SET-1:1Q

```
import numpy as np
from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import accuracy_score, classification_report

print("Downloading MNIST…")
X, y = fetch_openml('mnist_784', version=1, return_X_y=True, as_frame=False)
y = y.astype(int)

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=1/7, random_state=42, stratify=y
)

scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled  = scaler.transform(X_test)

configs = [
    {
        'name': 'Config 1: [100] ReLU, solver=adam',
        'hidden_layer_sizes': (100,),
        'activation': 'relu',
        'solver': 'adam'
```

```
},
{
    'name': 'Config 2: [100,50] ReLU, solver=adam',
    'hidden_layer_sizes': (100, 50),
    'activation': 'relu',
    'solver': 'adam'
},
{
    'name': 'Config 3: [100] tanh, solver=sgd',
    'hidden_layer_sizes': (100,),
    'activation': 'tanh',
    'solver': 'sgd',
    'learning_rate_init': 0.01
},
{
    'name': 'Config 4: [200,100] ReLU, solver=adam',
    'hidden_layer_sizes': (200, 100),
    'activation': 'relu',
    'solver': 'adam'
},
{
    'name': 'Config 5: [50] logistic, solver=adam',
    'hidden_layer_sizes': (50,),
    'activation': 'logistic',
    'solver': 'adam'
}
```

```
]

for cfg in configs:
    print("\n" + "="*60)
    print(cfg['name'])
    mlp = MLPClassifier(
        hidden_layer_sizes=cfg['hidden_layer_sizes'],
        activation=cfg['activation'],
        solver=cfg['solver'],
        learning_rate_init=cfg.get('learning_rate_init', 0.001),
        max_iter=50,
        random_state=42,
        verbose=False
    )

    mlp.fit(X_train_scaled, y_train)

    y_pred = mlp.predict(X_test_scaled)
    acc = accuracy_score(y_test, y_pred)
    print(f"Test accuracy: {acc:.4f}")
    print("Classification report (first 5 classes):")
    print(classification_report(y_test, y_pred, labels=[0,1,2,3,4]))
```

SET-2:1Q

```
import numpy as np
import pandas as pd
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

import tensorflow as tf
from tensorflow.keras import layers, callbacks, models

# 1. Prepare data
X, y = load_breast_cancer(return_X_y=True)
X_train, X_val, y_train, y_val = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y
)

results = []

# 2. L2 (Ridge) logistic regression
lr = LogisticRegression(penalty='l2', C=1.0, solver='liblinear', random_state=42)
lr.fit(X_train, y_train)
train_acc = accuracy_score(y_train, lr.predict(X_train))
val_acc   = accuracy_score(y_val,   lr.predict(X_val))
results.append({
    'Method': 'L2 (Ridge)',
    'Train Acc': train_acc,
    'Val Acc': val_acc,
    'Notes': 'penalty="l2", C=1.0'
})
```

```
# 3. Neural net with Dropout
def build_dropout_model(input_dim, dropout_rate=0.5):
    model = models.Sequential([
        layers.Dense(64, activation='relu', input_shape=(input_dim,)),
        layers.Dropout(dropout_rate),
        layers.Dense(32, activation='relu'),
        layers.Dropout(dropout_rate),
        layers.Dense(1, activation='sigmoid')
    ])
    model.compile(
        optimizer='adam',
        loss='binary_crossentropy',
        metrics=['accuracy']
    )
    return model

dropout_model = build_dropout_model(X_train.shape[1], dropout_rate=0.5)
history_drop = dropout_model.fit(
    X_train, y_train,
    epochs=100,
    batch_size=32,
    validation_data=(X_val, y_val),
    verbose=0
)
results.append({
    'Method': 'Dropout Neural Net',
    'Train Acc': history_drop.history['accuracy'][-1],
    'Val Acc':   history_drop.history['val_accuracy'][-1],
```

```
    'Notes': 'dropout_rate=0.5'
})

# 4. Neural net with Early Stopping
early_stop_cb = callbacks.EarlyStopping(
    monitor='val_loss',
    patience=5,
    restore_best_weights=True
)
es_model = build_dropout_model(X_train.shape[1], dropout_rate=0.0)
history_es = es_model.fit(
    X_train, y_train,
    epochs=100,
    batch_size=32,
    validation_data=(X_val, y_val),
    callbacks=[early_stop_cb],
    verbose=0
)
results.append({
    'Method': 'Early Stopping NN',
    'Train Acc': history_es.history['accuracy'][-1],
    'Val Acc':   history_es.history['val_accuracy'][-1],
    'Notes': f'early_stop patience=5, epochs run={len(history_es.history["loss"])}'
})

# 5. Neural net with Gaussian Noise
def build_noise_model(input_dim, noise_std=0.1):
    model = models.Sequential([
        layers.GaussianNoise(noise_std, input_shape=(input_dim,)),
