

De La Salle University – Manila

Gokongwei College of Engineering

Department of Electronics and Computer Engineering

CpE Elective 3 Laboratory

LBYCPC4

Summative Laboratory Report

Prepared by:

Banal, Kenan A.

LBYCPC4 – EQ3

# TABLE OF CONTENTS

Table of Contents

[TABLE OF CONTENTS 2](#_Toc212710335)

[I. INTRODUCTION <Replace the guide text with your own introduction; Approx. 1-2 pages> 4](#_Toc212710336)

[II. METHODOLOGY, RESULTS, AND ANALYSIS 4](#_Toc212710337)

[Activity 1: Autoencoders 4](#_Toc212710338)

[1.1. Objectives 4](#_Toc212710339)

[1.2. Experimental Procedure 4](#_Toc212710340)

[1.3. Results and Analysis 5](#_Toc212710341)

[1.4. Answer to Guide Questions, Insights, and Reflections 24](#_Toc212710342)

[Activity 2: Variational Autoencoders and Generative Adversarial Networks 25](#_Toc212710343)

[2.1. Objectives 25](#_Toc212710344)

[2.2. Experimental Procedure 25](#_Toc212710345)

[2.3. Results and Analysis 25](#_Toc212710346)

[2.4. Answer to Guide Questions, Insights, and Reflections 44](#_Toc212710347)

[Activity 3: Text Generation Models 45](#_Toc212710348)

[1.1. Objectives 45](#_Toc212710349)

[1.2. Experimental Procedure 45](#_Toc212710350)

[1.3. Results and Analysis 45](#_Toc212710351)

[1.4. Answer to Guide Questions, Insights, and Reflections 68](#_Toc212710352)

[Activity 4: Diffusion-based Generative Models 69](#_Toc212710353)

[1.1. Objectives 69](#_Toc212710354)

[1.2. Experimental Procedure 69](#_Toc212710355)

[1.3. Results and Analysis 69](#_Toc212710356)

[1.4. Answer to Guide Questions, Insights, and Reflections 85](#_Toc212710357)

[Activity 5: Familiarization with Single Board Computer (SBC) 86](#_Toc212710358)

[1.1. Objectives 86](#_Toc212710359)

[1.2. Experimental Procedure 86](#_Toc212710360)

[1.3. Results and Analysis 86](#_Toc212710361)

[1.4. Answer to Guide Questions, Insights, and Reflections 96](#_Toc212710362)

[<Add more activity sections as needed> 96](#_Toc212710363)

[Activity 6: <Title> 97](#_Toc212710364)

[1.1. Objectives 97](#_Toc212710365)

[1.2. Experimental Procedure 97](#_Toc212710366)

[1.3. Results and Analysis 97](#_Toc212710367)

[1.4. Answer to Guide Questions, Insights, and Reflections 97](#_Toc212710368)

[<Add more activity sections as needed> 98](#_Toc212710369)

[II. CONCLUSION AND OVERALL UNDERSTANDING <Replace the guide text with your own conclusions. Approx. 1-2 pages> 98](#_Toc212710370)

[III. REFERENCES <Use APA Citation Style> 99](#_Toc212710371)

# INTRODUCTION <Replace the guide text with your own introduction; Approx. 1-2 pages>

<This report serves as a summative assessment of your understanding and application of the concepts and theory in the design and development of generative deep learning models and the deployment of deep learning models on a single-board computer. It should demonstrate your ability to analyze problems, design and implement working models, and critically reflect on your learning.

In this section, write a complete introduction with the following guidelines:

* State the purpose of this report – e.g., to demonstrate your understanding and application of concepts learned in all laboratory activities.
* Discuss all the concepts and theories you have learned. Ideally, there should be one paragraph that discusses the concepts covered for each laboratory activity. Provide in-text citations that support the statements you have written in relation to the concepts or theories learned.
* In-text citations should refer to reputable sources and should have a corresponding entry in the references section.

>

# METHODOLOGY, RESULTS, AND ANALYSIS

## Activity 1: Autoencoders

### Objectives

1. Understand the working principle and architecture of autoencoders.
2. Build and train an autoencoder using deep learning framework.
3. Visualize the output of an autoencoder
4. Assess the performance of the implemented autoencoder

### Experimental Procedure

**A. Dimensionality Reduction with Autoencoders**

1. Download the Wine dataset from the UCI ML Repository and load it using Scikit-Learn.
2. Normalize the dataset features using MinMaxScaler.
3. Split the dataset into training (80%) and testing (20%) sets.
4. Build an autoencoder model using Keras Sequential API based on the given architecture.
5. Compile the model with Adam optimizer and mean squared error loss.
6. Train the autoencoder with a batch size of 10, using early stopping until validation loss ≤ 0.02.
7. Plot the training and validation loss per epoch with proper labels and title.
8. Extract the encoder subnetwork from the trained autoencoder.
9. Obtain the encoded representation of both training and testing samples.
10. Generate a 3D scatter plot of the encoder output and analyze the clusters.
11. Perform Principal Component Analysis (PCA) with 3 components and plot in 3D. Compare the PCA results with the encoder output.
12. Build another autoencoder with only 2 hidden nodes in the middle layer.
13. Repeat steps 5–9 for the 2-node autoencoder, this time generating 2D plots.
14. Compare the validation loss and clustering results with the first autoencoder.

**B. Building a Denoising Convolutional Autoencoder**

1. Download and load the Fashion MNIST dataset. Normalize pixel values to [0,1].
2. Display at least ten sample training images in a single row.
3. Generate noisy versions of training and testing images by adding Gaussian noise.
4. Display at least ten noisy images from the training set.
5. Build a convolutional autoencoder model using Keras based on the given architecture.
6. Compile the model with an appropriate optimizer and loss function.
7. Train the model with a batch size of 100, using early stopping until validation loss ≤ 0.01.
8. Plot the training and validation loss per epoch with labels and title.
9. Obtain the denoised outputs for at least ten noisy test images.
10. Display the noisy inputs (first row) and corresponding denoised outputs (second row).
11. Measure the reconstruction error between original clean images and denoised outputs using a suitable error metric.
12. Compute and report the average reconstruction error across all test images.
13. Collect at least five external clothing images (normalized grayscale, same size as dataset).
14. Apply noise to these external images in the same manner as step 3.
15. Use the trained denoising autoencoder to predict denoised versions.
16. Display the noisy inputs and their denoised outputs in two rows, then record observations.

### Results and Analysis

|  |
| --- |
| # Load the Wine dataset as a Numpy array  # Perform feature normalization using MinMaxScaler  # Perform train-test split with 20% test data  # Save the training data to variable X\_train  # Save the testing data to variable X\_test  # Save the target to variable y  ### --YOUR CODE HERE-- ###  wine = load\_wine()  X = wine.data  y = wine.target  scaler = MinMaxScaler()  X\_normalized = scaler.fit\_transform(X)  X\_train, X\_test, y\_train, y\_test = train\_test\_split(X\_normalized,                                    y, test\_size=0.2, random\_state=42)  y = y  print("X\_train shape:", X\_train.shape)  print("X\_test shape:", X\_test.shape)  print("y shape:", y.shape) |

Figure 1.1. Loading the Wine Dataset and Splitting the Dataset

As seen on Figure 1.1, the wine dataset is imported and the X is set as the data and y is set as the target. Furthermore, normalization is used using the MinMaxScaler which was imported. This was used to normalize the wide X data. Likewise, the normalized data is split to 80-20 where 20% is the test size while the 80% is the train size. Noted that the X train shape is 142 by 13 and the X test shape is 36 by 13 while the y shape is 178.

|  |
| --- |
| # Create the autoencoder model based from the illustrated model plot  # Save the model as autoencoder\_ann variable  ### --YOUR CODE HERE-- ###  input\_dim = X\_train.shape[1]  autoencoder\_ann = Sequential([Input(shape=(input\_dim,)),                                Dense(3, activation="sigmoid"),                                Dense(input\_dim, activation="sigmoid")]) |

Figure 1.2 Creating the Autoencoder

A diagram of a code

AI-generated content may be incorrect.

Figure 1.2.1 Autoencoder Architecture

The coded autoencoder architecture of Figure 1.2.1 is show on Figure 1.2. The encoder’s architecture contains 2 activation sigmoid with an input shape of 13 which is the input dim or the X train other shape.

|  |
| --- |
| # Configure the network for training using the compile method  # Set the optimizer to Adam and determine the appropriate loss function  ### --YOUR CODE HERE-- ###  autoencoder\_ann.compile(optimizer='adam', loss='mse')  # Train the model using X\_train. Use validation with X\_test  # Set the batch size to 10 and the verbosity to 2.  # Ensure the validation loss to be no greater than 0.02  # Use early stopping to determine the appropriate training epochs  # Assign the output to hist\_autoencoder\_ann variable  ### --YOUR CODE HERE-- ###  from keras.callbacks import EarlyStopping  early\_stopping = EarlyStopping(monitor='val\_loss', mode='min', baseline=0.02)  hist\_autoencoder\_ann = autoencoder\_ann.fit(      X\_train, X\_train,      validation\_data=(X\_test, X\_test),      epochs=1000,      batch\_size=10,      verbose=2,      callbacks=[early\_stopping]  )  eval\_base = autoencoder\_ann.evaluate(X\_test, X\_test, verbose=0) |

Figure 1.3. Configuring the Autoencoder and Making of Early Stopper

As seen on Figure 1.3, adam optimizer with an MSE loss is set for the training network. Furthermore, the validation loss is set to be no more than 0.02 in the early stopper and the batch size of 10 with verbose of 2.

|  |
| --- |
| # Extract the losses during training and validation  losses = hist\_autoencoder\_ann.history["loss"]  validation\_losses = hist\_autoencoder\_ann.history["val\_loss"]  epochs = range(1, len(losses) + 1)  # Plot the history of training and validation losses  plt.figure(figsize=(10, 5), constrained\_layout=True)  plt.subplot(1, 2, 1)  ### --YOUR CODE HERE-- ###  plt.plot(epochs, losses)  plt.title('Training Loss per Epoch')  plt.xlabel('Epoch')  plt.ylabel('Loss')  plt.grid(True)  plt.subplot(1, 2, 2)  plt.plot(epochs, validation\_losses)  plt.title('Validation Loss per Epoch')  plt.xlabel('Epoch')  plt.ylabel('Loss')  plt.grid(True)  plt.show() |

Figure 1.4. Plotting the Training and Validation loss

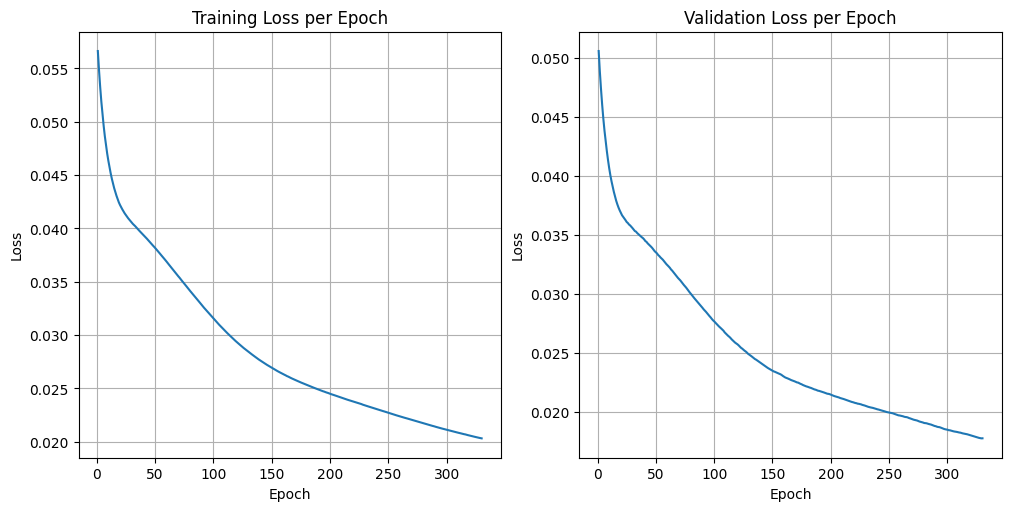


Figure 1.4.1. Training and Validation Loss per Epoch

As seen on Figure 1.4.1, the validation loss is no more than 0.020 above the 250 epoch. Furthermore, the code to plot the training and validation loss is shown in Figure 1.4.

|  |
| --- |
| # Extract the encoder subnetwork  encoder\_ann = Sequential(autoencoder\_ann.layers[:1])  # Obtain the output of the encoder for each dataset sample  # Save the results to X\_encoded variable  # The results should include both training and testing data  ### --YOUR CODE HERE-- ###  X\_encoded = encoder\_ann.predict(X\_normalized) |

Figure 1.5. Getting of Encoder Prediction

Figure 1.5 shows the extraction of the encoder’s output and the getting the normalized encoder’s output.

|  |
| --- |
| # Import PCA from Scikit-Learn  from sklearn.decomposition import PCA  X\_PCA = PCA(n\_components=3).fit\_transform(np.vstack((X\_train, X\_test)))  # Create a 3D plot of the PCA components for each dataset point  # Put appropriate title and axis labels  ### --YOUR CODE HERE-- ###  fig = plt.figure(1, figsize=(8, 6))  ax = fig.add\_subplot(111, projection="3d", elev=45, azim=-45)  ax.scatter(      X\_PCA[:, 0],      X\_PCA[:, 1],      X\_PCA[:, 2],      c=y,      s=20  )  # Set titles and axis labels  ax.set\_title("PCA output")  ax.set\_xlabel("Component 1")  ax.set\_ylabel("Component 2")  ax.set\_zlabel("Component 3")  plt.show() |

Figure 1.6. 3D Plotting Code the PCA

A graph of a function

AI-generated content may be incorrect.

Figure 1.6.1 3D Plot of Given PCA

A diagram of a computer generated graph

AI-generated content may be incorrect.

Figure 1.6.2 3D Plot of Our PCA

Figure 1.6.2 as seen above shows the PCA or the Principal Component Analysis using the X train and X test data. The respective code to plot Figure 1.6.2 is shown in Figure 1.6. For the observations of Figure 1.6.1 and Figure 1.6.2, Figure 1.6.1 is clearly more separated than Figure 1.6.2 for all the targets in the clusters.

|  |
| --- |
| # Define another autoencoder with 2 nodes at hidden layer  ### --YOUR CODE HERE-- ###  autoencoder\_2\_nodes = Sequential([      Input(shape=(13,)),      Dense(2, activation='sigmoid'),      Dense(13, activation='sigmoid')  ])  autoencoder\_2\_nodes.summary()  # Configure and train the network  ### --YOUR CODE HERE-- ###  autoencoder\_2\_nodes.compile(      optimizer='adam',      loss='mse'  )  es = EarlyStopping(monitor='val\_loss', mode='min', baseline=0.02)  history\_2\_nodes = autoencoder\_2\_nodes.fit(      X\_train, X\_train,      epochs=1000,      validation\_data=(X\_test, X\_test),      verbose=2,      batch\_size=10,      callbacks=[es]  )  # Plot the history of training and validation losses  ### --YOUR CODE HERE-- ###  train\_losses = history\_2\_nodes.history["loss"]  val\_losses = history\_2\_nodes.history["val\_loss"]  epochs = range(1, len(train\_losses) + 1)  plt.figure(figsize=(10, 5), constrained\_layout=True)  plt.subplot(1, 2, 1)  plt.plot(epochs, train\_losses)  plt.xlabel("Epochs")  plt.ylabel("Training Loss")  plt.title("Training Loss per Epoch")  plt.subplot(1, 2, 2)  plt.plot(epochs, val\_losses)  plt.xlabel("Epochs")  plt.ylabel("Validation Loss")  plt.title("Validation Loss per Epoch")  plt.show()  # Obtain the output of the encoder for each dataset sample  ### --YOUR CODE HERE-- ###  encoder\_2\_nodes = Sequential(autoencoder\_2\_nodes.layers[:1])  X\_encoded\_2\_nodes = encoder\_2\_nodes.predict(X\_normalized)  plt.figure(figsize=(8, 6))  plt.scatter(      X\_encoded\_2\_nodes[:, 0],      X\_encoded\_2\_nodes[:, 1],      c=y,      s=20  )  plt.title("Encoder Output")  plt.xlabel("Component 1")  plt.ylabel("Component 2")  plt.show() |

Figure 1.7 Making Another Autoencoder but with 2D Plots

A graph of loss and loss

AI-generated content may be incorrect.

Figure 1.7.1 Training and Validation Loss per Epoch

A graph of colored dots

AI-generated content may be incorrect.

