * d_W1
 b1 -= learning_rate * d_b1
 if epoch % 100 == 0:
 print(f'Epoch {epoch}, Loss: {loss}')
z1 = np.dot(X_test, W1) + b1
a1 = sigmoid(z1)
z2 = np.dot(a1, W2) + b2
a2 = sigmoid(z2)
predictions = np.argmax(a2, axis=1)
y_test_labels = np.argmax(y_test, axis=1)
accuracy = accuracy_score(y_test_labels, predictions)
print(f'Accuracy: {accuracy * 100:.2f}%')
```

2. TensorFlow Basics

program

```
import tensorflow as tf
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.utils import to_categorical
from sklearn.metrics import accuracy_score
(X_train, y_train), (X_test, y_test) = mnist.load_data()
X_train = X_train / 255.0
X_test = X_test / 255.0
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)
model = Sequential([
 Flatten(input_shape=(28, 28)),
 Dense(128, activation='relu'),
```



```
 Dense(64, activation='relu'),
 Dense(10, activation='softmax')
])
model.compile(optimizer='adam',
 loss='categorical_crossentropy',
 metrics=['accuracy'])
model.fit(X_train, y_train, epochs=10, batch_size=32,
validation_split=0.2)
test_loss, test_accuracy = model.evaluate(X_test, y_test)
print(f'Test accuracy: {test_accuracy * 100:.2f}%')
y_pred = model.predict(X_test)
y_pred_classes = tf.argmax(y_pred, axis=1)
y_true = tf.argmax(y_test, axis=1)
accuracy = accuracy_score(y_true, y_pred_classes)
print(f'Accuracy calculated using sklearn: {accuracy * 100:.2f}%')
```

Output

3. Building a CNN

program

```
import tensorflow as tf
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.callbacks import EarlyStopping
(X_train, y_train), (X_test, y_test) = cifar10.load_data()
X_train = X_train / 255.0
X_test = X_test / 255.0
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)
def build_and_compile_model(filter_size, stride, padding):
 model = Sequential([
 Conv2D(32, kernel_size=filter_size, strides=stride, padding=padding,
activation='relu', input_shape=(32, 32, 3)),
```

Department of Artificial Intelligence and Data Science

```
 MaxPooling2D(pool_size=(2, 2)),
 Conv2D(64, kernel_size=filter_size, strides=stride, padding=padding,
activation='relu'),
 MaxPooling2D(pool_size=(2, 2)),
 Conv2D(128, kernel_size=filter_size, strides=stride, padding=padding,
activation='relu'),
 Flatten(),
 Dense(128, activation='relu'),
 Dense(10, activation='softmax')
 ])

 model.compile(optimizer='adam',
 loss='categorical_crossentropy',
 metrics=['accuracy'])
 return model
filter_sizes = [(3, 3), (5, 5)]
strides = [(1, 1), (2, 2)]
paddings = ['valid', 'same']
for filter_size in filter_sizes:
 for stride in strides:
 for padding in paddings:
 print(f"\nExperimenting with filter_size={filter_size}, stride={stride},
padding={padding}")

 model = build_and_compile_model(filter_size, stride, padding)

 early_stopping = EarlyStopping(monitor='val_loss', patience=3)
 history = model.fit(X_train, y_train, epochs=20, batch_size=64,
validation_split=0.2, callbacks=[early_stopping], verbose=1)

 test_loss, test_accuracy = model.evaluate(X_test, y_test)
 print(f"Test accuracy: {test_accuracy * 100:.2f}%")
 model.summary()
```

4. Improving CNN Performance

program

```
import tensorflow as tf
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense,
Dropout, BatchNormalization
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.callbacks import EarlyStopping
(X_train, y_train), (X_test, y_test) = cifar10.load_data()
X_train = X_train / 255.0
X_test = X_test / 255.0
y_train = to_categorical(y_train, 10)
```