```

```
        layers.Dense(64, activation='relu'),
        layers.Dense(32, activation='relu'),
        layers.Dense(1, activation='sigmoid')
    ])
    model.compile(
        optimizer='adam',
        loss='binary_crossentropy',
        metrics=['accuracy']
    )
    return model
noise_model = build_noise_model(X_train.shape[1], noise_std=0.1)
history_noise = noise_model.fit(
    X_train, y_train,
    epochs=100,
    batch_size=32,
    validation_data=(X_val, y_val),
    callbacks=[early_stop_cb],
    verbose=0
)
results.append({
    'Method': 'Gaussian Noise NN',
    'Train Acc': history_noise.history['accuracy'][-1],
    'Val Acc':   history_noise.history['val_accuracy'][-1],
    'Notes': f'noise_std=0.1, epochs run={len(history_noise.history["loss"])}'
})
# 6. Summarize results
df = pd.DataFrame(results)
print(df[['Method', 'Train Acc', 'Val Acc', 'Notes']].to_markdown(index=False))
```

SET-3:

```
#set -3 1 '''Implementation of RNN, LSTM, and GRU for Sentiment Analysis on the IMDb dataset'''

import numpy as np
import matplotlib.pyplot as plt
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, SimpleRNN, LSTM, GRU, Dense, Dropout
from tensorflow.keras.callbacks import EarlyStopping
import pandas as pd

# 1. Load and preprocess data
vocab_size = 10000
max_len = 200
(x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=vocab_size)
x_train = pad_sequences(x_train, maxlen=max_len, padding='post')
x_test  = pad_sequences(x_test,  maxlen=max_len, padding='post')

# 2. Model builder

def build_model(rnn_type='SimpleRNN', embed_dim=128, rnn_units=64, dropout=0.2):
    model = Sequential()
    model.add(Embedding(vocab_size, embed_dim, input_length=max_len))
    if rnn_type == 'SimpleRNN':
        model.add(SimpleRNN(rnn_units))
    elif rnn_type == 'LSTM':
        model.add(LSTM(rnn_units))
    elif rnn_type == 'GRU':
        model.add(GRU(rnn_units))
```

```
    else:
        raise ValueError("Invalid rnn_type")
    model.add(Dropout(dropout))
    model.add(Dense(1, activation='sigmoid'))
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
    return model

# 3. Train and evaluate each RNN type
results = []
for rnn_type in ['SimpleRNN', 'LSTM', 'GRU']:
    print(f"\nTraining {rnn_type}...")
    model = build_model(rnn_type=rnn_type)
    es = EarlyStopping(monitor='val_loss', patience=3, restore_best_weights=True)
    history = model.fit(
        x_train, y_train,
        validation_split=0.2,
        epochs=10,
        batch_size=128,
        callbacks=[es],
        verbose=2
    )
    # Plot training curves
    plt.figure(figsize=(12,4))
    # accuracy
    plt.subplot(1,2,1)
    plt.plot(history.history['accuracy'], label='train_acc')
    plt.plot(history.history['val_accuracy'], label='val_acc')
    plt.title(f'{rnn_type} Accuracy')
    plt.legend()
```

```
    # loss
    plt.subplot(1,2,2)
    plt.plot(history.history['loss'], label='train_loss')
    plt.plot(history.history['val_loss'], label='val_loss')
    plt.title(f'{rnn_type} Loss')
    plt.legend()
    plt.show()

    # Test performance
    loss, acc = model.evaluate(x_test, y_test, verbose=0)
    print(f'{rnn_type} Test accuracy: {acc:.4f}, loss: {loss:.4f}')

    # Sample predictions
    sample_idxs = np.random.choice(len(x_test), 5, replace=False)
    sample_texts = x_test[sample_idxs]
    sample_labels = y_test[sample_idxs]
    sample_preds = (model.predict(sample_texts) > 0.5).astype(int).flatten()
    for i, idx in enumerate(sample_idxs):
        print(f'Sample {i+1}: True={sample_labels[i]}, Pred={sample_preds[i]}')

    results.append({'Model': rnn_type, 'Test Accuracy': acc, 'Test Loss': loss})

# 4. Comparison table
results_df = pd.DataFrame(results)
print("\nComparison of RNN Types:\n", results_df)
```

SET-4:

```
# set 4# IMPLEMENTATION OF DENOISING AUTOENCODERS
```

```
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.datasets import mnist
import numpy as np
import matplotlib.pyplot as plt
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0
x_train = np.expand_dims(x_train, axis=-1)
x_test = np.expand_dims(x_test, axis=-1)

def add_noise(images, noise_factor=0.5):
    noise = np.random.normal(scale=noise_factor, size=images.shape)
    noisy_images = images + noise
    noisy_images = np.clip(noisy_images, 0., 1.)