Figure 1.7.2 2D Plots of the Encoder

As seen on Figure 1.7.1 and Figure 1.7.2, the training and validation loss plot per epoch shown and the 2D encoder output. Likewise, the code for these two figures is shown in Figure 1.7. The code inside Figure 1.7 contains the defining of autoencoder together with the training and configuration of the autoencoder together with the early stopping and its parameters for training it and plotting both the training and validation loss together with the 2D Encoder plot.

|  |
| --- |
| # Import functions and classes  from keras.datasets import fashion\_mnist  from keras.utils import to\_categorical  # Load the Fashion MNIST dataset  # Rescale the pixel values to be between 0 and 1  # Save the training images and labels to x\_train and y\_train variables  # Save the testing images and labels to x\_test and y\_test variables  # Plot ten (10) images from training set in a single row  ### --YOUR CODE HERE-- ###  (x\_train, y\_train), (x\_test, y\_test) = fashion\_mnist.load\_data()  x\_train = x\_train.astype('float32') / 255.0  x\_test = x\_test.astype('float32') / 255.0  class\_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',                 'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']  fig, axes = plt.subplots(1, 10, figsize=(15, 2),                           subplot\_kw={'xticks':[], 'yticks':[]})  for i, ax in enumerate(axes.flat):      ax.imshow(x\_train[i], cmap='binary')      ax.set\_title(f'{class\_names[y\_train[i]]}', fontsize=10)  plt.suptitle('Original Fashion MNIST Images')  plt.tight\_layout()  plt.show() |

Figure 1.8. Loading and Splitting the Fashion MNIST dataset

A black and white picture of a chair

AI-generated content may be incorrect.

Figure 1.8.1 Loading and Splitting the Fashion MNIST dataset

The code shown on Figure 1.8 shows that loading of the Fashion MNIST dataset while assigning the training and testing images and labels together with the plotting of the 10 images from the dataset. The sample images of the Fashion MNIST dataset which was plotted can be seen on the Figure 1.8.1.

|  |
| --- |
| # Apply noise to the image  x\_train\_noisy = x\_train + 0.2\*np.random.normal(loc=0, scale=1,                                                 size=x\_train.shape)  x\_test\_noisy = x\_test + 0.2\*np.random.normal(loc=0, scale=1,                                               size=x\_test.shape)  x\_train\_noisy = np.clip(x\_train\_noisy, 0, 1)  x\_test\_noisy = np.clip(x\_test\_noisy, 0, 1)  fig, axes = plt.subplots(1, 10, figsize=(10, 10),                           subplot\_kw={'xticks':[], 'yticks':[]},                           gridspec\_kw=dict(hspace=0.1, wspace=0.1))  for i, ax in enumerate(axes.flat):    ax.imshow(x\_train\_noisy[i], cmap='binary', interpolation='none') |

Figure 1.9. Adding Noise to the Images and Plotting it

Figure 1.9 shows the code for adding noise to the training and testing image dataset which was also plotted as seen on the Figure 1.9.1.

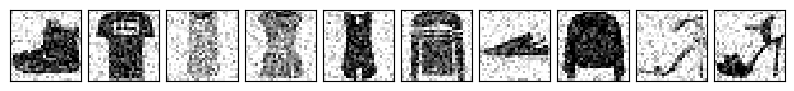


Figure 1.9.1 Noise Image Dataset Plot

|  |
| --- |
| # Import functions and classes  from keras.layers import Conv2D, MaxPooling2D, UpSampling2D  # Create the autoencoder model based from the illustrated model plot  # Save the model as autoencoder\_cnn variable  ### --YOUR CODE HERE-- ###  autoencoder\_cnn = Sequential([      Input(shape=(28, 28, 1)),      Conv2D(32, (3, 3), activation='relu', padding='same', name='conv2d\_1'),      MaxPooling2D((2, 2), padding='same', name='max\_pooling2d'),      Conv2D(32, (3, 3), activation='relu', padding='same', name='conv2d\_2'),      MaxPooling2D((2, 2), padding='same', name='code'),      Conv2D(32, (3, 3), activation='relu', padding='same', name='conv2d\_3'),      UpSampling2D((2, 2), name='up\_sampling2d'),      Conv2D(32, (3, 3), activation='relu', padding='same', name='conv2d\_4'),      UpSampling2D((2, 2), name='up\_sampling2d\_1'),      Conv2D(1, (3, 3), activation='sigmoid', padding='same', name='decode')  ])  autoencoder\_cnn.summary() |

Figure 1.10 Making a Denoising Autoencoder

A black and white diagram

AI-generated content may be incorrect.

Figure 1.10.1 Denoising Autoencoder Architecture

The coding of the denoising autoencoder architecture which is shown on Figure 1.10.1 is seen in Figure 1.10. The architecture of the denoising autoencoder contains four 2D convolution with activation function of relu, two up 2D sampling, and two 2D max pooling.

|  |
| --- |
| # Configure the network for training  ### --YOUR CODE HERE-- ###  autoencoder\_cnn.compile(optimizer='adam', loss='mse')  # Train the model. Perform validation as well  # Set the batch size to 100 and the verbosity to 2  # Ensure the validation loss to be no greater than 0.01  # Use early stopping to determine the appropriate training epochs  # Assign the output to hist\_autoencoder\_cnn variable  ### --YOUR CODE HERE-- ###  early\_stopping = EarlyStopping(      monitor='val\_loss',      baseline = 0.01,      mode = 'min'  )  hist\_autoencoder\_cnn = autoencoder\_cnn.fit(      x\_train\_noisy, x\_train\_noisy,      validation\_data=(x\_test\_noisy, x\_test\_noisy),      epochs=900,      batch\_size=100,      verbose=2,      callbacks=[early\_stopping]  ) |

Figure 1.11 Configuring the Autoencoder Parameters with Early Stopper

Likewise, Figure 1.11 shows the configuration of the denoising autoencoder with an adam optimizer and MSE loss. For the training, an early stopper is used with a baseline of 0.01 for the validation loss and the batch size is 100 and verbosity of 2.

|  |
| --- |
| # Extract the losses during training and validation  losses = hist\_autoencoder\_cnn.history["loss"]  validation\_losses = hist\_autoencoder\_cnn.history["val\_loss"]  epochs = range(1, len(losses) + 1)  # Plot the history of training and validation losses  plt.figure(figsize=(10, 5), constrained\_layout=True)  plt.subplot(1, 2, 1)  ### --YOUR CODE HERE-- ###  plt.plot(epochs, losses)  plt.xlabel("Epochs")  plt.ylabel("Training Loss")  plt.title("Training Loss per Epoch")  plt.subplot(1, 2, 2)  plt.plot(epochs, validation\_losses)  plt.xlabel("Epochs")  plt.ylabel("Validation Loss")  plt.title("Validation Loss per Epoch")  plt.show() |

Figure 1.12 Plotting the Training and Validation Loss of Autoencoder

A graph of a loss

AI-generated content may be incorrect.

Figure 1.12.1 Training and Validation Loss of Autoencoder

For the plotting of the training and validation loss of the denoising autoencoder, the code is shown on Figure 1.12 and the plot curve is shown on Figure 1.12.1 where the validation loss is not able to reach below 0.01. This might be due to using adam optimizer for training the model.

|  |
| --- |
| # Obtain model output for ten (10) input noisy images  # Display both the input noisy images and the output denoised image in two rows  # The first row contains the input noisy images  # The second row contains the output denoised images  ### --YOUR CODE HERE-- ###  denoised\_images = autoencoder\_cnn.predict(x\_test\_noisy[:10])  fig, axes = plt.subplots(2, 10, figsize=(15, 4))  for i in range(10):    axes[0,i].imshow((x\_test\_noisy[i]).reshape(28,28), cmap='gray')    axes[0,i].set\_title("Noisy Img")    axes[1,i].imshow((denoised\_images[i]).reshape(28,28), cmap='gray')    axes[1,i].set\_title("Denoised Img")  plt.suptitle("Image Denoising")  plt.show() |

Figure 1.13. Displaying the Noisy and Denoised Images

A collage of images of a person's body

AI-generated content may be incorrect.

Figure 1.13.1. Image Denoising Dataset

Figure 1.13 shows the code for displaying the denoise and noisy images in the dataset. Furthermore, the plot with the images can be seen on Figure 1.13.1 where the denoising is able to remove higher noticeable noise while keeping the clothes distinguishable. However, the denoised image still has some noticeable noise at the background.

|  |
| --- |
| ### --YOUR CODE HERE-- ###  from sklearn.metrics import mean\_absolute\_error  decoded\_images = autoencoder\_cnn.predict(x\_test\_noisy)  errors = []  for i in range(len(x\_test)):      mae = mean\_absolute\_error(          x\_test[i].reshape(-1),          decoded\_images[i].reshape(-1)      )      errors.append(mae)  errors = np.array(errors)  avg\_error = np.mean(errors)  print("Average MAE:", avg\_error)  plt.hist(errors, bins=50)  plt.title("Distribution of MAE")  plt.xlabel("MAE")  plt.ylabel("Number of Images")  plt.show() |

Figure 1.14. Displaying the Average MAE and its Image Distribution

A graph of a distribution of a number

AI-generated content may be incorrect.

Figure 1.14.1 MAE vs Number of Images Distribution

The code for doing the MAE distribution and the plot of the images to MAE distribution is shown at Figure 1.14 and Figure 1.14.1. Likewise, the average MAE is shown to be 0.09466809390932321.

|  |
| --- |
| ### --YOUR CODE HERE-- ###  import requests  from PIL import Image  from io import BytesIO  clothing\_urls = [    "https://storage.googleapis.com/kagglesdsdata/datasets/929774/1572891/images\_original/00805d0e-7fe5-4251-b577-86065e4f6587.jpg?X-Goog-Algorithm=GOOG4-RSA-SHA256&X-Goog-Credential=databundle-worker-v2%40kaggle-161607.iam.gserviceaccount.com%2F20250914%2Fauto%2Fstorage%2Fgoog4\_request&X-Goog-Date=20250914T111709Z&X-Goog-Expires=345600&X-Goog-SignedHeaders=host&X-Goog-Signature=",    "https://storage.googleapis.com/kagglesdsdata/datasets/139630/329006/fashion-dataset/images/10020.jpg?X-Goog-Algorithm=GOOG4-RSA-SHA256&X-Goog-Credential=databundle-worker-v2%40kaggle-161607.iam.gserviceaccount.com%2F20250914%2Fauto%2Fstorage%2Fgoog4\_request&X-Goog-Date=20250914T102554Z&X-Goog-Expires=345600&X-Goog-SignedHeaders=host&X-Goog-Signature=",    "https://storage.googleapis.com/kagglesdsdata/datasets/139630/329006/fashion-dataset/images/10029.jpg?X-Goog-Algorithm=GOOG4-RSA-SHA256&X-Goog-Credential=databundle-worker-v2%40kaggle-161607.iam.gserviceaccount.com%2F20250914%2Fauto%2Fstorage%2Fgoog4\_request&X-Goog-Date=20250914T102554Z&X-Goog-Expires=345600&X-Goog-SignedHeaders=host&X-Goog-Signature=",    "https://storage.googleapis.com/kagglesdsdata/datasets/139630/329006/fashion-dataset/images/10013.jpg?X-Goog-Algorithm=GOOG4-RSA-SHA256&X-Goog-Credential=databundle-worker-v2%40kaggle-161607.iam.gserviceaccount.com%2F20250914%2Fauto%2Fstorage%2Fgoog4\_request&X-Goog-Date=20250914T102554Z&X-Goog-Expires=345600&X-Goog-SignedHeaders=host&X-Goog-Signature=",    "https://storage.googleapis.com/kagglesdsdata/datasets/929774/1572891/images\_original/002eb5b8-6541-42a3-9596-0d94f7b866ae.jpg?X-Goog-Algorithm=GOOG4-RSA-SHA256&X-Goog-Credential=databundle-worker-v2%40kaggle-161607.iam.gserviceaccount.com%2F20250914%2Fauto%2Fstorage%2Fgoog4\_request&X-Goog-Date=20250914T111709Z&X-Goog-Expires=345600&X-Goog-SignedHeaders=host&X-Goog-Signature=",  ]  x\_external = []  for url in clothing\_urls:    response = requests.get(url, timeout=10)    img = Image.open(BytesIO(response.content))    img = img.convert("L")    img = img.resize((28, 28))    img\_array = np.array(img) / 255.0    x\_external.append(img\_array.reshape(28, 28))  x\_external = np.array(x\_external)  x\_external\_noisy = x\_external + 0.2 \* np.random.normal(      loc=0, scale=1, size=x\_external.shape  )  x\_external\_noisy = np.clip(x\_external\_noisy, 0, 1)  decoded\_external = autoencoder\_cnn.predict(x\_external\_noisy)  fig, axes = plt.subplots(2, 5, figsize=(12, 5))  for i in range(5):      axes[0, i].imshow(x\_external\_noisy[i].reshape(28, 28), cmap="binary")      axes[0, i].set\_title("Noisy")      axes[1, i].imshow(decoded\_external[i].reshape(28, 28), cmap="binary")      axes[1, i].set\_title("Denoised")  plt.suptitle("Noisy and Denoised Images")  plt.show() |

Figure 1.15. Displaying the 5 Tested Noisy to Denoised Images

A group of images of a person

AI-generated content may be incorrect.

Figure 1.15.1 Five Noisy to Denoised Images

For the last activity, five noisy to denoised images were tested where the images are taken from the internet and not from the same dataset. The code of this can be seen on Figure 1.15 and the plot with the images can be seen on Figure 1.15.1. For the observations, the first image was not able to denoise because the image has a light-colored background which shows the weakness to the grayscale to the image. Furthermore, the second and third image was successfully able to denoise the image however it became blurry with some noise in the background. Lastly, the fourth and fifth image were not successful in deblurring while making the clothes distinguishable.

### Answer to Guide Questions, Insights, and Reflections

A.1. Is the final validation loss better than the previous model?

The data shown on Figure 1.7.1 shows that the final validation loss is 0.0165 while the other previous model in Figure 1.4 has a final validation loss of 0.0177. This implies that the second model is better than the initially trained autoencoder model.

A.2. Can the model clearly separate the targets into distinguishable clusters?

The data on Figure 1.6.2 and Figure 1.6.1 shows that Figure 1.6.2 has more distinguishable clusters in comparison to Figure 1.6.1.

For my insight and reflection, I learned a lot about autoencoders and the different parts which are the original data, encoder, code, decoder and reconstructed data. This means that autoencoders are becoming less like a black box concept to me because parts such as the hidden layer are being explained how to be fine tuned for the input and output layer.

## Activity 2: Variational Autoencoders and Generative Adversarial Networks

### Objectives

* Understand the working principle and architecture of variational autoencoders and generative adversarial networks
* Build and train a variational autoencoder using deep learning framework
* Build and train a generative adversarial network using deep learning framework
* Visualize the output of the generative networks
* Assess the performance of the implemented generative networks

### Experimental Procedure

<…>

### Results and Analysis

|  |
| --- |
| # Import libraries  from torchvision.datasets import EMNIST  # Get the EMNIST dataset. Use the balanced split  # Normalize the image pixel values to within the range [0, 1]  # Save the training data to variable X\_train as a Numpy array  # Save the testing data to variable X\_test as a Numpy array  # Save the targets to variable y as a Numpy array  # Note that the images provided are inverted horizontally and  # rotated 90 anti-clockwise. It must be oriented properly  ### --YOUR CODE HERE-- ###  from torchvision import transforms  emnist\_train = EMNIST("./data", split="balanced", train=True, download=True,  transform=transforms.ToTensor()  )  emnist\_test = EMNIST("./data", split="balanced", train=False, download=True,  transform=transforms.ToTensor()  )  X\_train = emnist\_train.data.numpy().astype("float32")/255  X\_train = np.rot90(X\_train, k=2)  X\_train = np.flipud(X\_train)  #X\_train = np.fliplr(X\_train)  X\_train = np.expand\_dims(X\_train, axis=-1)  X\_test = emnist\_test.data.numpy().astype("float32")/255  X\_test = np.rot90(X\_test, k=2)  X\_test = np.flipud(X\_test)  #X\_test = np.fliplr(X\_test)  X\_test = np.expand\_dims(X\_test, axis=-1)  y = emnist\_train.targets.numpy()  # Display the first thirty (30) images from the training split  # Place ten (10) images per row  # Each image should have a label underneath  ### --YOUR CODE HERE-- ###  figure, axs = plt.subplots(3, 10, figsize=(20, 6))  print(X\_train.shape)  for i, j in enumerate(axs.flatten()):  j.imshow(X\_train[i])  j.set\_title(y[i])  plt.tight\_layout()  plt.show() |