```
y_test = to_categorical(y_test, 10)
datagen = ImageDataGenerator(
 rotation_range=20,
 width_shift_range=0.2,
 height_shift_range=0.2,
 horizontal_flip=True,
 fill_mode='nearest'
)
datagen.fit(X_train)
def build_and_compile_model():
 model = Sequential([
 Conv2D(32, (3, 3), padding='same', activation='relu', input_shape=(32, 32,
3)),
 BatchNormalization(),
 MaxPooling2D((2, 2)),
 Dropout(0.25),
 Conv2D(64, (3, 3), padding='same', activation='relu'),
 BatchNormalization(),
 MaxPooling2D((2, 2)),
 Dropout(0.25),
 Conv2D(128, (3, 3), padding='same', activation='relu'),
 BatchNormalization(),
 MaxPooling2D((2, 2)),
 Dropout(0.25),
 Flatten(),
 Dense(512, activation='relu'),
 Dropout(0.5),
 Dense(10, activation='softmax')
```



```
 ])
 model.compile(optimizer='adam',
 loss='categorical_crossentropy',
 metrics=['accuracy'])
 return model
model = build_and_compile_model()
```

```
early_stopping = EarlyStopping(monitor='val_loss', patience=5,
restore_best_weights=True)
history = model.fit(datagen.flow(X_train, y_train, batch_size=64),
 epochs=50,
 validation_data=(X_test, y_test),
 callbacks=[early_stopping],
 verbose=1)
test_loss, test_accuracy = model.evaluate(X_test, y_test)
print(f'Test accuracy: {test_accuracy * 100:.2f}%')
import matplotlib.pyplot as plt
plt.figure(figsize=(12, 4))
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Model accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(['Train', 'Validation'])
plt.subplot(1, 2, 2)
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model loss')
```
Department of Artificial Intelligence and Data Science
```
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend(['Train', 'Validation'])
plt.show()
```

5. RNN for sequence prediction using TensorFlow

program

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import SimpleRNN, Dense
from sklearn.preprocessing import MinMaxScaler
import matplotlib.pyplot as plt
def generate_sine_wave(seq_length, num_samples):
 x = np.linspace(0, 2 * np.pi, seq_length)
```
Department of Artificial Intelligence and Data Science
```
 data = np.sin(x)
 sequences = []
 for i in range(num_samples):
 start = np.random.randint(0, len(data) - seq_length)
 sequences.append(data[start:start + seq_length])
 return np.array(sequences)
seq_length = 50
num_samples = 1000
num_epochs = 10
batch_size = 32
data = generate_sine_wave(seq_length, num_samples)
```

```
X = data[:, :-1]
y = data[:, -1]
scaler = MinMaxScaler()
X = scaler.fit_transform(X)
y = scaler.transform(y.reshape(-1, 1)).flatten()
X = X.reshape((num_samples, seq_length - 1, 1))
model = Sequential([
 SimpleRNN(50, activation='relu', input_shape=(X.shape[1], 1)),
 Dense(1)
])
model.compile(optimizer='adam', loss='mse')
model.fit(X, y, epochs=num_epochs, batch_size=batch_size)
predictions = model.predict(X)
y = scaler.inverse_transform(y.reshape(-1, 1)).flatten()
predictions = scaler.inverse_transform(predictions).flatten()
plt.figure(figsize=(12, 6))
plt.plot(range(len(y)), y, label='True Values')
```

Department of Artificial Intelligence and Data Science

```
plt.plot(range(len(predictions)), predictions, label='Predicted Values')
plt.legend()
plt.show()
```

Output

6. LSTM network for text generation

program

```
import numpy as np
import tensorflow as tf
```

Department of Artificial Intelligence and Data Science

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense, Dropout
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.preprocessing.sequence import pad_sequences
# Load and preprocess the text data
def load_text(file_path):
 with open(file_path, 'r') as file:
 text = file.read().lower()
 return text
def prepare_data(text, seq_length):
 tokenizer = Tokenizer(char_level=True)
 tokenizer.fit_on_texts([text])
 total_chars = len(tokenizer.word_index) + 1
 sequences = []
 next_chars = []
 for i in range(0, len(text) - seq_length, 1):
 seq = text[i:i + seq_length]
 label = text[i + seq_length]
 sequences.append([tokenizer.char_index[c] for c in seq])
 next_chars.append(tokenizer.char_index[label])
 X = np.array(sequences)
```