    return noisy_images

x_train_noisy = add_noise(x_train)
x_test_noisy = add_noise(x_test)
input_img = layers.Input(shape=(28, 28, 1))
x = layers.Conv2D(32, (3, 3), activation='relu', padding='same')(input_img)
x = layers.MaxPool2D((2, 2), padding='same')(x)
x = layers.Conv2D(64, (3, 3), activation='relu', padding='same')(x)
x = layers.MaxPool2D((2, 2), padding='same')(x)
x = layers.Conv2D(64, (3, 3), activation='relu', padding='same')(x)
x = layers.UpSampling2D((2, 2))(x)
x = layers.Conv2D(32, (3, 3), activation='relu', padding='same')(x)
x = layers.UpSampling2D((2, 2))(x)
decoded_img = layers.Conv2D(1, (3, 3), activation='sigmoid', padding='same')(x)
```

```
autoencoder = models.Model(input_img, decoded_img)

encoder = models.Model(input_img, x)

encoded_input = layers.Input(shape=(7, 7, 64))

decoder_layer = autoencoder.layers[-3](encoded_input)

decoder_layer = autoencoder.layers[-2](decoder_layer)

decoder_layer = autoencoder.layers[-1](decoder_layer)

decoder = models.Model(encoded_input, decoder_layer)

autoencoder.compile(optimizer='adam', loss='binary_crossentropy')

autoencoder.fit(x_train_noisy, x_train, epochs=10, batch_size=128,
validation_data=(x_test_noisy, x_test))

decoded_imgs = autoencoder.predict(x_test_noisy)

n = 10

plt.figure(figsize=(20, 6))

for i in range(n):

    ax = plt.subplot(3, n, i + 1)

    plt.imshow(x_test[i].reshape(28, 28), cmap='gray')

    ax.set_title("Original")

    ax.axis('off')

    ax = plt.subplot(3, n, i + 1 + n)

    plt.imshow(x_test_noisy[i].reshape(28, 28), cmap='gray')

    ax.set_title("Noisy")

    ax.axis('off')

    ax = plt.subplot(3, n, i + 1 + 2*n)

    plt.imshow(decoded_imgs[i].reshape(28, 28), cmap='gray')

    ax.set_title("Denoised")

    ax.axis('off')

plt.show()
```

SET-5:

```
# set 5 IMPLEMENTATION OF ENCODER DECODER MODELS
```

```
import numpy as np
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, LSTM, Dense

text_pairs = [
    ("I love Python", "J'adore Python"),
    ("Hello world", "Bonjour le monde"),
    ("Good morning", "Bonjour"),
    ("How are you", "Comment allez-vous"),
    ("See you later", "A plus tard")
]

input_texts = [pair[0] for pair in text_pairs]
target_texts = ['\t' + pair[1] + '\n' for pair in text_pairs]

input_characters = sorted(list(set(''.join(input_texts))))
target_characters = sorted(list(set(''.join(target_texts))))
num_encoder_tokens = len(input_characters)
num_decoder_tokens = len(target_characters)
max_encoder_seq_length = max([len(txt) for txt in input_texts])
max_decoder_seq_length = max([len(txt) for txt in target_texts])

input_token_index = dict([(char, i) for i, char in enumerate(input_characters)])
target_token_index = dict([(char, i) for i, char in enumerate(target_characters)])
reverse_target_char_index = dict((i, char) for char, i in target_token_index.items())

encoder_input_data = np.zeros((len(input_texts), max_encoder_seq_length,
num_encoder_tokens), dtype='float32')
decoder_input_data = np.zeros((len(input_texts), max_decoder_seq_length,
num_decoder_tokens), dtype='float32')
```

```
decoder_target_data = np.zeros((len(input_texts), max_decoder_seq_length,
num_decoder_tokens), dtype='float32')

for i, (input_text, target_text) in enumerate(zip(input_texts, target_texts)):
    for t, char in enumerate(input_text):
        encoder_input_data[i, t, input_token_index[char]] = 1.0
    for t, char in enumerate(target_text):
        decoder_input_data[i, t, target_token_index[char]] = 1.0
        if t > 0:
            decoder_target_data[i, t - 1, target_token_index[char]] = 1.0

# Model building
embedding_size = 256
lstm_units = 256

encoder_inputs = Input(shape=(None, num_encoder_tokens))