Figure 2.1. Code for Loading the EMNIST Dataset and Plotting it

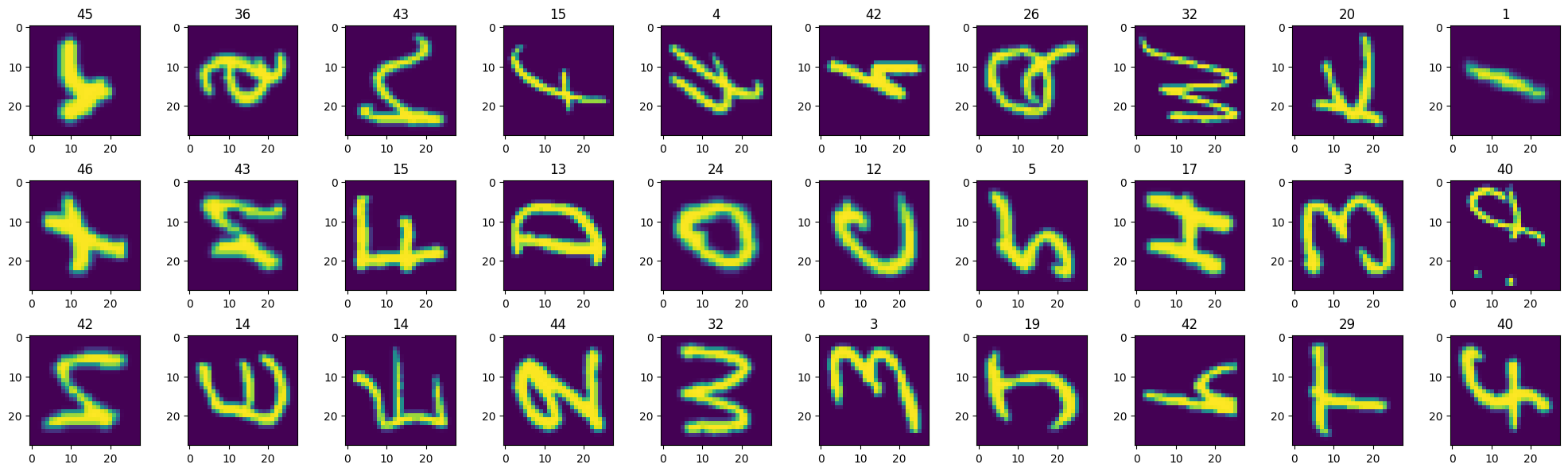


Figure 2.1.1 EMNIST Dataset Plot

Figure 2.1. and Figure 2.1.1 show

|  |
| --- |
| # Create the encoder network layers from the illustrated model plot  # Use the latent\_dim variable to define the probability distribution parameters  # Save the input layer to input\_img variable  # Save the mean layer output to z\_mean variable  # Save the logarithm of the variance layer output to z\_log\_var variable  # Save the sampling layer output to z\_sample variable  ### --YOUR CODE HERE-- ###  input\_img = Input(shape=(28,28,1))  x = Conv2D(32, 1, activation="relu")(input\_img)  print(x.shape)  m = MaxPooling2D(2)(x)  print(m.shape)  x1 = Conv2D(64, 1, activation="relu")(m)  print(x1.shape)  m1 = MaxPooling2D(2, padding="same")(x1)  print(m1.shape)  xm = Flatten()(m1)  print(xm.shape)  d = Dense(64, activation="relu")(xm)  print(d.shape)  z\_mean = Dense(latent\_dim, activation="linear")(d)  print(z\_mean.shape)  z\_log\_var = Dense(latent\_dim, activation="linear")(d)  print(z\_log\_var.shape)  z\_sample = Sampling()([z\_mean, z\_log\_var])  print(z\_sample.shape)  # Create the encoder network using the layers defined above  vae\_encoder = Model(input\_img, [z\_mean, z\_log\_var, z\_sample], name="encoder") |

Figure 2.2. Code for Building Encoder Network Layers

A black and white diagram

AI-generated content may be incorrect.

Figure 2.2.1 Decoder Subnetwork Architecture

Figure 2.2. and Figure 2.2.1 show

|  |
| --- |
| # Import functions and classes from Keras library  from keras.layers import Reshape, UpSampling2D  # Create the decoder network layers from the illustrated model plot  # Use the latent\_dim variable to define the probability distribution parameters  # Save the input layer to input\_latent variable  # Save the last convolutional layer output to decoded variable  ### --YOUR CODE HERE-- ###  input\_latent = Input(shape=(8,))  print(input\_latent.shape)  d = Dense(3136, activation="relu")(input\_latent)  print(d.shape)  r = Reshape((7,7,64))(d)  print(r.shape)  x = Conv2D(64, 1, activation="relu")(r)  print(x.shape)  u = UpSampling2D(2)(x)  print(u.shape)  x1 = Conv2D(32, 1, activation="relu")(u)  print(x1.shape)  u1 = UpSampling2D(2)(x1)  print(u1.shape)  decoded = Conv2D(1, 1, activation="sigmoid")(u1)  print(decoded.shape)  # Create the decoder network using the layers defined above  vae\_decoder = Model(input\_latent, decoded, name="decoder") |

Figure 2.3. Code for Building the Decoder Subnetwork

A black and white diagram

AI-generated content may be incorrect.

Figure 2.3.1 Decoder Subnetwork Architecture

As seen on Figure 2.3 and Figure 2.3.1

|  |
| --- |
| # Configure the network for training using the compile method  # Set the optimizer to your choice and determine the appropriate loss function  ### --YOUR CODE HERE-- ###  from keras.optimizers import Adam  from keras.callbacks import EarlyStopping  vae\_cnn.compile(optimizer="adam", loss='mse')  # Train the model. Set the batch size to 100  # Use early stopping to determine the appropriate training epochs  # Perform model hyperparameter tuning as needed  # Assign the output to hist\_vae\_cnn variable  ### --YOUR CODE HERE-- ###  callback = EarlyStopping(patience=5, monitor="total\_loss", mode='min')  hist\_vae\_cnn = vae\_cnn.fit(X\_train, batch\_size=100, epochs=100, callbacks=[callback], validation\_split=0.2) |

Figure 2.4. Code for Configuring the training VAE with Early Stopping

As seen on Figure 2.4

|  |
| --- |
| # Plot the history of training and validation losses  # Put the training losses subplots in the first row  # Put the validation losses subplots in the second row  ### --YOUR CODE HERE-- ###  figure, axs = plt.subplots(2, 3, figsize=(15,6))  train\_losses = [reconstruction\_losses, kl\_losses, losses]  validation\_losses = [validation\_reconstruction\_losses, validation\_kl\_losses, validation\_losses]  titles = ["Reconstruction Loss", "KL Divergence Loss", "Total Loss"]  colors = ["red", "blue", "green"]  for i in range(3):    axs[0, i].plot(epochs, train\_losses[i], label="Train", color=colors[i])    axs[0, i].set\_title(f"Training {titles[i]}")    axs[0, i].set\_xlabel("Epoch")    axs[0, i].set\_ylabel("Loss")    axs[0, i].legend()  for i in range(3):    axs[1, i].plot(epochs, train\_losses[i], label="Validation", color=colors[i])    axs[1, i].set\_title(f"Validation {titles[i]}")    axs[1, i].set\_xlabel("Epoch")    axs[1, i].set\_ylabel("Loss")    axs[1, i].legend()  plt.tight\_layout()  plt.show() |

Figure 2.5. Code for Plotting the Six Plots

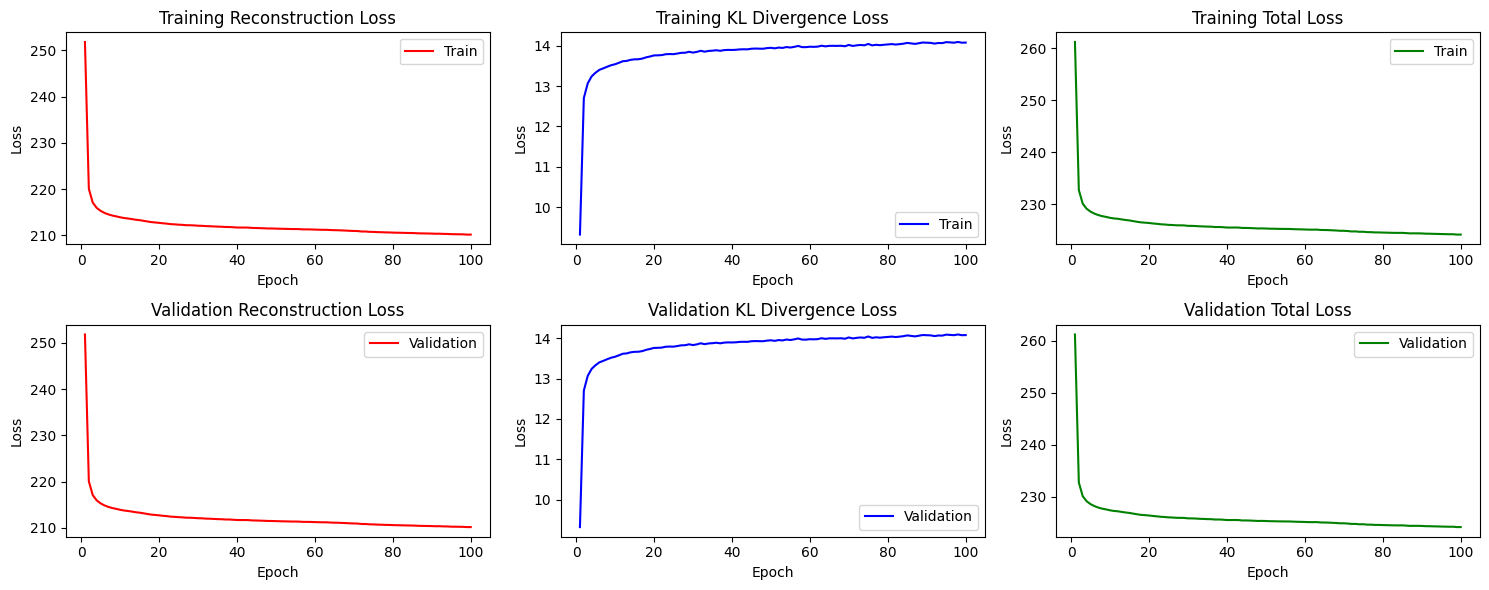


Figure 2.5.1 Training and Validation Reconstruction, Divergence, and Total Loss

As seen on Figure 2.5 and Figure 2.5.1

|  |
| --- |
| # Select at least 15 test images  # Feed the test images to the encoder network. Save it to z\_img variable  # Feed the encoder outputs to the decode network  ### --YOUR CODE HERE-- ###  num\_images = 15  test\_images = X\_test[:num\_images]  z\_mean, z\_log\_var, z\_img = vae\_cnn.encoder.predict(test\_images)  decoded\_images = vae\_cnn.decoder.predict(z\_img)  # Display the selected test images in the first row  # Display the resulting images in the second row  ### --YOUR CODE HERE-- ###  figure, axs = plt.subplots(2, num\_images, figsize=(20, 6))  for i in range(num\_images):    axs[0, i].imshow(test\_images[i])    axs[1, i].imshow(decoded\_images[i])  plt.tight\_layout()  plt.show() |

Figure 2.6 Code for Plotting the VAE Tested Images

A group of symbols in squares

AI-generated content may be incorrect.

Figure 2.6.1 Plot of the 15 VAE Tested Images

As seen on Figure 2.6 and Figure 2.6.1

|  |
| --- |
| # Select at least 1000 test images  # Feed the test images to the encoder network. Save it to z\_lat variable  ### --YOUR CODE HERE-- ###  num\_images = 1000  test\_images = X\_test[:num\_images]  z\_mean, z\_log\_var, z\_lat = vae\_cnn.encoder.predict(test\_images)  # Create a histogram for each latent space variable into subplots  # The subplots should be arrange into a grid with 2 rows and 4 columns  ### --YOUR CODE HERE-- ###  figure, axs = plt.subplots(2, 4, figsize=(15,6))  for i in range(8):    axs[i//4, i%4].hist(z\_lat[:i])    axs[i//4, i%4].set\_title(f"Latent Variable {i+1}")    axs[i//4, i%4].set\_xlabel("Value")    axs[i//4, i%4].set\_ylabel("Frequency")  plt.tight\_layout()  plt.show() |

Figure 2.7. Code for Histogram of all Eight Latent Variables

A group of graphs with different colored lines

AI-generated content may be incorrect.

Figure 2.7.1 Histogram Plot for the Eight Latent Variables

As seen on Figure 2.7 and Figure 2.7.1

|  |
| --- |
| # Select the test image and display them in a single row  ### --YOUR CODE HERE-- ###  lower1 = X\_test[0]  upper1 = X\_test[13]  upper2 = X\_test[1]  figure, axs = plt.subplots(1, 3, figsize=(4,4))  axs[0].imshow(lower1)  axs[1].imshow(upper1)  axs[2].imshow(upper2)  plt.tight\_layout()  plt.show()  # Obtain the output of the encoder for each test image  # Save the encoder output for the first image to z1 variable  # Save the encoder output for the second image to z2 variable  # Save the encoder output for the third image to z3 variable  # Perform the difference z1 - z2 and store to zd variable  # Perform the sum z3 + zd and store to z3new variable  ### --YOUR CODE HERE-- ###  z1, z\_log\_var, z\_img1 = vae\_cnn.encoder.predict(np.expand\_dims(lower1, axis=0))  z2, z\_log\_var, z\_img2 = vae\_cnn.encoder.predict(np.expand\_dims(upper1, axis=0))  z3, z\_log\_var, z\_img3 = vae\_cnn.encoder.predict(np.expand\_dims(upper2, axis=0))  zd = z1 - z2  z3new = z3 + zd  # Feed the new latent vector z3new to the decoder network  # Display the resulting image  ### --YOUR CODE HERE-- ###  decoded\_image = vae\_cnn.decoder.predict(z3new)  plt.figure(figsize=(4,4))  plt.imshow(decoded\_image.squeeze())  plt.tight\_layout()  plt.show() |

Figure 3.8. Instructions to get Difference Between Lower to Uppercase Letter with Plot

A yellow and green logo

AI-generated content may be incorrect.

Figure 3.8.1 Plot for Z1, Z2, and Z3

A colorful squares with numbers

AI-generated content may be incorrect.

Figure 3.8.2 Plot for Z3new

As seen on Figure 3.8, Figure 3.8.1, and Figure 3.8.2,

|  |
| --- |
| # Import libraries  from torchvision import transforms  from torchvision.datasets import EMNIST  # added these 3 imports  import tensorflow as tf  import numpy as np  import matplotlib.pyplot as plt  # Load the EMNIST dataset  emnist\_train = EMNIST("./data", split="balanced", train=True, download=True,                        transform=transforms.ToTensor()                        )  emnist\_test = EMNIST("./data", split="balanced", train=False, download=True,                       transform=transforms.ToTensor()                       )  # Define the generator function for Tensorflow dataset  def emnist\_ds\_generator(dataset):    for image, \_ in dataset:      yield np.transpose(image.numpy(), axes=(0, 2, 1)).reshape(28, 28, 1)  # Define the Tensorflow dataset output signature  emnist\_ds\_osig = tf.TensorSpec(shape=(28, 28, 1), dtype=tf.float32)  # Create the Tensorflow dataset  emnist\_ds\_train = tf.data.Dataset.from\_generator(      lambda: emnist\_ds\_generator(emnist\_train),      output\_signature=emnist\_ds\_osig  )  emnist\_ds\_test = tf.data.Dataset.from\_generator(      lambda: emnist\_ds\_generator(emnist\_test),      output\_signature=emnist\_ds\_osig  )  emnist\_ds = emnist\_ds\_train.concatenate(emnist\_ds\_test)  emnist\_ds = emnist\_ds\_train.concatenate(emnist\_ds\_test)  emnist\_ds = emnist\_ds.shuffle(buffer\_size=1000)  # Display 30 images from the Tensorflow dataset in three rows (10 images per row)  ### --YOUR CODE HERE-- ###  sample\_images = list(emnist\_ds.take(30))  fig, axes = plt.subplots(3, 10, figsize=(15, 6))  for i, image in enumerate(sample\_images):      row = i // 10  # get which row (0, 1, or 2)      col = i % 10   # get which column (0-9)      img\_array = image.numpy().squeeze()      axes[row, col].imshow(img\_array, cmap='gray')      axes[row, col].axis('off')  plt.tight\_layout()  plt.suptitle('30 Random Images from EMNIST Dataset')  plt.show() |

Figure 3.9. Code for Loading EMNIST Image Dataset

A black and white photo of letters

AI-generated content may be incorrect.

Figure 3.9.1 Plot for Displaying 30 EMNIST Images

As seen in Figure 3.9 and Figure 3.9.1

|  |
| --- |
| # Import functions and classes from Keras library  from keras import Input, Sequential  from keras.layers import LeakyReLU, Dense, Dropout  from keras.layers import Conv2D, Flatten  # Create the discriminator. Save it to gan\_discriminator variable  # Convolutions should have a filter size of 5 and a stride of 2  # Dropout rate is set to 0.3  ### --YOUR CODE HERE-- ###  gan\_discriminator = Sequential([      Input(shape=(28, 28, 1)),      Conv2D(64, kernel\_size=5, strides=2, padding='same'),      LeakyReLU(alpha=0.2),      Dropout(0.3),      Conv2D(128, kernel\_size=5, strides=2, padding='same'),      LeakyReLU(alpha=0.2),      Dropout(0.3),      Flatten(),      Dense(1)  ])  gan\_discriminator.summary() |

Figure 3.10 Code for Making the Discriminator

A black and white diagram

AI-generated content may be incorrect.