```
 y = np.array(next_chars)
 X = pad_sequences(X, maxlen=seq_length)
 y = to_categorical(y, num_classes=total_chars)
 return X, y, tokenizer, total_chars
```



```
def build_model(seq_length, total_chars):
 model = Sequential([
 LSTM(128, input_shape=(seq_length, total_chars),
return_sequences=True),
 Dropout(0.2),
 LSTM(128),
 Dense(total_chars, activation='softmax')
 ])
 model.compile(loss='categorical_crossentropy', optimizer='adam',
metrics=['accuracy'])
 return model
def train_model(model, X, y, epochs=20, batch_size=128):
 model.fit(X, y, epochs=epochs, batch_size=batch_size, verbose=1)
def generate_text(model, tokenizer, total_chars, seed_text, seq_length,
num_chars):
 reverse_char_index = {i: char for char, i in tokenizer.char_index.items()}
 generated_text = seed_text
 for _ in range(num_chars):
 x_pred = pad_sequences([tokenizer.texts_to_sequences([seed_text])[0]],
maxlen=seq_length)
 pred_probs = model.predict(x_pred, verbose=0)[0]
 next_index = np.argmax(pred_probs)
 next_char = reverse_char_index[next_index]
 generated_text += next_char
 seed_text = seed_text[1:] + next_char
 return generated_text
def main():
 text = load_text('shakespeare.txt')
 seq_length = 40
```



```
 X, y, tokenizer, total_chars = prepare_data(text, seq_length)
 model = build_model(seq_length, total_chars)
 train_model(model, X, y, epochs=20, batch_size=128)
 seed_text = text[:seq_length]
 generated_text = generate_text(model, tokenizer, total_chars, seed_text,
seq_length, num_chars=400)
 print(generated_text)
if __name__ == "__main__":
 main()
```

Output

7. Q-learning algorithm to solve the FrozenLake environment

program

```
import gym
import numpy as np
env = gym.make('FrozenLake-v1', is_slippery=True, render_mode=None)
learning_rate = 0.1
discount_factor = 0.99
epsilon = 1.0 # Exploration rate
max_epsilon = 1.0
min_epsilon = 0.01
decay_rate = 0.005 # Decay rate for exploration probability
state_size = env.observation_space.n
action_size = env.action_space.n
Department of Artificial Intelligence and Data Science
q_table = np.zeros((state_size, action_size))
num_episodes = 10000
max_steps = 100
for episode in range(num_episodes):
 state = env.reset()[0]
 done = False
 for step in range(max_steps):
 if np.random.uniform(0, 1) < epsilon:
 action = env.action_space.sample() # Explore
 else:
 action = np.argmax(q_table[state, :]) # Exploit
 new_state, reward, done, truncated, info = env.step(action)
 q_table[state, action] = q_table[state, action] + learning_rate * (
 reward + discount_factor * np.max(q_table[new_state, :]) -
q_table[state, action])
 state = new_state
 if done:
 break
 epsilon = min_epsilon + (max_epsilon - min_epsilon) * np.exp(-decay_rate *
episode)
num_test_episodes = 100
total_rewards = 0
for episode in range(num_test_episodes):
 state = env.reset()[0]
 done = False
 episode_rewards = 0
 for step in range(max_steps):
 action = np.argmax(q_table[state, :]) # Choose best action
Department of Artificial Intelligence and Data Science
 new_state, reward, done, truncated, info = env.step(action)
 episode_rewards += reward
 state = new_state
 if done:
 break
 total_rewards += episode_rewards
print(f"Average reward over {num_test_episodes} test episodes: {total_rewards
/ num_test_episodes}")
```