encoder_lstm = LSTM(lstm_units, return_state=True)
encoder_outputs, state_h, state_c = encoder_lstm(encoder_inputs)
encoder_states = [state_h, state_c]

decoder_inputs = Input(shape=(None, num_decoder_tokens))
decoder_lstm = LSTM(lstm_units, return_sequences=True, return_state=True)
decoder_outputs, _, _ = decoder_lstm(decoder_inputs, initial_state=encoder_states)
decoder_dense = Dense(num_decoder_tokens, activation='softmax')
decoder_outputs = decoder_dense(decoder_outputs)

model = Model([encoder_inputs, decoder_inputs], decoder_outputs)
model.compile(optimizer='rmsprop', loss='categorical_crossentropy', metrics=['accuracy'])
model.fit([encoder_input_data, decoder_input_data], decoder_target_data, batch_size=64,
epochs=100, validation_split=0.2)
```

```
# Inference Models
encoder_model = Model(encoder_inputs, encoder_states)

decoder_state_input_h = Input(shape=(lstm_units,))
decoder_state_input_c = Input(shape=(lstm_units,))
decoder_states_inputs = [decoder_state_input_h, decoder_state_input_c]

decoder_outputs, state_h, state_c = decoder_lstm(decoder_inputs,
initial_state=decoder_states_inputs)
decoder_states = [state_h, state_c]
decoder_outputs = decoder_dense(decoder_outputs)

decoder_model = Model([decoder_inputs] + decoder_states_inputs, [decoder_outputs] +
decoder_states)

# Translation Function
def translate(input_sentence):
    input_seq = np.zeros((1, max_encoder_seq_length, num_encoder_tokens),
dtype='float32')
    for t, char in enumerate(input_sentence):
        if char in input_token_index:
            input_seq[0, t, input_token_index[char]] = 1.0

    states_value = encoder_model.predict(input_seq)

    target_seq = np.zeros((1, 1, num_decoder_tokens))
    target_seq[0, 0, target_token_index['\t']] = 1.0

    decoded_sentence = ''
```

```
    stop_condition = False
    while not stop_condition:
        output_tokens, h, c = decoder_model.predict([target_seq] + states_value)
        sampled_token_index = np.argmax(output_tokens[0, -1, :])
        sampled_char = reverse_target_char_index[sampled_token_index]
        decoded_sentence += sampled_char

        if sampled_char == '\n' or len(decoded_sentence) > max_decoder_seq_length:
            stop_condition = True

        target_seq = np.zeros((1, 1, num_decoder_tokens))
        target_seq[0, 0, sampled_token_index] = 1.0
        states_value = [h, c]

    return decoded_sentence.strip()

# Test Translations
test_sentences = ["Hello world", "Good morning", "I love Python"]
print("\nTranslations:")
for sentence in test_sentences:
    translation = translate(sentence)
    print(f"Input: {sentence}")
    print(f"Output: {translation}\n")
```

SET-6:

```
#set 6 MLP classifier using tensorflow - SGD,momnetum,ADAM,nestrov,ADAMoptimiser
import tensorflow as tf
import time
```

```
import numpy as np
import matplotlib.pyplot as plt
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

(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0
y_train_cat, y_test_cat = to_categorical(y_train), to_categorical(y_test)

def create_mlp_model():
    model = Sequential([
        Flatten(input_shape=(28, 28)),
        Dense(128, activation='relu'),
        Dense(64, activation='relu'),
        Dense(10, activation='softmax')
    ])
    return model

optimizers = {
    "SGD": tf.keras.optimizers.SGD(),
    "SGD with Momentum": tf.keras.optimizers.SGD(momentum=0.9),
    "Adam": tf.keras.optimizers.Adam(),
    "SGD with Nesterov": tf.keras.optimizers.SGD(momentum=0.9, nesterov=True),
    "Adam LR=0.0005": tf.keras.optimizers.Adam(learning_rate=0.0005)
}

results = {}
```

```
for name, optimizer in optimizers.items():
    print(f"\nTraining with {name}...")