Figure 3.10.1 Discriminator Subnetwork Architecture

As seen on Figure 3.10 and Figure 3.10.1,

|  |
| --- |
| # Import functions and classes from Keras library  from keras import Input, Sequential  from keras.layers import LeakyReLU, Dense  from keras.layers import Reshape, Conv2DTranspose, BatchNormalization  # Specify the latent space dimension  latent\_dim = 128  # Create the generator. Save it to gan\_generator variable  # Transpose convolutions should have a filter size of 5  # The first transpose convolution has stride of 1. Others have stride of 2  ### --YOUR CODE HERE-- ###  gan\_generator = Sequential([      Input(shape=(latent\_dim,)),      # Dense layer: 128 -> 12544 (7\*7\*256 = 12544)      Dense(12544),      BatchNormalization(),      LeakyReLU(alpha=0.2),      Reshape((7, 7, 256)),      Conv2DTranspose(128, kernel\_size=5, strides=1, padding='same'),      BatchNormalization(),      LeakyReLU(alpha=0.2),      Conv2DTranspose(64, kernel\_size=5, strides=2, padding='same'),      BatchNormalization(),      LeakyReLU(alpha=0.2),      Conv2DTranspose(1, kernel\_size=5, strides=2, padding='same', activation='relu')  ]) |

Figure 3.11. Code making the Generator

A black and white diagram

AI-generated content may be incorrect.

Figure 3.11.1 Generator Subnetwork

As seen on Figure 3.11 and Figure 3.11.1

|  |
| --- |
| # Configure the network for training  from keras.optimizers import Adam  from keras.losses import BinaryCrossentropy  gan.compile(      d\_optimizer=Adam(learning\_rate=0.0002),      g\_optimizer=Adam(learning\_rate=0.0002),      loss\_fn=BinaryCrossentropy(from\_logits=True),  )  # Train the model. Set the dataset batch size to 100  # You may take 100 batches only to speed up the training  # Assign the output to gan\_hist variable  ### --YOUR CODE HERE-- ###  batch\_size = 100  batch\_count = 100  train\_dataset = emnist\_ds.batch(batch\_size).take(batch\_count)  gan\_hist = gan.fit(      train\_dataset,      epochs = 100,      verbose = 1,  )  print(f"Final Generator Loss: {gan\_hist.history['g\_loss'][-1]:.4f}")  print(f"Final Discriminator Loss: {gan\_hist.history['d\_loss'][-1]:.4f}") |

Figure 3.12. Code for Training the EMNIST using the GAN Generator and Discriminator

As seen on Figure 3.12

|  |
| --- |
| # Extract the losses during training  dis\_losses = gan\_hist.history["d\_loss"]  gen\_losses = gan\_hist.history["g\_loss"]  epochs = range(1, len(dis\_losses) + 1)  # Plot the history of training and validation losses  ### --YOUR CODE HERE-- ###  import importlib  import matplotlib.pyplot  importlib.reload(matplotlib.pyplot)  import matplotlib.pyplot as plt  plt.figure(figsize=(10,6))  plt.plot(epochs, gen\_losses, 'b-', label='Generator Loss', linewidth=2, marker='o')  plt.plot(epochs, dis\_losses, 'r-', label='Discriminator Loss', linewidth=2, marker='s')  plt.title('Custom GAN: Generator and Discriminator Losses per Epoch')  plt.xlabel('Epoch')  plt.ylabel('Loss')  plt.legend()  plt.grid(True, alpha=0.3)  plt.tight\_layout()  plt.show() |

Figure 3.13. Code for Plotting the Generator and Discriminator Loss Plot

As seen on Figure 3.13

|  |
| --- |
| # Import functions and classes from Keras library  from keras.random import normal  import numpy as np  import matplotlib.pyplot as plt  importlib.reload(matplotlib.pyplot)  # Generate 30 random 128-dimensional vectors as input  # Feed the random inputs to the generator network  # Display the output images of the generator  ### --YOUR CODE HERE-- ###  num\_samples = 30  random\_inputs = normal(shape=(num\_samples, latent\_dim), seed=42)  generated\_images = gan\_generator.predict(random\_inputs, verbose=0)  plt.figure(figsize=(15, 10))  for i in range(num\_samples):      plt.subplot(5, 6, i+1)      img = generated\_images[i].squeeze()      plt.imshow(img, cmap='gray')      plt.axis('off')      plt.title(f'#{i+1}', fontsize=8)  plt.suptitle('Random Inputs', fontsize=14)  plt.tight\_layout()  plt.show()  # Add a small random offsets to the initial inputs  # Feed the new input to the generator network  # Display the new output images of the generator  ### --YOUR CODE HERE-- ###  offset\_magnitude = 0.1  random\_offsets = normal(shape=(num\_samples, latent\_dim), seed=123) \* offset\_magnitude  new\_inputs = random\_inputs + random\_offsets  new\_generated\_images = gan\_generator.predict(new\_inputs, verbose=0)  plt.figure(figsize=(15, 10))  for i in range(num\_samples):      plt.subplot(5, 6, i+1)      img = new\_generated\_images[i].squeeze()      plt.imshow(img, cmap='gray')      plt.axis('off')      plt.title(f'#{i+1}', fontsize=8)  plt.suptitle('Generated output images', fontsize=14)  plt.tight\_layout()  plt.show() |

Figure 3.14. Code for Feeding Generator with 30 Random Inputs

A collage of images of a person's body

AI-generated content may be incorrect.

Figure 3.14.1 Plot for Displaying the Random Inputs Images

A collage of images of a person's body

AI-generated content may be incorrect.

Figure 3.14.2 Plot for Displaying the Generated Output Images

As seen on Figure 3.14, Figure 3.14.1, and Figure 3.14.2 are the images generated by a GAN (Generative Adversarial Network) using random 128-dimensional noise vectors as inputs. Each image represents the generator’s interpretation of a unique point in the latent space, resulting in slightly different outputs that resemble similar abstract patterns. The first figure displays the initial random samples, while the second figure shows new images generated after adding small random offsets to the original inputs. This demonstrates the GAN’s sensitivity to variations in latent space, where small input changes lead to gradual and coherent changes in the generated outputs, indicating that the generator has learned a smooth and continuous representation of the data distribution.

### Answer to Guide Questions, Insights, and Reflections

For the insights and reflections, due to the lack of time we weren’t able to train the correct custom GAN generator and discriminator because the generated output images on Figure 3.14.1 and Figure 3.14.2 do not resemble handwritten letters. This can also be seen on the generator and discriminator loss plot where the loss for the generator is increasing as the epoch increases from 1.0 to 2.0. Despite this the rest of the item’s have sufficient results.

## Activity 3: Text Generation Models

### Objectives

* Familiarize with the architecture of LSTM and GRU models
* Familiarize with the transformer architecture
* Build and train various text generation models using deep learning framework
* Assess the performance of the model's ability to generate text accurately

### Experimental Procedure

Preprocess the WikiAuto data by extracting the simple\_sentence field, build character-level vocabularies and mappings, create sliding-window input–target pairs, and convert them into tf.data datasets for training and testing; then define and build an LSTM-based sequence model with an Embedding layer, LSTM layer, Dropout and a final softmax Dense layer, compile it with categorical crossentropy and Adam, train it while saving the loss history, plot the training loss per epoch, and generate at least five text samples by seeding the model with test snippets while measuring performance with the Perplexity metric; repeat text generation experiments with different temperature values to observe sampling diversity; implement a GRU variant with the same preprocessing, training, and evaluation steps, record training time and perplexity, and compare results with the LSTM; for the transformer section, vectorize the RACE dataset at the word level using TextVectorization, define TokenAndPositionEmbedding and TransformerBlock custom layers, assemble the GPT-style model with token/position embeddings, transformer block, and final linear outputs, compile with a sparse categorical loss, train for the specified epochs while plotting loss, generate multiple samples from different seed sentences using top-k sampling, continue training until validation loss plateaus, regenerate outputs to assess qualitative improvement and the frequency of unknown tokens,

### Results and Analysis

|  |
| --- |
| # Get the wiki-auto dataset. Load the wiki\_auto/manual configuration  # https://www.tensorflow.org/datasets/catalog/wiki\_auto  # Save the dev split to variable ds\_train  # Save the test split to variable ds\_test  # Save the dataset info to variable ds\_info  ### --YOUR CODE HERE-- ###  datasets, ds\_test = tfds.load("wiki\_auto/manual", split = {"dev": "dev", "test": "test"}, with\_info=True)  ds\_info = tfds.builder("wiki\_auto/manual").info  ds\_train = datasets["dev"]  ds\_test = datasets["test"]  # Preprocess the dataset to include only the 'simple\_sentence' feature  ds\_train = ds\_train.map(lambda x: x["simple\_sentence"])  ds\_test = ds\_test.map(lambda x: x["simple\_sentence"])  ds\_train\_text = str()  for text in ds\_train:  ds\_train\_text += text.numpy().decode() + "\n"  ds\_test\_text = str()  for text in ds\_test:  ds\_test\_text += text.numpy().decode() + "\n"  # Display at least five (5) sample texts from the dataset, one per row  ### --YOUR CODE HERE-- ###  for text in ds\_train.take(5):  print(text.numpy().decode()) |

Figure 3.1. Loading and Plotting of WikiAutoDataset Code

A black screen with white text

AI-generated content may be incorrect.

Figure 3.1.1 Sample output of WikiAutoDataset

Figure 3.1 shows the code for splitting the WikiAutoDataset which is a text based dataset rather than an image dataset. It is divided into info, train, and test. This is then preprocessed to simple sentences. Likewise, Figure 3.1.1 shows the 5 sample tests from the dataset per row.

The given code on item 2 prepares character-level training and testing datasets for a text generation model using Keras and TensorFlow. It first defines a sequence length of 50 and builds a vocabulary of all unique characters found in both the training and testing texts, then maps each character to a unique index (`chr2idx`) and vice versa (`idx2chr`). Next, it constructs training input–target pairs using a sliding window approach: for every 50-character sequence, the following character is used as the target label, with the window moving by 10 characters (`slide\_size`). These sequences are then converted into tensors, and the target characters are one-hot encoded using `to\_categorical`, creating numerical data suitable for neural network training. The same process is repeated for the test dataset, limited to the first 1,000,000 characters for efficiency. Finally, both the training and testing datasets are wrapped into TensorFlow `Dataset` objects (`ds\_train` and `ds\_test`) to streamline batching and model training.

|  |
| --- |
| # Import functions and classes from Keras library  from keras import Input, Sequential  from keras.layers import Embedding, LSTM, Dropout, Dense  # Create the RNN that uses LSTM. Save it to rnn\_lstm variable  # The Embedding layer should have 128-dimensional output vector  # Yuu may use a reasonable rate for the Dropout layer  ### --YOUR CODE HERE-- ###  rnn\_lstm = Sequential()  rnn\_lstm.add(Input(shape=(50,)))  rnn\_lstm.add(Embedding(input\_dim=vocab\_size, output\_dim=128))  rnn\_lstm.add(LSTM(256, return\_sequences=False))  rnn\_lstm.add(Dropout(0.3))  rnn\_lstm.add(Dense(vocab\_size, activation='softmax'))  model\_summary = rnn\_lstm.summary()  print(model\_summary) |

Figure 3.2. Code for the RNN model using the LSTM layer

Figure 3.2 builds this RNN model using the LSTM layer that contains several layers that work together to perform character-level text generation. It begins with an Input layer that accepts sequences of 50 characters, defining the shape of the input data. Next is an Embedding layer with an input dimension equal to the vocabulary size and an output dimension of 128, which converts each character index into a dense 128-dimensional vector representation. This is followed by an LSTM layer with 256 units and return\_sequences=False, meaning it outputs only the final hidden state that summarizes the information from the entire sequence. After that, a Dropout layer with a dropout rate of 0.3 is applied to reduce overfitting by randomly deactivating 30% of neurons during training. Finally, the model ends with a Dense layer whose output dimension equals the vocabulary size and uses a softmax activation function to generate a probability distribution over all possible next characters, allowing the model to predict the most likely next character in the sequence.

|  |
| --- |
| # Configure the network for training  # Use the appropriate loss function  # You may use Adam as optimizer  ### --YOUR CODE HERE-- ###  rnn\_lstm.compile(loss='categorical\_crossentropy', optimizer='adam', metrics=['accuracy'])  # Train the model. Set the number of epochs accordingly  # Set the dataset batch size of your choice  # Assign the output to rnn\_lstm\_hist variable  ### --YOUR CODE HERE-- ###  rnn\_lstm\_hist = rnn\_lstm.fit(ds\_train.batch(128), epochs=10) |

Figure 3.3 Network Configuration for Training the RNN

Figure 3.3 show the network configuration for training the RNN which contains a loss function of categorical cross entropy with an Adam optimizer and an accuracy metric. It is also trained with a batch size of 128 and epochs of 10. The plot can be seen on Figure 3.4.1 and the code for plotting it is on Figure 3.4. Notice that the loss decreases exponentially to less than 0.5 at the 6th epoch.

|  |
| --- |
| # Extract the losses during training  ### --YOUR CODE HERE-- ###  rnn\_lstm\_loss = rnn\_lstm\_hist.history['loss']  # Plot the history of losses  ### --YOUR CODE HERE-- ###  plt.plot(rnn\_lstm\_loss)  plt.title('Training Loss')  plt.xlabel('Epoch')  plt.ylabel('Loss')  plt.show() |

Figure 3.4 Code for Plotting the Training Loss of the RNN

A graph of a training loss

AI-generated content may be incorrect.

Figure 3.4.1 Training Loss per Epoch

|  |
| --- |
| # Define the function to convert model output to character  def predictions\_to\_char(predictions, temperature=1.0):  predictions = np.asarray(predictions).astype("float64")  exp\_predictions = np.exp(np.log(predictions) / temperature)  predictions = exp\_predictions / np.sum(exp\_predictions)  probabilities = np.random.multinomial(1, predictions, 1)  return idx2chr[np.argmax(probabilities)]  # Get at least (5) seed sentences from the test dataset  # Obtain model output for each sentence and use the function defined above  # Generate a sentence that is at least 100 characters in length  # after the seed sentence.  # Save the seed sentences to a list (not TFDS) named seed\_sentences  # Save the next character for each sentence to a list named next\_chars  ### --YOUR CODE HERE-- ###  ds\_test\_sentences = ds\_test\_text.split("\n")  seed\_sentences = [s[:50] for s in ds\_test\_sentences if len(s) >= 50][:5]  next\_chars = [s[50] if len(s) > 50 else " " for s in ds\_test\_sentences[:5]]  for idx, seed\_sentence in enumerate(seed\_sentences):  print(f"\n=== Sample sentence {idx + 1} ===")  print(seed\_sentence, end="")  generated = seed\_sentence  for \_ in range(100):  # Use last 50 characters as input window  input\_seq = generated[-50:]  input\_tokens = tf.convert\_to\_tensor([[chr2idx[ch] for ch in input\_seq]])  # Predict next character  predictions = rnn\_lstm.predict(input\_tokens, verbose=0)  next\_char = predictions\_to\_char(predictions[0], temperature=0.5)  # Append next char and continue  generated += next\_char  print(next\_char, end="")  print("\n")  # Compute the perplexity metric for each seed sentence as you selected above  # Use the saved next character as ground truth  # Import Perplexity metric from Keras NLP  from keras\_hub.metrics import Perplexity  ### --YOUR CODE HERE-- ###  perplexity\_metric = Perplexity(from\_logits=False)  for i, seed\_sentence in enumerate(seed\_sentences):  input\_tokens = tf.convert\_to\_tensor([[chr2idx[ch] for ch in seed\_sentence]])  target\_token = tf.convert\_to\_tensor([[chr2idx[next\_chars[i]]]])  predictions = rnn\_lstm.predict(input\_tokens, verbose=0)  perplexity\_value = perplexity\_metric(target\_token, predictions).numpy()  print(f"Perplexity for Sample {i + 1}: {perplexity\_value:.4f}") |

Figure 3.5. Code for Assessing Performance of the RNN model

Figure 3.5 shows how the trained RNN model generates new text and evaluates its performance using the Perplexity metric. It first extracts five seed sentences from the test dataset, each containing 50 characters, and stores them in a list named seed\_sentences, while their corresponding next characters are saved in next\_chars. For each seed sentence, the model generates an additional 100 characters by repeatedly predicting the next character based on the most recent 50-character window. Each prediction is processed through the predictions\_to\_char() function, which applies temperature scaling to control randomness and samples the next character from the resulting probability distribution. The predicted character is appended to the growing sentence, producing a continuous, coherent text sequence. After generating text, the Perplexity metric from Keras NLP is used to assess how well the model predicts the actual next character for each seed sentence.

A screenshot of a computer

AI-generated content may be incorrect.