```
env.close()
```

Output

8. Q-values in a Deep Q-Network (DQN)

program

```
import gym
import numpy as np
import random
from collections import deque
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import Adam
EPISODES = 1000
STATE_SIZE = 4
ACTION_SIZE = 2
GAMMA = 0.95
EPSILON = 1.0
EPSILON_MIN = 0.01
EPSILON_DECAY = 0.995
LEARNING_RATE = 0.001
BATCH_SIZE = 32
MEMORY_SIZE = 2000
TARGET_UPDATE = 10
class DQNAgent:
 def __init__(self):
 self.memory = deque(maxlen=MEMORY_SIZE)
 self.epsilon = EPSILON
 self.model = self._build_model()
 self.target_model = self._build_model()
```

Department of Artificial Intelligence and Data Science

```
 self.update_target_model()
 def _build_model(self):
 model = Sequential()
 model.add(Dense(24, input_dim=STATE_SIZE, activation='relu'))
 model.add(Dense(24, activation='relu'))
 model.add(Dense(ACTION_SIZE, activation='linear'))
 model.compile(loss='mse',
optimizer=Adam(learning_rate=LEARNING_RATE))
 return model
 def update_target_model(self):
 self.target_model.set_weights(self.model.get_weights())
 def remember(self, state, action, reward, next_state, done):
 self.memory.append((state, action, reward, next_state, done))
 def act(self, state):
 if np.random.rand() <= self.epsilon:
 return random.randrange(ACTION_SIZE)
 q_values = self.model.predict(state)
 return np.argmax(q_values[0])
 def replay(self):
```

```
 if len(self.memory) < BATCH_SIZE:
 return
 minibatch = random.sample(self.memory, BATCH_SIZE)
 for state, action, reward, next_state, done in minibatch:
 target = self.model.predict(state)
 if done:
 target[0][action] = reward
 else:
 t = self.target_model.predict(next_state)
```



```
 target[0][action] = reward + GAMMA * np.amax(t[0])
 self.model.fit(state, target, epochs=1, verbose=0)
 if self.epsilon > EPSILON_MIN:
 self.epsilon *= EPSILON_DECAY
 def load(self, name):
 self.model.load_weights(name)
 def save(self, name):
 self.model.save_weights(name)
if __name__ == "__main__":
 env = gym.make('CartPole-v1')
 agent = DQNAgent()
 done = False
 for e in range(EPISODES):
 state = env.reset()
 state = np.reshape(state, [1, STATE_SIZE])
 for time in range(500):
 action = agent.act(state)
 next_state, reward, done, _ = env.step(action)
 reward = reward if not done else -10
 next_state = np.reshape(next_state, [1, STATE_SIZE])
 agent.remember(state, action, reward, next_state, done)
 state = next_state
 if done:
 agent.update_target_model()
 print(f"Episode: {e}/{EPISODES}, score: {time}, epsilon:
{agent.epsilon:.2}")
 break
 agent.replay()
```

9. Policy gradient method (REINFORCE) to solve the LunarLander environment

program

```
import gym
import numpy as np
```



```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import Adam
EPISODES = 1000
```

```
LEARNING_RATE = 0.001
GAMMA = 0.99
class REINFORCEAgent:
 def __init__(self, state_size, action_size):
 self.state_size = state_size
 self.action_size = action_size
 self.model = self._build_model()
 self.optimizer = Adam(learning_rate=LEARNING_RATE)
 def _build_model(self):
 model = Sequential()
 model.add(Dense(24, input_dim=self.state_size, activation='relu'))
 model.add(Dense(24, activation='relu'))
 model.add(Dense(self.action_size, activation='softmax'))
 return model
 def choose_action(self, state):
 state = state[np.newaxis, :]
 probabilities = self.model.predict(state, verbose=0)[0]
 return np.random.choice(self.action_size, p=probabilities)
 def compute_discounted_rewards(self, rewards):
 discounted_rewards = np.zeros_like(rewards, dtype=np.float32)
 cumulative_reward = 0
```