    tf.keras.backend.clear_session()
    model = create_mlp_model()
    model.compile(optimizer=optimizer, loss='categorical_crossentropy', metrics=['accuracy'])

    start = time.time()
    history = model.fit(x_train, y_train_cat, validation_split=0.1, epochs=2, batch_size=128,
verbose=0)
    end = time.time()

    train_acc = history.history['accuracy'][-1]
    test_loss, test_acc = model.evaluate(x_test, y_test_cat, verbose=0)
    convergence_time = end - start

    results[name] = {
        'Train Accuracy': train_acc,
        'Test Accuracy': test_acc,
        'Time': convergence_time,
        'History': history.history
    }

    print(f"Train Accuracy: {train_acc:.4f}")
    print(f"Test Accuracy: {test_acc:.4f}")
    print(f"Convergence Time: {convergence_time:.2f}s")

print("\n\n=== Comparison Table ===")
print(f"{'Configuration':<25} {'Train Acc':<12} {'Test Acc':<12} {'Time (s)':<10}")
```

```
for key, val in results.items():
    print(f"{key:<25} {val['Train Accuracy']:.4f}      {val['Test Accuracy']:.4f}     {val['Time']:.2f}")
```

SET-7:

```
#set 7# Classification model using - l2,dropout,early stoppage, adding noise
import tensorflow as tf
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
```

```
from tensorflow.keras.layers import Dense, Flatten, Dropout, GaussianNoise
from tensorflow.keras.regularizers import l2
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.utils import to_categorical
import matplotlib.pyplot as plt
import time

(X_train, y_train), (X_test, y_test) = mnist.load_data()
X_train, X_test = X_train / 255.0, X_test / 255.0
y_train, y_test = to_categorical(y_train, 10), to_categorical(y_test, 10)

def build_model(regularization_type=None, reg_value=0.01):
    model = Sequential()
    model.add(Flatten(input_shape=(28, 28)))

    if regularization_type == 'noise':
        model.add(GaussianNoise(0.2))

    if regularization_type == 'l2':
        model.add(Dense(128, activation='relu', kernel_regularizer=l2(reg_value)))
        model.add(Dense(64, activation='relu', kernel_regularizer=l2(reg_value)))
    else:
        model.add(Dense(128, activation='relu'))
        if regularization_type == 'dropout':
            model.add(Dropout(0.5))
        model.add(Dense(64, activation='relu'))
        if regularization_type == 'dropout':
            model.add(Dropout(0.5))
```

```
    model.add(Dense(10, activation='softmax'))
    return model

methods = ['l2', 'dropout', 'early_stopping', 'noise']
results = {}

for method in methods:
    print(f"\nTraining with {method} regularization...")
    tf.keras.backend.clear_session()
    model = build_model(regularization_type=method if method != 'early_stopping' else
None)

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

    callbacks = []
    if method == 'early_stopping':
        callbacks.append(EarlyStopping(monitor='val_loss', patience=3,
restore_best_weights=True))

    start_time = time.time()
    history = model.fit(X_train, y_train, epochs=4, batch_size=64, verbose=0,
                        validation_split=0.2, callbacks=callbacks)
    end_time = time.time()

    train_acc = history.history['accuracy'][-1]
    val_acc = history.history['val_accuracy'][-1]
    test_loss, test_accuracy = model.evaluate(X_test, y_test, verbose=0)
    train_time = end_time - start_time
```

```
    results[method] = {
        "train_accuracy": train_acc,
        "val_accuracy": val_acc,
        "test_accuracy": test_accuracy,
        "train_time": train_time
    }

    print(f"Train Acc: {train_acc:.4f}, Val Acc: {val_acc:.4f}, Test Acc: {test_accuracy:.4f}, Time: {train_time:.2f}s")

print("\n\n--- Summary Table ---")
print(f"{'Method':<15} {'Train Acc':<12} {'Val Acc':<12} {'Test Acc':<12} {'Time(s)':<10}")
for method, data in results.items():
    print(f"{method:<15} {data['train_accuracy']:.4f} {data['val_accuracy']:.4f} {data['test_accuracy']:.4f} {data['train_time']:.2f}")
```

SET-8:

```
#set 8# Denoising Autoencoders
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.datasets import mnist
```

```
import numpy as np
import matplotlib.pyplot as plt
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0
x_train = np.expand_dims(x_train, axis=-1)
x_test = np.expand_dims(x_test, axis=-1)

def add_noise(images, noise_factor=0.5):
    noise = np.random.normal(scale=noise_factor, size=images.shape)
    noisy_images = images + noise
    noisy_images = np.clip(noisy_images, 0., 1.)