Figure 3.5.1 Sample Output with Perplexity

Figure 3.5.1 shows five sample sentences generated by the LSTM text generation model, each followed by a calculated perplexity score, which measures how well the model predicts the next character in a sequence. A lower perplexity value indicates better model confidence and prediction accuracy, while higher values suggest greater uncertainty. In this case, the perplexities vary widely from around 27.1 to 713.8, showing that the model performs inconsistently across different text samples. Sentences with lower perplexity, such as Samples 3 and 4, are generally more coherent and grammatically stable, indicating that the model was more confident in its character predictions. On the other hand, sentences with very high perplexity, like Sample 1, suggest the model struggled with those sequences, likely due to unusual words, character combinations, or limited training examples resembling that text.

|  |
| --- |
| # Use the following temperature values  temp\_vals = [0.2, 0.5, 1.5, 2.0]  # Repeat step A.6 but with different temperature values defined above  ### --YOUR CODE HERE-- ###  for temp in temp\_vals:  print(f"\n{'='\*25}")  print(f"Generating text with temperature = {temp}")  print(f"{'='\*25}")  for idx, seed\_sentence in enumerate(seed\_sentences):  print(f"\n--- Sample sentence {idx + 1} (T={temp}) ---")  print(seed\_sentence, end="")  generated = seed\_sentence  for \_ in range(100): # generate 100 new characters  seed\_tokens = tf.convert\_to\_tensor([[chr2idx[ch] for ch in generated[-50:]]])  predictions = rnn\_lstm.predict(seed\_tokens, verbose=0)  next\_char = predictions\_to\_char(predictions[0], temperature=temp)  generated += next\_char  print(next\_char, end="")  print("\n") |

Figure 3.6 Code with Different temperature and predictions to char function

Figure 3.6 differs from Figure 3.5 because it introduces temperature scaling as a variable factor to control the randomness and creativity of the model’s predictions. Instead of using a fixed temperature value such as 0.5 in the earlier code, this script tests multiple temperature settings 0.2, 0.5, 1.5, and 2.0 and generates text for each one. The temperature parameter adjusts how the model samples from the probability distribution of predicted characters; lower temperatures like 0.2 make the output more deterministic and focused on the most probable characters, resulting in repetitive or conservative text, while higher temperatures like 1.5 or 2.0 introduce more randomness, making the output more diverse but potentially less coherent. The loop structure ensures that for each temperature, the model generates new 100-character continuations of the seed sentences, allowing comparison of how temperature affects text fluency, creativity, and variability.

|  |
| --- |
| =========================  Generating text with temperature = 0.2  =========================  --- Sample sentence 1 (T=0.2) ---  In May 2002, Frank Iero joined as the rhythm guital of the Lonely Mountain.  The dwarves are attacked and captured by giant spiders.  The dwarves are at  --- Sample sentence 2 (T=0.2) ---  High achievers in fields like music, mathematics and host and movies.  The dwarves are attacked and captured by giant spiders.  The dwarves are attacked  --- Sample sentence 3 (T=0.2) ---  People could live inside of it, and also store this tensis, and the most know part of the Freetown.  The dwarves are attacked and captured by giant spi  --- Sample sentence 4 (T=0.2) ---  He is known for his roles as Tom Frank in Robert Accently alto wather armans become home to several businesses such as carpenters, blocks and movies.  --- Sample sentence 5 (T=0.2) ---  There were four protective towers for the Inner Warkine the Egyptian Anger and the Countian centre for sear comminication of the hobbit Bilbo Baggins  =========================  Generating text with temperature = 0.5  =========================  --- Sample sentence 1 (T=0.5) ---  In May 2002, Frank Iero joined as the rhythm guitally making the through the strategically from the police found the most weapons features from show a  --- Sample sentence 2 (T=0.5) ---  High achievers in fields like music, mathematics and host, and tries to find a way out by himself.  After the military had left, the area was only guar  --- Sample sentence 3 (T=0.5) ---  People could live inside of it, and also store thins and hydroxide is an also tunnels east by theee minustion and sometimes of homeless people went in  --- Sample sentence 4 (T=0.5) ---  He is known for his roles as Tom Frank in Robert Are on the 2011 census to the sites the dragon Smaug, which has stolen the treasures and home of the  --- Sample sentence 5 (T=0.5) ---  There were four protective towers for the Inner Warsial Ware Ludergen 5, the Mountain".  The story takes place before "The Lord of the Rings".  The move  =========================  Generating text with temperature = 1.5  =========================  --- Sample sentence 1 (T=1.5) ---  In May 2002, Frank Iero joined as the rhythm guitaull bring king under times.  People sile the Christianites aftourd not many many for the tupul that t  --- Sample sentence 2 (T=1.5) ---  High achievers in fields like music, mathematics aleamon's where the lother.  Spitsbergen Vitc are sodium.  At the eny "Tunall" Pather Nat has been besm  --- Sample sentence 3 (T=1.5) ---  People could live inside of it, and also store thias in the citying langain Represent" in Aunequel to Uninter engenined that Bilbo flaw scowed by Norb  --- Sample sentence 4 (T=1.5) ---  He is known for his roles as Tom Frank in Robert Amer, Elvos ;orbine Kann"%lísida.  Ancend aghant in Christiania realise "Batthes.  He Murata", athth C  --- Sample sentence 5 (T=1.5) ---  There were four protective towers for the Inner Warkine the budning is one of the "Canstans in Durs 1995, is Bilbo the host, Motely Mountain, ved to t  =========================  Generating text with temperature = 2.0  =========================  --- Sample sentence 1 (T=2.0) ---  In May 2002, Frank Iero joined as the rhythm guitain Gandulf and noims pridacents and rack for 9-kil well-anc Maustor Tara, Aquin U Ae, 1950s Refrem G  --- Sample sentence 2 (T=2.0) ---  High achievers in fields like music, mathematics and the chemain "habitian Agais and Thorin's pockemaly people movies have waters devesil Ornound were  --- Sample sentence 3 (T=2.0) ---  People could live inside of it, and also store thing-dorughord, elvistean.  Second, now and dissures in Anyla00: burna140, gil Coritagy Or Ruyshar Frot  --- Sample sentence 4 (T=2.0) ---  He is known for his roles as Tom Frank in Robert A re:0 Dáin Aru, N sherc sipH were Emitsanning mieding which he would gelaured ovomioss gives them.  --- Sample sentence 5 (T=2.0) ---  There were four protective towers for the Inner Wars, and Kíli Deariveltives raured, Edran" loibs.  Smarcishenc, Christiny teach eallogy, "Chrusp, in |

Figure 3.6.1 Output Text with Temperature of 0.2

Figure 3.6.1 shows text generated by the LSTM model at different temperature values, demonstrating how temperature affects creativity and coherence in text generation. At low temperatures like 0.2, the output is more repetitive and predictable but grammatically stable, while at moderate temperatures like 0.5, the text becomes slightly more varied and natural. At higher temperatures such as 1.5 and 2.0, the text becomes much more random and less coherent, showing that increasing temperature introduces more creativity but reduces linguistic consistency.

|  |
| --- |
| # Create the RNN that uses GRU. Save it to rnn\_gru variable  ### --YOUR CODE HERE-- ###  import time  import matplotlib.pyplot as plt  from keras import Input, Sequential  from keras.layers import Embedding, GRU, Dropout, Dense  from keras\_hub.metrics import Perplexity  rnn\_gru = Sequential()  rnn\_gru.add(Input(shape=(50,)))  rnn\_gru.add(Embedding(input\_dim=vocab\_size, output\_dim=128))  rnn\_gru.add(GRU(256, return\_sequences=False))  rnn\_gru.add(Dropout(0.3))  rnn\_gru.add(Dense(vocab\_size, activation='softmax'))  rnn\_gru.summary()  rnn\_gru.compile(      optimizer='adam',      loss='categorical\_crossentropy',      metrics=['accuracy']  )  # Configure the network for training  # Use the Perplexity as an additional metric during training  # Assign the output to rnn\_gru\_hist variable  ### --YOUR CODE HERE-- ###  start\_time = time.time()  rnn\_gru\_hist = rnn\_gru.fit(ds\_train.batch(128), epochs=10, verbose=1)  training\_time = time.time() - start\_time  print(f"\nTotal training time (GRU model): {training\_time:.2f} seconds")  # Plot the history of losses and the perplexity values per epoch  ### --YOUR CODE HERE-- ###  loss\_values = rnn\_gru\_hist.history['loss']  perplexity\_values = np.exp(loss\_values)  plt.figure(figsize=(10,4))  plt.subplot(1,2,1)  plt.plot(loss\_values, label='Loss')  plt.title('GRU Model - Training Loss per Epoch')  plt.xlabel('Epoch')  plt.ylabel('Loss')  plt.legend()  plt.subplot(1,2,2)  plt.plot(perplexity\_values, label='Perplexity', color='orange')  plt.title('GRU Model - Perplexity per Epoch')  plt.xlabel('Epoch')  plt.ylabel('Perplexity')  plt.legend()  plt.show()  # Obtain at least five (5) text predictions from the model  ### --YOUR CODE HERE-- ###  def predictions\_to\_char(predictions, temperature=1.0):      predictions = np.asarray(predictions).astype("float64")      exp\_predictions = np.exp(np.log(predictions) / temperature)      predictions = exp\_predictions / np.sum(exp\_predictions)      probabilities = np.random.multinomial(1, predictions, 1)      return idx2chr[np.argmax(probabilities)]  # Reuse the same seed sentences from the LSTM experiment  for idx, seed\_sentence in enumerate(seed\_sentences):      print(f"\n=== GRU Sample sentence {idx + 1} ===")      print(seed\_sentence, end="")      generated = seed\_sentence      for \_ in range(100):  # generate 100 new characters          seed\_tokens = tf.convert\_to\_tensor([[chr2idx[ch] for ch in generated[-50:]]])          predictions = rnn\_gru.predict(seed\_tokens, verbose=0)          next\_char = predictions\_to\_char(predictions[0], temperature=0.5)          generated += next\_char          print(next\_char, end="")      print("\n") |

Figure 3.7. Code for Training RNN using GRU

Figure 3.7 builds and trains a RNN using a GRU (Gated Recurrent Unit) layer instead of an LSTM for character-level text generation. It defines a model with an embedding layer, a GRU layer of 256 units, dropout for regularization, and a dense softmax output layer, then compiles and trains it for ten epochs while tracking loss and computing perplexity to evaluate performance. After training, it plots the loss and perplexity per epoch and generates text samples from the trained GRU model using the same seed sentences as the LSTM model to compare the two architectures.

A graph of a graph of a graph

AI-generated content may be incorrect.

Figure 3.7.1 Training Loss and Perplexity per Epoch for GRU Model

Figure 3.7.1 shows that the GRU model’s training loss and perplexity both decrease rapidly during the first few epochs, indicating effective learning and improvement in prediction accuracy. However, after around the fourth epoch, both metrics begin to rise slightly, suggesting that the model may be starting to overfit or lose generalization efficiency as training continues.

|  |
| --- |
| === GRU Sample sentence 1 ===  In May 2002, Frank Iero joined as the rhythm guitashopentices wat the foreed of the architectural competition.  The group contry feats the dwarves and  === GRU Sample sentence 2 ===  High achievers in fields like music, mathematics and treasures for the tubl them and the group of dwarves are attacked by Goblins in Morylay and Kíli  === GRU Sample sentence 3 ===  People could live inside of it, and also store things their or miest of the hash of the shownush parts of the dwarves.  The top of the group from the r  === GRU Sample sentence 4 ===  He is known for his roles as Tom Frank in Robert Angensing at the heman partic or make a ring, the dwarves.  The top of the treasures wart are attacked  === GRU Sample sentence 5 ===  There were four protective towers for the Inner War Inding at atmesicing to the Lonely Mountain.  In order to presine productically far pridiced a thri |

Figure 3.7.2 GRU Sample Sentences

Figure 3.7.2 shows sentences generated by the GRU-based text generation model using the same seed inputs as the LSTM model. The generated text is mostly coherent and contextually related to the input, demonstrating that the GRU effectively captures patterns and dependencies within the training data. However, some phrases appear repetitive or slightly nonsensical, indicating that while the GRU performs well in maintaining structure, it still struggles with long-term coherence and realistic language flow.

|  |
| --- |
| # Get the race dataset. Load the race/high configuration  # https://www.tensorflow.org/datasets/catalog/race  # Do not split the dataset. Combine all splits  # Save the dataset into race\_ds variable  # Save the dataset info to variable ds\_info  ### --YOUR CODE HERE-- ###  import tensorflow\_datasets as tfds  ds\_info = tfds.builder('race/high').info  train\_ds = tfds.load('race/high', split='train', as\_supervised=False)  dev\_ds = tfds.load('race/high', split='dev', as\_supervised=False)  test\_ds = tfds.load('race/high', split='test', as\_supervised=False)  race\_ds = train\_ds.concatenate(dev\_ds).concatenate(test\_ds)  # Preprocess the dataset to include only the 'article' feature  ### --YOUR CODE HERE-- ###  race\_ds = race\_ds.map(lambda x: x['article'])  # Display a sample text from the dataset  ### --YOUR CODE HERE-- ###  for sample in race\_ds.take(1):      print(sample.numpy().decode('utf-8')) |

Figure 3.8 Code for Importing and Preprocess of Race Dataset

Figure 3.8 loads the RACE dataset with the high school configuration from TensorFlow Datasets and combines the training, development, and test splits into one dataset. It then preprocesses the data to keep only the article text feature which contains the main content for each sample. Finally it prints one sample article from the dataset to verify that the data was loaded and processed correctly.

|  |
| --- |
| WARNING:absl:Variant folder /root/tensorflow\_datasets/race/high/2.0.0 has no dataset\_info.json  Downloading and preparing dataset Unknown size (download: Unknown size, generated: Unknown size, total: Unknown size) to /root/tensorflow\_datasets/race/high/2.0.0...  Dl Completed...: 100%   1/1 [01:03<00:00, 12.99s/ url]  Dl Size...: 100%   24/24 [01:03<00:00,  1.95 MiB/s]  Extraction completed...: 100%   27933/27933 [01:03<00:00, 1525.61 file/s]  WARNING:absl:`TensorInfo.dtype` is deprecated. Please change your code to use NumPy with the field `TensorInfo.np\_dtype` or use TensorFlow with the field `TensorInfo.tf\_dtype`.  WARNING:absl:`TensorInfo.dtype` is deprecated. Please change your code to use NumPy with the field `TensorInfo.np\_dtype` or use TensorFlow with the field `TensorInfo.tf\_dtype`.  WARNING:absl:`TensorInfo.dtype` is deprecated. Please change your code to use NumPy with the field `TensorInfo.np\_dtype` or use TensorFlow with the field `TensorInfo.tf\_dtype`.  WARNING:absl:`TensorInfo.dtype` is deprecated. Please change your code to use NumPy with the field `TensorInfo.np\_dtype` or use TensorFlow with the field `TensorInfo.tf\_dtype`.  WARNING:absl:`TensorInfo.dtype` is deprecated. Please change your code to use NumPy with the field `TensorInfo.np\_dtype` or use TensorFlow with the field `TensorInfo.tf\_dtype`.  WARNING:absl:`TensorInfo.dtype` is deprecated. Please change your code to use NumPy with the field `TensorInfo.np\_dtype` or use TensorFlow with the field `TensorInfo.tf\_dtype`.  WARNING:absl:`TensorInfo.dtype` is deprecated. Please change your code to use NumPy with the field `TensorInfo.np\_dtype` or use TensorFlow with the field `TensorInfo.tf\_dtype`.  WARNING:absl:`TensorInfo.dtype` is deprecated. Please change your code to use NumPy with the field `TensorInfo.np\_dtype` or use TensorFlow with the field `TensorInfo.tf\_dtype`.  WARNING:absl:`TensorInfo.dtype` is deprecated. Please change your code to use NumPy with the field `TensorInfo.np\_dtype` or use TensorFlow with the field `TensorInfo.tf\_dtype`.  WARNING:absl:`TensorInfo.dtype` is deprecated. Please change your code to use NumPy with the field `TensorInfo.np\_dtype` or use TensorFlow with the field `TensorInfo.tf\_dtype`.  Dataset race downloaded and prepared to /root/tensorflow\_datasets/race/high/2.0.0. Subsequent calls will reuse this data.  Answering the Community Needs of Our City  The Silver City Council recognizes that citizens have certain needs. To better meet your needs, we have made several changes to community facilities in 2004. This chart shows how we have tried to make your life better.  Transport Three stations for the suburbs have been added to the western train service.20 new buses for the southern line were purchased in January. 50 percent of city bus-stops have been upgraded . Buses to the eastern suburbs will run every15 minutes.  Communication Broadband cable is now available to all parts of the city. All of the new Government buildings are ' smart'-wired for better computer service!  Medical Facilities The new state-of-the-art Nightingale Hospital was opened in June. To overcome a shortage of trained medical staff at Dover Hospital, 10 doctors have been employed from overseas.Some facilities at Station Street Hospital have been upgraded.  Education Textbooks will be free to all primary students in 2004 ! Rent for private schools has been reduced. Teachers report that the 'no hat - no play' rule has been successful.  Protection and Security Extra police now patrol ( ) the tourist areas. 50 new police officers graduated in July and have taken up duties in the city area.  Entertainment / Recreation The John Street basketball courts have been re-surfaced ! The new Central Community Building opened in May. 5,000 new fiction books were bought for the Silver City Library. |

Figure 3.8.1 Sample Output of the Race Dataset

Figure 3.8.1 shows that the RACE high school dataset was successfully downloaded, extracted, and prepared for use, although several warnings appeared due to deprecated TensorFlow attributes. These warnings do not affect the dataset loading but indicate that some functions in the TensorFlow Datasets library use outdated syntax. After the dataset was processed, a sample article titled “Answering the Community Needs of Our City” was printed, demonstrating that the dataset contains informative reading passages typically used for reading comprehension tasks.

