1) import numpy as np

import matplotlib.pyplot as plt

from tensorflow.keras.datasets import mnist

from tensorflow.keras.models import Model

from tensorflow.keras.layers import Input, Dense

from tensorflow.keras.optimizers import Adam

from tensorflow.keras.losses import BinaryCrossentropy

# 1. Load the MNIST dataset

(x\_train, \_), (x\_test, \_) = mnist.load\_data()

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

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

x\_train = x\_train.reshape((len(x\_train), -1)) # Flatten to (num\_samples, 784)

x\_test = x\_test.reshape((len(x\_test), -1))

# Function to create autoencoder

def build\_autoencoder(latent\_dim):

input\_img = Input(shape=(784,))

# Encoder

encoded = Dense(latent\_dim, activation='relu')(input\_img)

# Decoder

decoded = Dense(784, activation='sigmoid')(encoded)

# Autoencoder model

autoencoder = Model(input\_img, decoded)

return autoencoder

# 3. Compile and train

def train\_autoencoder(latent\_dim, epochs=20):

autoencoder = build\_autoencoder(latent\_dim)

autoencoder.compile(optimizer=Adam(), loss=BinaryCrossentropy())

autoencoder.fit(x\_train, x\_train,

epochs=epochs,

batch\_size=256,

shuffle=True,

validation\_data=(x\_test, x\_test),

verbose=2)

return autoencoder

# 4. Plot original vs reconstructed images

def plot\_reconstructions(model, x\_test, num\_images=10):

decoded\_imgs = model.predict(x\_test)

plt.figure(figsize=(20, 4))

for i in range(num\_images):

# Original

ax = plt.subplot(2, num\_images, i + 1)

plt.imshow(x\_test[i].reshape(28, 28), cmap="gray")

plt.title("Original")

plt.axis("off")

# Reconstruction

ax = plt.subplot(2, num\_images, i + 1 + num\_images)

plt.imshow(decoded\_imgs[i].reshape(28, 28), cmap="gray")

plt.title("Reconstructed")

plt.axis("off")

plt.show()

# 5. Experiment with different latent dimensions

for latent\_dim in [16, 32, 64]:

print(f"\nTraining Autoencoder with latent dimension: {latent\_dim}")

autoencoder = train\_autoencoder(latent\_dim)

plot\_reconstructions(autoencoder, x\_test)

**Notes:**

* The input images are flattened from (28, 28) to (784,).
* You can adjust the number of epochs or batch size if needed.
* Try increasing latent\_dim to see better reconstructions (but lower compression).
* This implementation uses sigmoid in the output and binary cross-entropy loss, as MNIST is grayscale and normalized between 0 and 1.

2)Let me know if you’d like to visualize the encoded (latent) space or convert it into a convolutional autoencoder.

Here's the full code to implement a **Denoising Autoencoder** using the MNIST dataset and compare it with the **Basic Autoencoder** from Q1.

1. Add Gaussian noise to the input images.
2. Train the autoencoder with noisy images as input but clean images as targets.
3. Visualize and compare.
4. Provide a real-world use case.

import numpy as np

import matplotlib.pyplot as plt

from tensorflow.keras.datasets import mnist

from tensorflow.keras.models import Model

from tensorflow.keras.layers import Input, Dense

from tensorflow.keras.optimizers import Adam

from tensorflow.keras.losses import BinaryCrossentropy

# Load and normalize MNIST

(x\_train, \_), (x\_test, \_) = mnist.load\_data()

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

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

x\_train = x\_train.reshape((len(x\_train), -1)) # (60000, 784)

x\_test = x\_test.reshape((len(x\_test), -1)) # (10000, 784)

# Add Gaussian noise

def add\_noise(data, mean=0.0, std=0.5):

noise = np.random.normal(mean, std, data.shape)

noisy\_data = data + noise

noisy\_data = np.clip(noisy\_data, 0., 1.)

return noisy\_data

x\_train\_noisy = add\_noise(x\_train)

x\_test\_noisy = add\_noise(x\_test)

# Build autoencoder

def build\_autoencoder(latent\_dim):

input\_img = Input(shape=(784,))

encoded = Dense(latent\_dim, activation='relu')(input\_img)

decoded = Dense(784, activation='sigmoid')(encoded)

return Model(input\_img, decoded)

# Train function

def train\_denoising\_autoencoder(latent\_dim=32, epochs=20):

autoencoder = build\_autoencoder(latent\_dim)

autoencoder.compile(optimizer=Adam(), loss=BinaryCrossentropy())

autoencoder.fit(x\_train\_noisy, x\_train, # input: noisy, output: clean

epochs=epochs,

batch\_size=256,

shuffle=True,

validation\_data=(x\_test\_noisy, x\_test),

verbose=2)

return autoencoder

# Visualization

def plot\_denoising(autoencoder, x\_test\_noisy, x\_test, num\_images=10):

decoded\_imgs = autoencoder.predict(x\_test\_noisy)

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

for i in range(num\_images):

# Noisy input

ax = plt.subplot(3, num\_images, i + 1)

plt.imshow(x\_test\_noisy[i].reshape(28, 28), cmap="gray")

plt.title("Noisy")

plt.axis("off")

# Clean ground truth

ax = plt.subplot(3, num\_images, i + 1 + num\_images)

plt.imshow(x\_test[i].reshape(28, 28), cmap="gray")

plt.title("Original")

plt.axis("off")

# Reconstructed

ax = plt.subplot(3, num\_images, i + 1 + 2 \* num\_images)

plt.imshow(decoded\_imgs[i].reshape(28, 28), cmap="gray")

plt.title("Denoised")

plt.axis("off")

plt.show()

# Train and visualize

autoencoder = train\_denoising\_autoencoder(latent\_dim=32)

plot\_denoising(autoencoder, x\_test\_noisy, x\_test)

3) Here's a full implementation of a **character-level text generator** using an **LSTM-based RNN** in TensorFlow/Keras.