    return noisy_images

x_train_noisy = add_noise(x_train)
x_test_noisy = add_noise(x_test)

input_img = layers.Input(shape=(28, 28, 1))
x = layers.Conv2D(32, (3, 3), activation='relu', padding='same')(input_img)
x = layers.MaxPool2D((2, 2), padding='same')(x)
x = layers.Conv2D(64, (3, 3), activation='relu', padding='same')(x)
x = layers.MaxPool2D((2, 2), padding='same')(x)
x = layers.Conv2D(64, (3, 3), activation='relu', padding='same')(x)
x = layers.UpSampling2D((2, 2))(x)
x = layers.Conv2D(32, (3, 3), activation='relu', padding='same')(x)
x = layers.UpSampling2D((2, 2))(x)
decoded_img = layers.Conv2D(1, (3, 3), activation='sigmoid', padding='same')(x)
autoencoder = models.Model(input_img, decoded_img)
encoder = models.Model(input_img, x)
```

```
encoded_input = layers.Input(shape=(7, 7, 64))

decoder_layer = autoencoder.layers[-3](encoded_input)

decoder_layer = autoencoder.layers[-2](decoder_layer)

decoder_layer = autoencoder.layers[-1](decoder_layer)

decoder = models.Model(encoded_input, decoder_layer)

autoencoder.compile(optimizer='adam', loss='binary_crossentropy')

autoencoder.fit(x_train_noisy, x_train, epochs=2, batch_size=128,
validation_data=(x_test_noisy, x_test))

decoded_imgs = autoencoder.predict(x_test_noisy)

n = 10

plt.figure(figsize=(20, 6))

for i in range(n):

    ax = plt.subplot(3, n, i + 1)

    plt.imshow(x_test[i].reshape(28, 28), cmap='gray')

    ax.set_title("Original")

    ax.axis('off')

    ax = plt.subplot(3, n, i + 1 + n)

    plt.imshow(x_test_noisy[i].reshape(28, 28), cmap='gray')

    ax.set_title("Noisy")

    ax.axis('off')

    ax = plt.subplot(3, n, i + 1 + 2*n)

    plt.imshow(decoded_imgs[i].reshape(28, 28), cmap='gray')

    ax.set_title("Denoised")

    ax.axis('off')

plt.show()
```

SET-9:2Q:GAN

```
import torch

import torch.nn as nn
```

```
import torch.optim as optim

class Generator(nn.Module):
    def __init__(self, noise_dim, output_dim):
        super(Generator, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(noise_dim, 128),
            nn.ReLU(),
            nn.Linear(128, output_dim),
            nn.Tanh()
        )

    def forward(self, x):
        return self.model(x)

class Discriminator(nn.Module):
    def __init__(self, input_dim):
        super(Discriminator, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(input_dim, 128),
            nn.ReLU(),
            nn.Linear(128, 1),
            nn.Sigmoid()
        )

    def forward(self, x):
        return self.model(x)

noise_dim = 100
```

```
data_dim = 784

lr = 0.0002

epochs = 200

batch_size = 64


G = Generator(noise_dim, data_dim)

D = Discriminator(data_dim)

G_optimizer = optim.Adam(G.parameters(), lr=lr)

D_optimizer = optim.Adam(D.parameters(), lr=lr)


criterion = nn.BCELoss()


for epoch in range(epochs):

    real_data = torch.randn(batch_size, data_dim)

    real_labels = torch.ones(batch_size, 1)

    fake_labels = torch.zeros(batch_size, 1)


    D_optimizer.zero_grad()

    real_output = D(real_data)

    d_loss_real = criterion(real_output, real_labels)


    noise = torch.randn(batch_size, noise_dim)

    fake_data = G(noise)

    fake_output = D(fake_data.detach())

    d_loss_fake = criterion(fake_output, fake_labels)


    d_loss = d_loss_real + d_loss_fake

    d_loss.backward()

    D_optimizer.step()
```

```
    G_optimizer.zero_grad()
    fake_output = D(fake_data)
    g_loss = criterion(fake_output, real_labels)
    g_loss.backward()
    G_optimizer.step()

    print(f"Epoch {epoch}, D Loss: {d_loss.item()}, G Loss: {g_loss.item()}")

noise = torch.randn(10, noise_dim)
synthetic_data = G(noise)
```