|  |
| --- |
| # Import functions and classes from Keras library  from keras.layers import TextVectorization  # Define vectorization parameters  vocab\_size = 10000  seq\_len = 50  # Define the text tokenizer layer and update  vectorizer = TextVectorization(  max\_tokens=vocab\_size - 1,  output\_mode="int",  output\_sequence\_length=seq\_len + 1,  )  vectorizer.adapt(race\_ds)  vocab = vectorizer.get\_vocabulary() |

Figure 3.9 Code for Word Level Vectorization

Figure 3.9 prepares the RACE dataset for training by converting text into numerical sequences that a neural network can process. It first defines a TextVectorization layer that tokenizes text into integer sequences, limits the vocabulary to 10,000 words, and fixes the sequence length to 51 tokens. The preprocessing function ds\_preprocess takes each text sample, tokenizes it, and then splits the sequence into input tokens (x) and target tokens (y) that are offset by one position, allowing the model to learn to predict the next word in a sequence; the dataset is then mapped through this function and optimized with parallel processing and prefetching for efficient training.

|  |
| --- |
| # Import functions and classes from Keras library  from keras import Model, Input  # Define the model parameters  embed\_dim = 256  num\_heads = 2  feed\_forward\_dim  = 256  # Create the GPT model. Save it to gpt\_model variable  # The input shape is the same as the maximum sequence length  # Save the input layer to gpt\_input variable  # Save the TransformerBlock layer output to gpt\_layer variable  # The last Dense layer has 50 output each having the same size as the vocabulary  # Save the last layer output to gpt\_output variable  ### --YOUR CODE HERE-- ###  from keras.layers import Dense  gpt\_input = Input(      shape=(50,),      dtype='int32')  # Save the TransformerBlock layer output to gpt\_layer variable  # First, apply token and position embedding  embedding\_layer = TokenAndPositionEmbedding(      maxlen=50,      vocab\_size=10000,      embed\_dim=embed\_dim      )(gpt\_input)  # Apply TransformerBlock  gpt\_layer = TransformerBlock(      embed\_dim=embed\_dim,      num\_heads=num\_heads,      ff\_dim=feed\_forward\_dim      )(embedding\_layer)  # The last Dense layer has 10000 output units (vocabulary size)  # Save the last layer output to gpt\_output variable  gpt\_output = Dense(      10000,      activation='linear',      name='dense\_2'      )(gpt\_layer)  # Create the GPT model using the layers defined above  gpt\_model = Model(gpt\_input, [gpt\_output, gpt\_layer]) |

Figure 3.10 Code for Building the basic Generative Pre Trained Transformer (GPT)

Figure 3.10 builds a simplified GPT-style model using Keras by defining an input layer, embedding layers, a Transformer block, and a final dense output layer. The input layer, stored in the variable gpt\_input, accepts integer-encoded sequences with a length of 50. The TokenAndPositionEmbedding layer converts these input tokens into dense vector representations while incorporating positional information to help the model understand word order. The embedded data is then passed through a TransformerBlock, which performs multi-head attention and feed-forward operations to capture contextual relationships between words in the sequence. Finally, a dense layer with 10,000 output units, representing the vocabulary size, generates logits for each possible token, allowing the model to predict the next word in a sequence. The complete model, consisting of the input, transformer, and output layers, is compiled into a Keras Model object named gpt\_model.

|  |
| --- |
| # Import functions and classes from Keras library  from keras.losses import SparseCategoricalCrossentropy  # Configure the model for training  gpt\_loss\_fn = SparseCategoricalCrossentropy(from\_logits=True)  gpt\_model.compile(loss=[gpt\_loss\_fn, None], optimizer="adam")  # Train the model. Set the dataset batch size of your choice  # Set the number of epochs to 25  # Assign the output to gpt\_hist variable  ### --YOUR CODE HERE-- ###  gpt\_hist = gpt\_model.fit(      race\_ds.batch(32),      epochs=25  )  import matplotlib.pyplot as plt  # Extract the losses during training  ### --YOUR CODE HERE-- ###  losses = gpt\_hist.history['loss']  # Plot the history of losses  ### --YOUR CODE HERE-- ###  plt.figure(figsize=(8,5))  plt.plot(losses, marker='o')  plt.title("GPT Model Training Loss")  plt.xlabel("Epoch")  plt.ylabel("Loss")  plt.grid(True)  plt.show() |

Figure 3.11 Code for Building and Plotting the GPT Model

Figure 3.11 trains the GPT model using the Sparse Categorical Crossentropy loss function, which is suitable for predicting categorical tokens in a vocabulary. The model is compiled with the Adam optimizer and trained on the preprocessed RACE dataset for 25 epochs using a batch size of 32, with the training progress stored in the variable gpt\_hist. After training, the code extracts and plots the loss values from each epoch using Matplotlib to visualize how the model’s performance improves and stabilizes over time.

A graph with a line graph

AI-generated content may be incorrect.

Figure 3.11.1 Plot of Training loss for GPT Model

Figure 3.11.1 shows the GPT model’s training loss decreasing steadily over 25 epochs, indicating that the model is effectively learning to predict the next token in the sequence. The sharp decline in the early epochs suggests rapid improvement during the initial training phase, followed by a gradual and consistent reduction as the model fine-tunes its parameters. This overall downward trend demonstrates that the model is converging well and becoming more accurate at understanding language patterns in the dataset.

|  |
| --- |
| import numpy as np  # Import functions and classes from Keras library  from keras.activations import softmax  # Set five (5) seed sentences  ### --CHANGE CODE BELOW-- ###  seed\_sentences = [      "The city council announced new plans",      "Technology has changed the way we learn",      "Public transportation is essential",      "Education is the foundation of progress",      "Healthcare workers are vital to society"  ]  # Define the function to sample from model predictions  def sample\_from(logits, k):    logits, indices = ops.top\_k(logits, k, sorted=True)    indices = np.asarray(indices).astype("int32")    preds = softmax(ops.expand\_dims(logits, 0))[0]    preds = np.asarray(preds).astype("float32")    return np.random.choice(indices, p=preds)  # Define the function to generate text  def gpt\_output\_from\_prompt(start\_tokens, max\_tokens=seq\_len, top\_k=10):    start\_tokens = [\_ for \_ in start\_tokens]    num\_tokens\_generated = 0    tokens\_generated = []    while num\_tokens\_generated <= max\_tokens:      pad\_len = seq\_len - len(start\_tokens)      sample\_index = len(start\_tokens) - 1      if pad\_len < 0:        x = start\_tokens[:seq\_len]        sample\_index = seq\_len - 1      elif pad\_len > 0:        x = start\_tokens + [0] \* pad\_len      else:        x = start\_tokens      x = np.array([x])      y, \_ = gpt\_model.predict(x, verbose=0)      sample\_token = sample\_from(y[0][sample\_index], top\_k)      tokens\_generated.append(sample\_token)      start\_tokens.append(sample\_token)      num\_tokens\_generated = len(tokens\_generated)    gen\_text = " ".join(        [vocab[index] for index in start\_tokens + tokens\_generated]    )    return gen\_text  # Assemble word to index dictionary  word\_to\_index = {word: index for index, word in enumerate(vocab)}  # Process each starting sentence into tokens  seed\_tokens = [[word\_to\_index.get(word, 1) for word in sentence.split()]                 for sentence in seed\_sentences]  # Iterate through the seed sentences  # Apply the gpt\_output\_from\_prompt function as defined above  # for each starting sentence  ### --YOUR CODE HERE-- ###  for i, tokens in enumerate(seed\_tokens):      print(f"\n--- Generated text {i+1} ---")      print(gpt\_output\_from\_prompt(tokens, max\_tokens=seq\_len, top\_k=10)) |

Figure 3.12 Code for Obtaining 5 Text Samples

Figure 3.12 uses the trained GPT model to generate new text sequences starting from five predefined seed sentences. Each sentence is first converted into numerical tokens using a vocabulary dictionary, then passed to a function that predicts the next words one token at a time until the maximum sequence length is reached. The sample\_from function applies a top-k sampling strategy that randomly selects the next token from the k most likely predictions according to the model’s softmax probabilities, balancing coherence and creativity. The gpt\_output\_from\_prompt function iteratively feeds the model’s previous outputs back as input, allowing it to generate a continuous stream of text. Finally, the code prints out the generated text for each seed sentence, showing how the model expands on the given prompts.

|  |
| --- |
| --- Generated text 1 ---  [UNK] city council announced new plans to build a fit its [UNK] for the heaviest infrastructure ever two [UNK] ever two or two [UNK] ever [UNK] [UNK] [UNK] near [UNK] from your daily plan to help with your citys population growth attracted new cellphones and other attention in the vast range insight [UNK] insight chinese insight numbers to build a fit its [UNK] for the heaviest infrastructure ever two [UNK] ever two or two [UNK] ever [UNK] [UNK] [UNK] near [UNK] from your daily plan to help with your citys population growth attracted new cellphones and other attention in the vast range insight [UNK] insight chinese insight numbers  --- Generated text 2 ---  [UNK] has changed the way we learn from [UNK] to [UNK] [UNK] can be the first generation to [UNK] [UNK] a [UNK] of [UNK] into the life of his [UNK] the [UNK] [UNK] [UNK] a [UNK] of [UNK] he is [UNK] for an [UNK] in the uk his [UNK] [UNK] he to in [UNK] [UNK] to to [UNK] from [UNK] to [UNK] [UNK] can be the first generation to [UNK] [UNK] a [UNK] of [UNK] into the life of his [UNK] the [UNK] [UNK] [UNK] a [UNK] of [UNK] he is [UNK] for an [UNK] in the uk his [UNK] [UNK] he to in [UNK] [UNK] to to [UNK]  --- Generated text 3 ---  [UNK] transportation is essential to peoples freedom it will be able to share tasks under its easy for you to apply for teaching photography this course is a challenge here is a challenge for people to try new [UNK] and make the cash system for your project is the [UNK] [UNK] [UNK] [UNK] system challenge to peoples freedom it will be able to share tasks under its easy for you to apply for teaching photography this course is a challenge here is a challenge for people to try new [UNK] and make the cash system for your project is the [UNK] [UNK] [UNK] [UNK] system challenge  --- Generated text 4 ---  [UNK] is the foundation of progress in lifelong one of common ways to improve their ability to ease people wish to develop their expectations of the foundation from a teacher will be a teacher who is one of the biggest difficulty in getting better so that the education is important to to to to to to for in lifelong one of common ways to improve their ability to ease people wish to develop their expectations of the foundation from a teacher will be a teacher who is one of the biggest difficulty in getting better so that the education is important to to to to to to for  --- Generated text 5 ---  [UNK] workers are vital to society opportunities for keeping up of golden dreams and sunlight they cannot make ends of the purpose of giving up childhood obesity is to support themselves this is not the problem of it is not just for the elders there are discovering evidence that early childhood childhood exposure childhood civilizations exposure thing opportunities for keeping up of golden dreams and sunlight they cannot make ends of the purpose of giving up childhood obesity is to support themselves this is not the problem of it is not just for the elders there are discovering evidence that early childhood childhood exposure childhood civilizations exposure thing |

Figure 3.12.1 Output Text of Five Generated Text Samples

Figure 3.12.1 shows text generated by the trained GPT model based on the five given seed sentences, demonstrating how the model attempts to extend each prompt using learned language patterns. However, many of the generated sentences include the token “[UNK],” which indicates unknown or out-of-vocabulary words that the tokenizer could not recognize, suggesting that some input words or generated tokens were not part of the model’s vocabulary. While parts of the text show basic grammatical structure and thematic relevance to the prompts, the repetition and incoherence in phrases indicate that the model has not fully learned complex language relationships or context, likely due to limited training data or insufficient training time.

|  |
| --- |
| ### --YOUR CODE HERE-- ###  extra\_hist = gpt\_model.fit(      race\_ds.batch(32),      epochs=15  # try 10–20 more until loss stops improving  )  # Generate outputs again with the same seed sentences  for i, tokens in enumerate(seed\_tokens):      print(f"\n--- New Generated text {i+1} ---")      print(gpt\_output\_from\_prompt(tokens, max\_tokens=seq\_len, top\_k=10)) |

Figure 3.13 Code for Training the Model Further

Figure 3.13 trains the GPT model further by adding more epoch to 15. The output of this further training is shown on Figure 3.13.1. The training output shows that the GPT model continued to improve across 15 epochs, with the loss decreasing steadily from 2.56 to 2.26, indicating better prediction accuracy and stronger learning of text patterns. The newly generated sentences based on the seed prompts are slightly more coherent than before, but still contain repetitive structures, unnatural phrasing, and frequent “[UNK]” tokens, showing that the model has learned basic grammar and topic flow but struggles with vocabulary coverage and long-term context. Overall, the results suggest that the GPT model is improving with training yet still requires more epochs, a larger dataset, or a refined tokenizer to generate fluent and semantically meaningful text.

|  |
| --- |
| Epoch 1/15  **650/650** ━━━━━━━━━━━━━━━━━━━━ **13s** 20ms/step - loss: 2.5623  Epoch 2/15  **650/650** ━━━━━━━━━━━━━━━━━━━━ **13s** 20ms/step - loss: 2.5325  Epoch 3/15  **650/650** ━━━━━━━━━━━━━━━━━━━━ **13s** 20ms/step - loss: 2.5065  Epoch 4/15  **650/650** ━━━━━━━━━━━━━━━━━━━━ **13s** 20ms/step - loss: 2.4829  Epoch 5/15  **650/650** ━━━━━━━━━━━━━━━━━━━━ **13s** 20ms/step - loss: 2.4553  Epoch 6/15  **650/650** ━━━━━━━━━━━━━━━━━━━━ **13s** 19ms/step - loss: 2.4337  Epoch 7/15  **650/650** ━━━━━━━━━━━━━━━━━━━━ **13s** 19ms/step - loss: 2.4115  Epoch 8/15  **650/650** ━━━━━━━━━━━━━━━━━━━━ **13s** 19ms/step - loss: 2.3902  Epoch 9/15  **650/650** ━━━━━━━━━━━━━━━━━━━━ **13s** 20ms/step - loss: 2.3685  Epoch 10/15  **650/650** ━━━━━━━━━━━━━━━━━━━━ **13s** 20ms/step - loss: 2.3512  Epoch 11/15  **650/650** ━━━━━━━━━━━━━━━━━━━━ **13s** 20ms/step - loss: 2.3304  Epoch 12/15  **650/650** ━━━━━━━━━━━━━━━━━━━━ **13s** 20ms/step - loss: 2.3121  Epoch 13/15  **650/650** ━━━━━━━━━━━━━━━━━━━━ **13s** 20ms/step - loss: 2.2944  Epoch 14/15  **650/650** ━━━━━━━━━━━━━━━━━━━━ **13s** 20ms/step - loss: 2.2792  Epoch 15/15  **650/650** ━━━━━━━━━━━━━━━━━━━━ **13s** 20ms/step - loss: 2.2623  --- New Generated text 1 ---  [UNK] city council announced new plans and more people will allow the states to buy food hire the citizens to build your house guests will easily [UNK] from the railways food metropolitan amount of it is it it that are some [UNK] and [UNK] are unable to get this theory of of of of of of of and more people will allow the states to buy food hire the citizens to build your house guests will easily [UNK] from the railways food metropolitan amount of it is it it that are some [UNK] and [UNK] are unable to get this theory of of of of of of of  --- New Generated text 2 ---  [UNK] has changed the way we learn how to deal with nature [UNK] can [UNK] [UNK] [UNK] are [UNK] a [UNK] has been trained to help [UNK] the [UNK] is a [UNK] [UNK] university in ohio he set a public school graduation he can make us feel pretty upset all the of other the the the his the how to deal with nature [UNK] can [UNK] [UNK] [UNK] are [UNK] a [UNK] has been trained to help [UNK] the [UNK] is a [UNK] [UNK] university in ohio he set a public school graduation he can make us feel pretty upset all the of other the the the his the  --- New Generated text 3 ---  [UNK] transportation is essential to help people [UNK] projects the life of wisdom and that make the best expansion it services for students to take the horse to a horse in the [UNK] of a horse he [UNK] [UNK] horse and [UNK] [UNK] never sleeping in recent years [UNK] have become been been been never to help people [UNK] projects the life of wisdom and that make the best expansion it services for students to take the horse to a horse in the [UNK] of a horse he [UNK] [UNK] horse and [UNK] [UNK] never sleeping in recent years [UNK] have become been been been never  --- New Generated text 4 ---  [UNK] is the foundation of progress in our child has developed as we are being [UNK] by offering encouragement to donate again to praise the punishment is a child support system which is the child of child punishment we can improve peoples health tests or their ability to teach their child child child child child child child in our child has developed as we are being [UNK] by offering encouragement to donate again to praise the punishment is a child support system which is the child of child punishment we can improve peoples health tests or their ability to teach their child child child child child child child  --- New Generated text 5 ---  [UNK] workers are vital to society opportunities for a new trend according to [UNK] survey women and support [UNK] in [UNK] the latest statistics shows more likely [UNK] and [UNK] about 120 billion people in slightly less than a year after ten percent of those who suffer from their diets and and and and and those from opportunities for a new trend according to [UNK] survey women and support [UNK] in [UNK] the latest statistics shows more likely [UNK] and [UNK] about 120 billion people in slightly less than a year after ten percent of those who suffer from their diets and and and and and those from |

Figure 3.13.1 Output of Further Training the GPT

### Answer to Guide Questions, Insights, and Reflections

The experiment with the GPT model provided valuable insights into how transformer-based architectures learn language patterns and generate text from training data. Through multiple training epochs, the model’s loss consistently decreased, showing clear improvement in its ability to predict the next token in a sequence, although the generated text still contained repetitive patterns and unknown tokens due to limited vocabulary and training time. This activity emphasized the importance of data quality, model complexity, and sufficient training iterations in achieving coherent and contextually relevant text generation. It also deepened the understanding of how attention mechanisms enable models like GPT to capture dependencies between words more effectively than traditional RNNs. Overall, the process demonstrated both the potential and the challenges of building natural language generation systems, encouraging further exploration into model optimization and dataset expansion for better performance.