```
 for t in reversed(range(len(rewards))):
 cumulative_reward = rewards[t] + GAMMA * cumulative_reward
 discounted_rewards[t] = cumulative_reward
 return discounted_rewards
 def train(self, states, actions, discounted_rewards):
 with tf.GradientTape() as tape:
 action_probs = self.model(states, training=True)
 action_indices = tf.range(tf.shape(actions)[0]) * self.action_size + actions
 selected_action_probs = tf.gather(tf.reshape(action_probs, [-1]),
action_indices)
 loss = -tf.reduce_mean(tf.math.log(selected_action_probs) *
discounted_rewards)
 gradients = tape.gradient(loss, self.model.trainable_variables)
 self.optimizer.apply_gradients(zip(gradients,
self.model.trainable_variables))
if __name__ == "__main__":
 env = gym.make('LunarLander-v2')
 state_size = env.observation_space.shape[0]
 action_size = env.action_space.n
 agent = REINFORCEAgent(state_size, action_size)
 for episode in range(EPISODES):
 state = env.reset()
 states, actions, rewards = [], [], []
 while True:
 action = agent.choose_action(state)
 next_state, reward, done, _ = env.step(action)
 states.append(state)
```



```
 actions.append(action)
 rewards.append(reward)
```

```
 state = next_state
 if done:
 states = np.array(states)
 actions = np.array(actions)
 discounted_rewards = agent.compute_discounted_rewards(rewards)
 discounted_rewards -= np.mean(discounted_rewards)
 discounted_rewards /= np.std(discounted_rewards) + 1e-8
 agent.train(states, actions, discounted_rewards)
 print(f"Episode: {episode + 1}, Reward: {sum(rewards)}")
 break
 env.close()
```

10.Imitation learning to train an autonomous vehicle agent

program

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.optimizers import Adam
from sklearn.model_selection import train_test_split
def load_data():
 num_samples = 10000
 state_dim = 10 # Example state dimensions (e.g., sensor readings)
 action_dim = 3 # Example action dimensions (e.g., steer, throttle, brake)
 states = np.random.rand(num_samples, state_dim)
 actions = np.random.rand(num_samples, action_dim)
 return states, actions
```

```
def build_model(state_dim, action_dim):
 model = Sequential()
 model.add(Dense(64, input_dim=state_dim, activation='relu'))
 model.add(Dropout(0.1))
 model.add(Dense(64, activation='relu'))
 model.add(Dense(action_dim, activation='linear'))
 model.compile(optimizer=Adam(learning_rate=0.001), loss='mse')
 return model
if __name__ == "__main__":
 states, actions = load_data()
 states_train, states_val, actions_train, actions_val = train_test_split(states,
actions, test_size=0.2, random_state=42)
 model = build_model(states.shape[1], actions.shape[1])
 model.fit(states_train, actions_train, validation_data=(states_val, actions_val),
epochs=50, batch_size=32)
 loss = model.evaluate(states_val, actions_val)
 print(f"Validation Loss: {loss}")
 new_state = np.random.rand(1, states.shape[1]) # Dummy state
 predicted_action = model.predict(new_state)
 print(f"Predicted Action: {predicted_action}")
```

11.Simplified version of ChaufferNet

program

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Conv2D, Flatten, Dropout
from tensorflow.keras.optimizers import Adam
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
def load_data():
 num_samples = 1000
 img_height, img_width, img_channels = 64, 64, 3
```

Department of Artificial Intelligence and Data Science