We'll:

* Load a sample text (you can replace it with any book or poem),
* Preprocess it into sequences,
* Train an LSTM to predict the next character,
* Generate new text using **temperature scaling** for creativity/randomness.

import numpy as np

import tensorflow as tf

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import LSTM, Dense

import matplotlib.pyplot as plt

# 1. Load text dataset

path = tf.keras.utils.get\_file('shakespeare.txt',

'https://storage.googleapis.com/download.tensorflow.org/data/shakespeare.txt')

with open(path, 'r', encoding='utf-8') as f:

text = f.read().lower()

print(f"Corpus length: {len(text)}")

# 2. Preprocess: Create char -> index mappings

chars = sorted(set(text))

char2idx = {c: i for i, c in enumerate(chars)}

idx2char = np.array(chars)

vocab\_size = len(chars)

print(f"Unique characters: {vocab\_size}")

# Convert all text to integer indices

text\_as\_int = np.array([char2idx[c] for c in text])

# Create sequences of input and target characters

seq\_length = 100

step = 1

input\_seqs = []

target\_chars = []

for i in range(0, len(text\_as\_int) - seq\_length, step):

input\_seqs.append(text\_as\_int[i:i + seq\_length])

target\_chars.append(text\_as\_int[i + seq\_length])

x = np.array(input\_seqs)

y = np.array(target\_chars)

# One-hot encode y (optional, Dense layer with softmax will handle this)

y = tf.keras.utils.to\_categorical(y, num\_classes=vocab\_size)

# 3. Build the model

model = Sequential([

LSTM(128, input\_shape=(seq\_length, vocab\_size)),

Dense(vocab\_size, activation='softmax')

])

model.compile(optimizer='adam', loss='categorical\_crossentropy')

model.summary()

# 4. Prepare inputs (one-hot encode x)

x\_onehot = tf.keras.utils.to\_categorical(x, num\_classes=vocab\_size)

# Train

model.fit(x\_onehot, y, batch\_size=128, epochs=10)

# 5. Text generation with temperature scaling

def sample(preds, temperature=1.0):

preds = np.asarray(preds).astype('float64')

preds = np.log(preds + 1e-8) / temperature # temperature scaling

exp\_preds = np.exp(preds)

preds = exp\_preds / np.sum(exp\_preds)

return np.random.choice(len(preds), p=preds)

def generate\_text(model, seed\_text, length=300, temperature=1.0):

generated = seed\_text

input\_seq = [char2idx[c] for c in seed\_text.lower()]

for \_ in range(length):

input\_arr = tf.keras.utils.to\_categorical([input\_seq[-seq\_length:]], num\_classes=vocab\_size)

preds = model.predict(input\_arr, verbose=0)[0]

next\_index = sample(preds, temperature)

next\_char = idx2char[next\_index]

generated += next\_char

input\_seq.append(next\_index)

return generated

# Try generating text with different temperatures

seed = "To be, or not to be, that is the question:"

print("\n-- Temperature = 0.2 --")

print(generate\_text(model, seed, temperature=0.2))

print("\n-- Temperature = 1.0 --")

print(generate\_text(model, seed, temperature=1.0))

print("\n-- Temperature = 1.5 --")

print(generate\_text(model, seed, temperature=1.5))

**🔥 Temperature Scaling Explained:**

* **Low temperature (~0.2)** → Model chooses the most likely characters → conservative, repetitive text.
* **High temperature (~1.2–1.5)** → More randomness, creative/weird outputs.

4) Here's the full implementation for **Sentiment Classification using LSTM on the IMDB dataset** with evaluation using a **confusion matrix and classification report**.

import numpy as np

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

from sklearn.metrics import confusion\_matrix, classification\_report

import matplotlib.pyplot as plt

import seaborn as sns

# 1. Load IMDB dataset (only top 10,000 words)

vocab\_size = 10000

maxlen = 200

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

# 2. Preprocess: Pad sequences

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

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

# 3. Build LSTM-based model

model = Sequential([

Embedding(vocab\_size, 128, input\_length=maxlen),

LSTM(64, dropout=0.2, recurrent\_dropout=0.2),

Dense(1, activation='sigmoid')

])

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

model.summary()

# Train the model

model.fit(x\_train, y\_train,

batch\_size=128,

epochs=3,

validation\_split=0.2,

verbose=2)

# 4. Evaluate model

y\_pred\_probs = model.predict(x\_test)

y\_pred = (y\_pred\_probs > 0.5).astype(int)

# Confusion matrix and classification report

cm = confusion\_matrix(y\_test, y\_pred)

cr = classification\_report(y\_test, y\_pred, digits=4)

# Plot confusion matrix

plt.figure(figsize=(6, 4))

sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=["Neg", "Pos"], yticklabels=["Neg", "Pos"])

plt.xlabel("Predicted")

plt.ylabel("Actual")

plt.title("Confusion Matrix")

plt.show()

# Print classification report

print("Classification Report:\n", cr)