## Activity 4: Diffusion-based Generative Models

### Objectives

* Understand how diffusion and denoising processes can be utilized for generating new images
* Build and train a Denoising Diffusion Probabilistic model
* Visualize and qualitatively assess the output of diffusion-based models

### Experimental Procedure

First, the required Python packages, including TensorFlow, Keras, and TensorFlow Datasets, were installed to prepare the environment. The STL-10 dataset, a benchmark dataset for unsupervised and supervised image recognition tasks, was then loaded and visualized by displaying fifteen sample images arranged in a grid format to confirm proper loading and labeling. The images were preprocessed by cropping, resizing to 64×64 pixels, and normalizing pixel values to a range between –1 and 1 to standardize inputs for the neural network.

After data preparation, a custom class called GaussianDiffusion was defined to handle the forward and reverse diffusion processes. This class added random noise to images over a series of timesteps and provided functions for progressively denoising them. Next, several essential network components were implemented, including AttentionBlock and TimeEmbedding layers, along with residual, downsampling, and upsampling blocks that form the U-Net architecture. The model was designed to predict the added noise at each timestep, enabling it to reconstruct the original image from a noisy version through iterative refinement.

Once the network architecture was completed, the DiffusionModel class was implemented to manage training and image generation. This model incorporated both a primary network and an exponential moving average (EMA) network to stabilize learning. The model was compiled using the Adam optimizer and Mean Squared Error (MSE) as the loss function. The network was trained for several epochs with a batch size of 128, and training performance was monitored using a plot of the loss per epoch to visualize convergence. After training, a batch of 25 generated images was produced by reversing the diffusion process, converting pixel values back to the range [0, 1] for visualization.

Finally, the model was trained further until the loss no longer decreased significantly, and another set of 25 images was generated for comparison. The results were analyzed to determine improvements in image realism and quality after additional training. Observations and conclusions were drawn regarding how diffusion-based generative models can effectively produce realistic images through iterative denoising, showcasing the model’s ability to learn complex data distributions.

### Results and Analysis

|  |
| --- |
| # Get the stl-10 dataset. Load all data splits  # https://www.tensorflow.org/datasets/catalog/stl10  # Save the dataset to variable ds\_train  # Save the dataset info to variable ds\_info  ### --YOUR CODE HERE-- ###  # Load the STL-10 dataset  datasets, ds\_info = tfds.load(      "stl10",      split={"dev": "train", "test": "test"},      with\_info=True  )  # Assign the datasets  ds\_train = datasets["dev"]  ds\_test = datasets["test"]  # Display at least fifteen (15) sample images from the dataset, three (3) per row  ### --YOUR CODE HERE-- ###  plt.figure(figsize=(10, 10))  for i, example in enumerate(ds\_train.take(15)):      image = example["image"]      label = example["label"]      plt.subplot(5, 3, i + 1)      plt.imshow(image)      plt.title(ds\_info.features["label"].int2str(label))      plt.axis("off")  plt.tight\_layout()  plt.show() |

Figure 4.1. Displaying the 5 Tested Noisy to Denoised Images

In Figure 4.1 the STL-10 dataset was loaded using TensorFlow Datasets, where the training and testing sets were assigned to the variables ds\_train and ds\_test, respectively. The dataset information was stored in the variable ds\_info to access metadata such as class labels. To verify proper loading, fifteen sample images from the training set were displayed in a 5×3 grid layout using Matplotlib. Each image was shown with its corresponding label, confirming that the dataset was successfully imported and ready for preprocessing and model training. Figure 4.1.1 shows the images and its labels such as bird, airplane, ship, cat, truck, ship, airplane, car, deer, horse and more.

A collage of pictures of different animals

AI-generated content may be incorrect.

Figure 4.1.1 Sample Output Images from the STL-10 Dataset

|  |
| --- |
| # Define preprocessing parameters  img\_size = 64  # Define the preprocessing function  def ds\_preprocess(data):    # Extract image data    img = data["image"]    # Resize image    height = tf.shape(img)[0]    width = tf.shape(img)[1]    crop\_size = tf.minimum(height, width)    img = tf.image.crop\_to\_bounding\_box(img, (height - crop\_size) // 2,                                        (width - crop\_size) // 2,                                        crop\_size, crop\_size)    img = tf.cast(img, dtype=tf.float32)    img = tf.image.resize(img, size=(img\_size, img\_size), antialias=True)    # Rescale and normalize pixel values    img = img / 127.5 - 1.0    img = tf.clip\_by\_value(img, -1.0, 1.0)    return img # Return image only  # Preprocess the dataset. Apply the function as defined above  # Save the preprocessed dataset to same variable name  ### --YOUR CODE HERE-- ###  ds\_train = ds\_train.map(ds\_preprocess, num\_parallel\_calls=tf.data.AUTOTUNE)  ds\_test = ds\_test.map(ds\_preprocess, num\_parallel\_calls=tf.data.AUTOTUNE) |

Figure 4.2. Displaying the 5 Tested Noisy to Denoised Images

Figure 4.2 defines a preprocessing function that prepares images from the STL-10 dataset for training. Each image is cropped to a square shape based on its smallest dimension and then resized to 64×64 pixels to ensure uniform input size. The pixel values are converted to floating-point format and normalized to a range between –1 and 1 for better model performance. Finally, the preprocessing function is applied to both the training and testing datasets using TensorFlow’s parallel data mapping for efficient processing.

|  |
| --- |
| # Import functions and classes from Keras library  from keras import Model  from keras.layers import Layer, Input, Dense, Add, Concatenate  from keras.layers import Conv2D, GroupNormalization, UpSampling2D  from keras.initializers import VarianceScaling  from keras.activations import swish  # Define the model parameters  first\_conv\_channels = 64  channel\_multiplier = [1, 2, 4, 8]  widths = [first\_conv\_channels \* mult for mult in channel\_multiplier]  has\_attention = [False, False, True, True]  num\_res\_blocks = 2  # Number of residual blocks  # Define kernel initializer  # This is to maintain consistent variance of activations  def kernel\_init(scale):    return VarianceScaling(max(scale, 1e-10),                           mode="fan\_avg",                           distribution="uniform")  # Define AttentionBlock as a custom layer  class AttentionBlock(Layer):    """Applies self-attention    Args:        units: Number of units in the dense layers        groups: Number of groups to be used for GroupNormalization layer    """    def \_\_init\_\_(self, units, groups=8, \*\*kwargs):      self.units = units      self.groups = groups      super().\_\_init\_\_(\*\*kwargs)      self.norm = GroupNormalization(groups=groups)      self.query = Dense(units, kernel\_initializer=kernel\_init(1.0))      self.key = Dense(units, kernel\_initializer=kernel\_init(1.0))      self.value = Dense(units, kernel\_initializer=kernel\_init(1.0))      self.proj = Dense(units, kernel\_initializer=kernel\_init(0.0))    def call(self, inputs):      batch\_size = tf.shape(inputs)[0]      height = tf.shape(inputs)[1]      width = tf.shape(inputs)[2]      scale = tf.cast(self.units, tf.float32) \*\* (-0.5)      inputs = self.norm(inputs)      q = self.query(inputs)      k = self.key(inputs)      v = self.value(inputs)      attn\_score = tf.einsum("bhwc, bHWc->bhwHW", q, k) \* scale      attn\_score = tf.reshape(attn\_score,                              [batch\_size, height, width, height \* width])      attn\_score = tf.nn.softmax(attn\_score, -1)      attn\_score = tf.reshape(attn\_score,                              [batch\_size, height, width, height, width])      proj = tf.einsum("bhwHW,bHWc->bhwc", attn\_score, v)      proj = self.proj(proj)      return inputs + proj  # Define TimeEmbedding as a custom layer  class TimeEmbedding(Layer):    def \_\_init\_\_(self, dim, \*\*kwargs):      super().\_\_init\_\_(\*\*kwargs)      self.dim = dim      self.half\_dim = dim // 2      self.emb = np.log(10000) / (self.half\_dim - 1)      self.emb = tf.exp(tf.range(self.half\_dim, dtype=tf.float32) \* -self.emb)    def call(self, inputs):      inputs = tf.cast(inputs, dtype=tf.float32)      emb = inputs[:, None] \* self.emb[None, :]      emb = tf.concat([tf.sin(emb), tf.cos(emb)], axis=-1)      return emb  # Define the residual block  def ResidualBlock(width, groups=8, activation\_fn=swish):    def apply(inputs):      x, t = inputs      input\_width = x.shape[3]      if input\_width == width:          residual = x      else:          residual = Conv2D(width,                            kernel\_size=1,                            kernel\_initializer=kernel\_init(1.0))(x)      temb = activation\_fn(t)      temb = Dense(width, kernel\_initializer=kernel\_init(1.0))(temb)[:, None, None, :]      x = GroupNormalization(groups=groups)(x)      x = activation\_fn(x)      x = Conv2D(width, kernel\_size=3, padding="same",                 kernel\_initializer=kernel\_init(1.0))(x)      x = Add()([x, temb])      x = GroupNormalization(groups=groups)(x)      x = activation\_fn(x)      x = Conv2D(width, kernel\_size=3, padding="same",                 kernel\_initializer=kernel\_init(0.0))(x)      x = Add()([x, residual])      return x    return apply  # Define the downsampling block  def DownSample(width):    def apply(x):      x = Conv2D(width, kernel\_size=3, strides=2, padding="same",                 kernel\_initializer=kernel\_init(1.0))(x)      return x    return apply  # Define the upsampling block  def UpSample(width, interpolation="nearest"):    def apply(x):      x = UpSampling2D(size=2, interpolation=interpolation)(x)      x = Conv2D(width, kernel\_size=3, padding="same",                 kernel\_initializer=kernel\_init(1.0))(x)      return x    return apply  # Define the time projection block  def TimeMLP(units, activation\_fn=swish):    def apply(inputs):      temb = Dense(units, activation=activation\_fn,                   kernel\_initializer=kernel\_init(1.0))(inputs)      temb = Dense(units, kernel\_initializer=kernel\_init(1.0))(temb)      return temb    return apply  # Define the U-net model  def build\_model(img\_size, img\_channels, widths, has\_attention, num\_res\_blocks=2,                  norm\_groups=8, interpolation="nearest", activation\_fn=swish):    image\_input = Input(shape=(img\_size, img\_size, img\_channels),                        name="image\_input")    time\_input = Input(shape=(), dtype=tf.int64, name="time\_input")    x = Conv2D(first\_conv\_channels, kernel\_size=(3, 3), padding="same",               kernel\_initializer=kernel\_init(1.0))(image\_input)    temb = TimeEmbedding(dim=first\_conv\_channels \* 4)(time\_input)    temb = TimeMLP(units=first\_conv\_channels \* 4, activation\_fn=activation\_fn)(temb)    skips = [x]    # DownBlock    for i in range(len(widths)):      for \_ in range(num\_res\_blocks):        x = ResidualBlock(            widths[i], groups=norm\_groups, activation\_fn=activation\_fn)([x, temb])        if has\_attention[i]:            x = AttentionBlock(widths[i], groups=norm\_groups)(x)        skips.append(x)      if widths[i] != widths[-1]:        x = DownSample(widths[i])(x)        skips.append(x)    # MiddleBlock    x = ResidualBlock(widths[-1], groups=norm\_groups,                      activation\_fn=activation\_fn)([x, temb])    x = AttentionBlock(widths[-1], groups=norm\_groups)(x)    x = ResidualBlock(widths[-1], groups=norm\_groups,                      activation\_fn=activation\_fn)([x, temb])    # UpBlock    for i in reversed(range(len(widths))):      for \_ in range(num\_res\_blocks + 1):        x = Concatenate(axis=-1)([x, skips.pop()])        x = ResidualBlock(            widths[i], groups=norm\_groups,            activation\_fn=activation\_fn)([x, temb])        if has\_attention[i]:          x = AttentionBlock(widths[i], groups=norm\_groups)(x)      if i != 0:        x = UpSample(widths[i], interpolation=interpolation)(x)    # End block    x = GroupNormalization(groups=norm\_groups)(x)    x = activation\_fn(x)    x = Conv2D(3, (3, 3), padding="same", kernel\_initializer=kernel\_init(0.0))(x)    return Model([image\_input, time\_input], x, name="unet") |

Figure 4.3. Code for AttentionBlock and TimeEmbedding layers

Figure 4.3 defines a U-Net–based neural network architecture used for diffusion modeling and image generation. It begins by importing necessary Keras components, setting model parameters, and initializing kernels to ensure stable activation variance. Custom layers such as AttentionBlock and TimeEmbedding are implemented to enhance the model’s ability to focus on important image regions and incorporate timestep information. The final build\_model function constructs the U-Net structure by combining residual, downsampling, and upsampling blocks with attention and normalization layers to effectively process and reconstruct images.

|  |
| --- |
| # Import Keras library classes for metrics  from keras.metrics import Mean  # Define the model paramaters  img\_channels = 3  norm\_groups = 8  total\_timesteps = 1000  # Define DiffusionModel as a custom model  class DiffusionModel(Model):  def \_\_init\_\_(self, network, ema\_network, timesteps, gdf\_util, ema=0.999):  super().\_\_init\_\_()  self.network = network  self.ema\_network = ema\_network  self.timesteps = timesteps  self.gdf\_util = gdf\_util  self.ema = ema  self.loss\_tracker = Mean(name="loss")  @property  def metrics(self): # define model metrics  return [self.loss\_tracker]  def train\_step(self, images):  # 1. Get the batch size  batch\_size = tf.shape(images)[0]  # 2. Sample timesteps uniformly  t = tf.random.uniform(minval=0, maxval=self.timesteps, shape=(batch\_size,),  dtype=tf.int64)  with tf.GradientTape() as tape:  # 3. Sample random noise to be added to the images in the batch  noise = tf.random.normal(shape=tf.shape(images), dtype=images.dtype)  # 4. Diffuse the images with noise  images\_t = self.gdf\_util.q\_sample(images, t, noise)  # 5. Pass the diffused images and time steps to the network  pred\_noise = self.network([images\_t, t], training=True)  # 6. Calculate the loss  loss = self.loss(noise, pred\_noise)  # 7. Get the gradients  gradients = tape.gradient(loss, self.network.trainable\_weights)  # 8. Update the weights of the network  self.optimizer.apply\_gradients(zip(gradients, self.network.trainable\_weights))  # 9. Updates the weight values for the network with EMA weights  for weight, ema\_weight in zip(self.network.weights, self.ema\_network.weights):  ema\_weight.assign(self.ema \* ema\_weight + (1 - self.ema) \* weight)  # 10. Update loss track and return loss values  self.loss\_tracker.update\_state(loss)  return {m.name: m.result() for m in self.metrics}  # Call this method to generate batch of images  def generate\_images(self, num\_images=16):  # 1. Randomly sample noise (starting point for reverse process)  samples = tf.random.normal(  shape=(num\_images, img\_size, img\_size, img\_channels), dtype=tf.float32  )  # 2. Sample from the model iteratively  for t in reversed(range(0, self.timesteps)):  tt = tf.cast(tf.fill(num\_images, t), dtype=tf.int64)  pred\_noise = self.ema\_network.predict(  [samples, tt], verbose=0, batch\_size=num\_images  )  samples = self.gdf\_util.p\_sample(  pred\_noise, samples, tt, clip\_denoised=True  )  # 3. Return generated samples  return samples  # Build the U-Net model  network = build\_model(img\_size=img\_size, img\_channels=img\_channels, widths=widths,  has\_attention=has\_attention, num\_res\_blocks=num\_res\_blocks,  norm\_groups=norm\_groups, activation\_fn=swish)  # Build a copy of the U-Net model  # This network weights will be updated using exponential moving average  ema\_network = build\_model(img\_size=img\_size, img\_channels=img\_channels, widths=widths,  has\_attention=has\_attention, num\_res\_blocks=num\_res\_blocks,  norm\_groups=norm\_groups, activation\_fn=swish)  ema\_network.set\_weights(network.get\_weights())  # Get an instance of the Gaussian Diffusion utilities  gdf\_util = GaussianDiffusion(timesteps=total\_timesteps)  # Instantiate the diffusion model. Save it to ddpm variable  # The first two parameters should be assigned to the instantiated networks above  # The third parameter should be assigned with an instance of GaussianDiffusion class  # The fourth parameter should be assigned with the value of total\_timesteps variable  ### --YOUR CODE HERE-- ###  ddpm = DiffusionModel(network, ema\_network, total\_timesteps, gdf\_util)  # Compile the model before showing summary (optional but good practice)  ddpm.compile(optimizer=tf.keras.optimizers.Adam(learning\_rate=1e-4),  loss=tf.keras.losses.MeanSquaredError())  # Show model summary. Take note of the number of trainable parameters  ### --YOUR CODE HERE-- ###  network.summary() |

Figure 4.4 Code for Diffusion model with Model Summary

Figure 4.4 defines and builds a diffusion model using a custom Keras class called DiffusionModel that simulates the forward and reverse diffusion processes for image generation. It trains a U-Net-based neural network to predict noise added to images at different timesteps and uses an exponential moving average of weights to improve stability during training. Finally, the model is compiled with the Adam optimizer and mean squared error loss, and a summary of the U-Net architecture is displayed to show its trainable parameters.