```
features = np.random.rand(num_samples, img_height, img_width,
img_channels)
 steering_angles = np.random.rand(num_samples) * 2 - 1 # Range [-1, 1]
 return features, steering_angles
def build_model(input_shape):
 model = Sequential()
 model.add(Conv2D(24, (5, 5), strides=(2, 2), activation='relu',
input_shape=input_shape))
 model.add(Dropout(0.2))
 model.add(Conv2D(36, (5, 5), strides=(2, 2), activation='relu'))
 model.add(Dropout(0.2))
 model.add(Conv2D(48, (5, 5), strides=(2, 2), activation='relu'))
 model.add(Dropout(0.2))
 model.add(Flatten())
 model.add(Dense(100, activation='relu'))
 model.add(Dropout(0.2))
 model.add(Dense(50, activation='relu'))
 model.add(Dense(10, activation='relu'))
 model.add(Dense(1))
 model.compile(optimizer=Adam(learning_rate=0.001), loss='mse')
 return model
if __name__ == "__main__":
 features, steering_angles = load_data()
 features_train, features_val, angles_train, angles_val =
train_test_split(features, steering_angles, test_size=0.2, random_state=42)
 input_shape = features_train.shape[1:]
 model = build_model(input_shape)
 model.fit(features_train, angles_train, validation_data=(features_val,
angles_val), epochs=10, batch_size=32)
```

Department of Artificial Intelligence and Data Science

```
 loss = model.evaluate(features_val, angles_val)
 print(f"Validation Loss: {loss}")
 new_feature = np.random.rand(1, *input_shape)
 predicted_angle = model.predict(new_feature)
 print(f"Predicted Steering Angle: {predicted_angle[0][0]}")
```

12.End-to-End Deep Learning for Autonomous Driving

program

```
import os
import numpy as np
import cv2
import csv
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, Flatten, Dense, Lambda,
Dropout
from tensorflow.keras.optimizers import Adam
from sklearn.model_selection import train_test_split
from sklearn.utils import shuffle
def load_data(data_dir):
 images = []
 steering_angles = []
 with open(os.path.join(data_dir, 'driving_log.csv')) as csvfile:
 reader = csv.reader(csvfile)
 next(reader)
```



```
 for line in reader:
center_image_path = os.path.join(data_dir, 'IMG', line[0].split('/')[-1])
 center_image = cv2.imread(center_image_path)
 center_image = cv2.cvtColor(center_image, cv2.COLOR_BGR2RGB)
 center_image = cv2.resize(center_image, (200, 66))
 images.append(center_image)
 steering_angles.append(float(line[3])) # Steering angle
 return np.array(images), np.array(steering_angles)
def build_model(input_shape):
 model = Sequential()
 model.add(Lambda(lambda x: x / 255.0 - 0.5,
input_shape=input_shape))
 model.add(Conv2D(24, (5, 5), strides=(2, 2), activation='relu'))
 model.add(Conv2D(36, (5, 5), strides=(2, 2), activation='relu'))
 model.add(Conv2D(48, (5, 5), strides=(2, 2), activation='relu'))
 model.add(Conv2D(64, (3, 3), activation='relu'))
 model.add(Conv2D(64, (3, 3), activation='relu'))
 model.add(Flatten())
 model.add(Dense(100, activation='relu'))
 model.add(Dropout(0.5))
 model.add(Dense(50, activation='relu'))
 model.add(Dense(10, activation='relu'))
 model.add(Dense(1)) # Output layer for steering angle
 model.compile(optimizer=Adam(learning_rate=0.0001), loss='mse')
 return model
if __name__ == "__main__":
```



```
 data_dir = 'path_to_simulator_data' # Replace with your data directory
 images, steering_angles = load_data(data_dir)
```

```
 X_train, X_val, y_train, y_val = train_test_split(images, 
steering_angles, test_size=0.2, random_state=42)
 input_shape = X_train.shape[1:]
 model = build_model(input_shape)
 model.fit(X_train, y_train, validation_data=(X_val, y_val), epochs=10, 
batch_size=32, shuffle=True)
 loss = model.evaluate(X_val, y_val)
 print(f"Validation Loss: {loss}")
 model.save('autonomous_driving_model.h5')
 new_image = np.expand_dims(X_val[0], axis=0)
 predicted_angle = model.predict(new_image)
 print(f"Predicted Steering Angle: {predicted_angle[0][0]}")
```