Deep Learning Exam - Bilet 2 Task Solutions

# Task 1: Anomaly Detection - Autoencoder

In this task, we apply an Autoencoder-based approach for anomaly detection. Autoencoders are unsupervised neural networks that learn to compress data (encoding) and then reconstruct it (decoding).   
During inference, inputs that produce high reconstruction errors are flagged as anomalies. This method is useful for detecting unusual patterns in time series data such as network traffic or sensor readings.

Code:

import numpy as np  
import tensorflow as tf  
from tensorflow.keras.models import Model  
from tensorflow.keras.layers import Input, Dense  
  
# Sample data (e.g., network traffic or sensor readings)  
normal\_data = np.random.normal(0, 1, (1000, 10)) # Normal data  
anomalous\_data = np.random.normal(0, 10, (100, 10)) # Anomalous data  
  
# Define Autoencoder model  
input\_dim = normal\_data.shape[1]  
input\_layer = Input(shape=(input\_dim,))  
encoded = Dense(16, activation='relu')(input\_layer)  
encoded = Dense(8, activation='relu')(encoded)  
decoded = Dense(16, activation='relu')(encoded)  
output\_layer = Dense(input\_dim, activation='sigmoid')(decoded)  
  
autoencoder = Model(inputs=input\_layer, outputs=output\_layer)  
  
# Compile the model  
autoencoder.compile(optimizer='adam', loss='mse')  
  
# Train on normal data  
autoencoder.fit(normal\_data, normal\_data, epochs=50, batch\_size=32, shuffle=True)  
  
# Test on both normal and anomalous data  
reconstructions = autoencoder.predict(np.concatenate([normal\_data, anomalous\_data]))  
mse = np.mean(np.power(np.concatenate([normal\_data, anomalous\_data]) - reconstructions, 2), axis=1)  
  
# Set a threshold for anomaly detection  
threshold = np.mean(mse) + 2\*np.std(mse)  
anomalies = mse > threshold  
  
# Output: Boolean array where True indicates an anomaly  
print(anomalies)

# Task 2: Reinforcement Learning - DQN with Different Reward Structures

In this task, we create a Deep Q-Network (DQN) agent that learns to play a game by interacting with the environment.   
By modifying the reward structure, we can influence the agent’s learning process. In this example, we use the CartPole environment and adjust the rewards based on the agent's state to encourage better performance.

Code:

import gym  
import numpy as np  
import tensorflow as tf  
from tensorflow.keras import Sequential  
from tensorflow.keras.layers import Dense  
from collections import deque  
import random  
  
# Environment setup (using CartPole as an example game)  
env = gym.make('CartPole-v1')  
state\_size = env.observation\_space.shape[0]  
action\_size = env.action\_space.n  
  
# Define DQN model  
def create\_dqn\_model():  
 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=tf.keras.optimizers.Adam(lr=0.001))  
 return model  
  
# Reward structure (modification of reward based on game dynamics)  
def reward\_function(state, reward, done):  
 if done: # Penalize falling  
 reward -= 10  
 if state[2] > 0: # Encourage moving the pole upright  
 reward += 1  
 return reward  
  
# DQN Agent class  
class DQNAgent:  
 def \_\_init\_\_(self):  
 self.model = create\_dqn\_model()  
  
 def act(self, state):  
 if np.random.rand() <= epsilon:  
 return random.randrange(action\_size)  
 act\_values = self.model.predict(state)  
 return np.argmax(act\_values[0])  
  
 def replay(self, batch):  
 for state, action, reward, next\_state, done in batch:  
 target = reward  
 if not done:  
 target = reward + gamma \* np.amax(self.model.predict(next\_state)[0])  
 target\_f = self.model.predict(state)  
 target\_f[0][action] = target  
 self.model.fit(state, target\_f, epochs=1, verbose=0)  
  
# Training loop  
agent = DQNAgent()  
for episode in range(1000):  
 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\_function(next\_state, reward, done)  
 next\_state = np.reshape(next\_state, [1, state\_size])  
 memory.append((state, action, reward, next\_state, done))  
 state = next\_state  
 if done:  
 break  
 if len(memory) > batch\_size:  
 minibatch = random.sample(memory, batch\_size)  
 agent.replay(minibatch)  
 if epsilon > epsilon\_min:  
 epsilon \*= epsilon\_decay

# Task 3: Generative Adversarial Networks (GANs) - Conditional GAN

In this task, we use a Conditional GAN (cGAN) to generate specific types of animals. cGANs are an extension of GANs that allow us to condition the generation process on specific labels, enabling control over the type of output generated.

Code:

import tensorflow as tf  
from tensorflow.keras.layers import Dense, Reshape, Flatten, Dropout, BatchNormalization, LeakyReLU, Embedding, Concatenate, Input  
from tensorflow.keras.models import Sequential, Model  
  
# Generator model for cGAN  
def build\_generator(noise\_dim, num\_classes):  
 label = Input(shape=(1,))  
 label\_embedding = Flatten()(Embedding(num\_classes, noise\_dim)(label))  
  
 noise = Input(shape=(noise\_dim,))  
 model\_input = Concatenate()([noise, label\_embedding])  
  
 x = Dense(128)(model\_input)  
 x = LeakyReLU(alpha=0.2)(x)  
 x = BatchNormalization()(x)  
 x = Dense(256)(x)  
 x = LeakyReLU(alpha=0.2)(x)  
 x = BatchNormalization()(x)  
 x = Dense(512)(x)  
 x = LeakyReLU(alpha=0.2)(x)  
 x = BatchNormalization()(x)  
 x = Dense(1024)(x)  
 x = LeakyReLU(alpha=0.2)(x)  
   
 output = Dense(28\*28\*1, activation='tanh')(x)  
 output = Reshape((28, 28, 1))(output)  
  
 return Model([noise, label], output)  
  
# Discriminator model for cGAN  
def build\_discriminator(num\_classes):  
 label = Input(shape=(1,))  
 label\_embedding = Flatten()(Embedding(num\_classes, 28\*28)(label))  
 label\_embedding = Reshape((28, 28, 1))(label\_embedding)  
  
 image = Input(shape=(28, 28, 1))  
 model\_input = Concatenate()([image, label\_embedding])  
  
 x = Flatten()(model\_input)  
 x = Dense(512)(x)  
 x = LeakyReLU(alpha=0.2)(x)  
 x = Dropout(0.4)(x)  
 x = Dense(256)(x)  
 x = LeakyReLU(alpha=0.2)(x)  
 x = Dropout(0.4)(x)  
  
 output = Dense(1, activation='sigmoid')(x)  
 return Model([image, label], output)