Based on the network summary, the U-Net model consists of multiple interconnected components designed for image generation and denoising tasks. It includes two input layers, one for the image and one for the time embedding, followed by dense layers for embedding and scaling the timestep information. The encoder path is composed of multiple convolutional (Conv2D) layers with SiLU activations, group normalization, and residual skip connections using Add layers. Attention blocks are incorporated at several stages to enhance feature representation and focus on important spatial regions. The decoder path mirrors the encoder structure, using UpSampling2D layers to progressively reconstruct the spatial resolution and concatenating skip connections from earlier layers to preserve spatial details. Throughout the network, GroupNormalization and SiLU layers maintain training stability and non-linearity, while Dense and GetItem layers manage timestep conditioning. The final convolutional layer (Conv2D) outputs a three-channel image, completing the diffusion-based U-Net architecture with approximately 63 million trainable parameters.

|  |
| --- |
| # Import Keras library classes for loss and optimizer  from keras.losses import MeanSquaredError  from keras.optimizers import Adam  from keras.callbacks import EarlyStopping  # Configure the network for training  # Use the MeanSquaredError loss function  # Use Adam optimizer with learning rate of 0.0002  ### --YOUR CODE HERE-- ###  ddpm.compile(  optimizer=Adam(learning\_rate=0.0002),  loss=MeanSquaredError()  )  # Train the model with at least 10 epochs  # Set the dataset batch size of your choice  # Assign the output to ddpm\_hist variable  ### --YOUR CODE HERE-- ###  early\_stop = EarlyStopping(  monitor="loss",  min\_delta=1e-4,  patience=5,  restore\_best\_weights=True  )  ddpm\_hist = ddpm.fit(  ds\_train.batch(128),  epochs=100, #10  # batch\_size=128 #32  callbacks=[early\_stop]  ) |

Figure 4.5 Code for Training the U-Net model

Figure 4.5 compiles and trains the diffusion model using the Adam optimizer with a learning rate of 0.0002 and the Mean Squared Error loss function to measure how well the model predicts noise. An early stopping callback is added to monitor the training loss and stop training automatically if the loss does not improve after five consecutive epochs, helping prevent overfitting. Finally, the model is trained on the training dataset for up to 100 epochs with a batch size of 128, and the training history is stored in the variable ddpm\_hist.

|  |
| --- |
| # Extract the losses during training  ### --YOUR CODE HERE-- ###  losses = ddpm\_hist.history["loss"]  # Plot the history of losses  ### --YOUR CODE HERE-- ###  plt.figure(figsize=(8, 5))  plt.plot(losses, marker='o')  plt.title("Training Loss per Epoch")  plt.xlabel("Epoch")  plt.ylabel("Loss")  plt.grid(True)  plt.show() |

Figure 4.6 Code for Training Loss per Epoch of U-Net

Figure 4.6 shows the code for plotting the U-Net’s training loss per epoch and its output can be seen on Figure 4.6.1. The training loss curve shows a steep decline during the first few epochs, indicating that the model quickly learned to minimize prediction errors. After around the fifth epoch, the loss stabilizes at a very low value, suggesting that the model has effectively converged and is no longer making significant improvements.

A graph with a line graph

AI-generated content may be incorrect.

Figure 4.6.1 Plot for Training Loss per Epoch of U-Net

|  |
| --- |
| # Generate images from the model and show at least 25 of them. Display 5 images per row  # Use the generate\_images() method of the model to get a batch of generated images  # Remember that the output images of the model is normalized. You need to reverse  # the normalization process to show the images properly.  ### --YOUR CODE HERE-- ###  # Generate images from the model  generated\_images = ddpm.generate\_images(num\_images=25)  # Reverse normalization: from [-1, 1] → [0, 1]  generated\_images = (generated\_images + 1.0) / 2.0  generated\_images = np.clip(generated\_images, 0.0, 1.0)  # Plot 25 generated images (5 per row)  plt.figure(figsize=(10, 10))  for i in range(25):      plt.subplot(5, 5, i + 1)      plt.imshow(generated\_images[i])      plt.axis("off")  plt.tight\_layout()  plt.show() |

Figure 4.7. Code for Getting 25 Generated Images from the Model

Figure 4.7 uses the trained diffusion model’s generate\_images method to produce 25 synthetic images based on the learned data distribution. The generated outputs are initially normalized between -1 and 1, so the code reverses this normalization by scaling the values to the 0–1 range for proper visualization. Finally, it displays the 25 generated images in a 5-by-5 grid using Matplotlib, turning off the axes to clearly show the generated results.

A grid of squares with different colored dots

AI-generated content may be incorrect.

Figure 4.7.1 Generated Images from the Model

Figure 4.7.1 shows 25 generated images that appear as random colorful noise rather than meaningful visual patterns. This indicates that the diffusion model has not yet learned to properly reverse the noise process or capture the structure of the training data. The model may need more training epochs, better hyperparameter tuning, or improved data preprocessing to produce recognizable images.

|  |
| --- |
| # Train the model further  ### --YOUR CODE HERE-- ###  ddpm\_hist2 = ddpm.fit(      ds\_train.batch(128),      epochs=30, #10      # batch\_size=32 #32      initial\_epoch=len(ddpm\_hist.history['loss'])  )  # Obtain a new set of generated images  ### --YOUR CODE HERE-- ###  # Obtain a new set of generated images (e.g., 25)  new\_generated\_images = ddpm.generate\_images(num\_images=25)  # Reverse normalization: [-1, 1] → [0, 1]  import numpy as np  import matplotlib.pyplot as plt  new\_generated\_images = (new\_generated\_images + 1.0) / 2.0  new\_generated\_images = np.clip(new\_generated\_images, 0.0, 1.0)  # Display the new generated images (5x5 grid)  plt.figure(figsize=(10, 10))  for i in range(25):      plt.subplot(5, 5, i + 1)      plt.imshow(new\_generated\_images[i])      plt.axis("off")  plt.tight\_layout()  plt.show() |

Figure 4.8. Code for Further Training the Model

Figure 4.8 continues training the diffusion model for an additional 30 epochs, starting from the previous training progress to further improve its learning and image generation capability. After retraining, it generates a new batch of 25 images using the model’s generate\_images function to evaluate visual improvements. The images are then denormalized from the range of -1 to 1 into 0 to 1 and displayed in a 5-by-5 grid for visualization using Matplotlib.

A grid of squares with different colored squares

AI-generated content may be incorrect.

Figure 4.8.1 Output from the Further Training the Model

Figure 4.8.1 shows 25 newly generated images that still appear as random noise rather than coherent or structured visuals. This suggests that even after additional training, the diffusion model has not yet learned the underlying data distribution well enough to reconstruct meaningful images. The model may require more training epochs, better-tuned hyperparameters, or a larger and higher-quality dataset to improve the generation results.

### Answer to Guide Questions, Insights, and Reflections

Activity 4 demonstrates the implementation of a Denoising Diffusion Probabilistic Model which progressively learns to generate images by reversing a noise diffusion process. Throughout the experiment the model was trained using the STL-10 dataset where the training loss consistently decreased indicating that the network successfully learned to denoise images step by step. However the generated outputs still appeared mostly as random noise which suggests that the model requires longer training higher computational resources or better hyperparameter tuning to produce realistic images. This highlights how diffusion models mimic the gradual process of image formation and reflects the balance between theory and computational limitations in deep learning.

## Activity 5: Familiarization with Single Board Computer (SBC)

### Objectives

* Familiarize with Raspberry Pi's software and hardware capabilities
* Program the Raspberry Pi's general purposes I/Os (GPIOs)
* Integrate the Raspberry Pi with a camera to capture real-time video streams

### Experimental Procedure

The experimental procedure involves first setting up the Raspberry Pi's GPIO in BCM mode, configuring Pin 17 as an input with an internal pull-up resistor to monitor the state of a connected push button circuit. Concurrently, the USB camera is initialized using OpenCV's cv2.VideoCapture(0) to begin the video stream. The program then enters a continuous loop where it repeatedly reads the latest frame from the camera and displays the live feed in a window labeled "Camera." Within this loop, the system continuously checks the state of the button pin: if the GPIO input is registered as LOW, which signifies the button has been pressed, a still image is captured from the current frame. Prior to saving, a helper function determines a unique, sequential filename such as frame\_1.jpg, frame\_2.jpg, and so on, to prevent overwriting any previously saved images. The captured image is then written to the file system under this new filename. Immediately after saving, the program enters a momentary waiting state, looping until the button is released to effectively debounce the switch and prevent multiple captures from a single press. The entire capture process continues until the ESC key is pressed on the keyboard, which breaks the main loop, at which point the program releases the camera resource, destroys all display windows, and cleans up the GPIO settings.

### Results and Analysis

|  |
| --- |
|  |

Figure 5.1. Command Line Interface Introduction

As seen on Figure 5.1, ls confirms the presence of the necessary files, while df -h and free -h verify that the system has adequate disk storage to save captured images and sufficient RAM for real-time video processing using OpenCV. Finally, ip a displays the network interface status and IP address, ensuring the device is connected for any necessary remote monitoring or access, and top provides a dynamic baseline of CPU and memory load against which the performance of the running script can later be measured.

|  |
| --- |
|  |

Figure 5.2. Raspberry Pi 4b GPIO

Figure 5.2 shows that the Raspberry Pi 4 Model B features a standard 40-pin GPIO (General Purpose Input/Output) header that serves as the primary physical interface for connecting the board to external components like sensors, LEDs, and switches. This header includes a mix of power pins (3.3V, 5V, and multiple Grounds), along with 26 controllable data pins that can be programmed as either digital inputs or outputs. These pins support various low-level communication protocols such as I2C, SPI, and UART, making the Raspberry Pi a versatile platform for hardware interfacing and electronic projects. The use of the GPIO header is fundamental to activities involving physical computing, such as the push-button image capture exercise.

|  |
| --- |
|  |

Figure 5.3. Connection of the Blinking LED

Figure 5.3 shows that the Blinking LED is connected to the GPIO pin 4 with a 220 ohm resistor and connected to the ground pin for the negative side of the LED. Likewise, Figure 5.4 shows how the code works. The GPIO mode is set to BCM, meaning the pin numbering follows the Broadcom chip’s GPIO scheme, and pin 4 is configured as an output pin. Inside an infinite loop, the program alternately turns the LED on and off every half second using GPIO.output and time.sleep, and when the user stops the program with a keyboard interrupt, GPIO.cleanup() is called to safely reset the GPIO settings.

|  |
| --- |
| import RPi.GPIO as GPIO  import time  led\_pin = 4  GPIO.setmode(GPIO.BCM)  GPIO.setup(led\_pin, GPIO.OUT)  try:  while True:  GPIO.output(led\_pin, GPIO.HIGH)  time.sleep(0.5)  GPIO.output(led\_pin, GPIO.LOW)  time.sleep(0.5)  except KeyboardInterrupt:  GPIO.cleanup() |

Figure 5.4. Code for Blinking LED

|  |
| --- |
|  |

Figure 5.5. Connection for Blinking LED with a button

Figure 5.5 shows the connection of a button to the SBC at GPIO 27. Figure 5.6 controls an LED using a push button connected to a Raspberry Pi. It uses the RPi.GPIO library to access the GPIO pins and the time module to add small delays. The code sets GPIO pin 4 as an output for the LED and pin 27 as an input for the button, then continuously checks the button’s state inside an infinite loop. When the button is pressed, the input signal reads as LOW, turning the LED on, and when it is not pressed, the LED turns off. The main difference from the previous code is that this program introduces user interaction by using a push button to control the LED’s behavior instead of making it blink automatically.

|  |
| --- |
| import RPi.GPIO as GPIO   import time    led\_pin = 4   button\_pin = 27    GPIO.setmode(GPIO.BCM)   GPIO.setup(led\_pin, GPIO.OUT)   GPIO.setup(button\_pin, GPIO.IN)    try:     while True:       button\_state = GPIO.input(button\_pin)       if button\_state == GPIO.LOW:         GPIO.output(led\_pin, GPIO.HIGH)       else:         GPIO.output(led\_pin, GPIO.LOW)       time.sleep(0.02)   except KeyboardInterrupt:     GPIO.cleanup() |

Figure 5.6. Code for Blinking LED with a button

|  |
| --- |
| # -- YOUR CODE HERE -- #  import RPi.GPIO as GPIO  import time  led\_pins = [4, 18, 27, 22]  # led\_pins =[17, 18, 27 22]  button\_pin = 23  GPIO.setmode(GPIO.BCM)  for pin in led\_pins:  GPIO.setup(pin, GPIO.OUT)  GPIO.output(pin, GPIO.LOW)  GPIO.setup(button\_pin, GPIO.IN, pull\_up\_down=GPIO.PUD\_UP)  direction = 1  current\_led = 0  last\_button\_state = GPIO.input(button\_pin )  try:  while True:  button\_state = GPIO.input(button\_pin)  if last\_button\_state == GPIO.HIGH and button\_state == GPIO.LOW:  direction = -direction  time.sleep(0.2)  last\_button\_state = button\_state  for pin in led\_pins:  GPIO.output(pin, GPIO.LOW)  GPIO.output(led\_pins[current\_led ], GPIO.HIGH)  current\_led += direction  if current\_led >= len(led\_pins):  current\_led = 0  elif current\_led < 0:  current\_led = len(led\_pins) -1  time.sleep(0.2)  except KeyboardInterrupt:  GPIO.cleanup() |

Figure 5.7. Code for led\_pattern.py

Figure 5.7 creates a moving LED light pattern using four LEDs and a push button connected to a Raspberry Pi. The RPi.GPIO library is used to control the GPIO pins, where each LED pin is set as an output and the button pin as an input with an internal pull-up resistor to ensure stable readings. The program continuously cycles through the LEDs, lighting one at a time to create a shifting pattern, with the variable direction determining whether the lights move forward or backward. When the button is pressed, the program detects the change in button state and reverses the direction of the LED movement, while the time.sleep() function adds a short delay to make the transitions visible and smooth.

|  |
| --- |
|  |

Figure 5.8. USB Camera Name

Figure 5.8 shows a Raspberry Pi terminal where the command v4l2-ctl --list-devices was executed to list all video capture devices connected to the system. The output confirms that multiple video devices are recognized, including the Raspberry Pi’s built-in camera interfaces such as bcm2835-codec, bcm2835-isp, and rpivid, which are internal video processing components. At the bottom, it shows the connected USB 2.0 HD 1080P PC Camera, which is the external USB camera detected by the system, with device paths /dev/video0 and /dev/media4. This indicates that the USB camera was successfully recognized and can be accessed using those device paths for image or video capture through OpenCV or other camera utilities.

|  |
| --- |
|  |

Figure 5.9. CLI List of Image Resolution

Figure 5.9 shows the output of the command v4l2-ctl --list-formats-ext, which is used to display all the supported video formats and resolutions of the camera connected to the Raspberry Pi. The result indicates that the camera supports multiple formats, including MJPG (Motion-JPEG) and YUYV (YUV 4:2:2), both of which are common for video capture. Under each format, several resolutions such as 1920x1080 (Full HD), 1280x720 (HD), 1024x576, and lower resolutions are listed, along with their frame rates, typically 30.000 fps. This confirms that the camera can record or stream video at various quality levels and frame rates, allowing users to choose the best configuration depending on their application requirements, such as balancing video quality and processing speed.

|  |
| --- |
|  |

Figure 5.10. Sample Picture with USB Camera

Figure 5.10 shows the rpi\_setup.jpg file when the command fswebcam -r $RESOLUTION --no-banner ~/Pictures/rpi\_setup.jpg is ran in the CLI.

|  |
| --- |
|  |

Figure 5.11. Thonny Python IDE

Figure 5.11 shows that the OpenCV is installed properly through executing the shown shell commands import cv cv2.\_\_version\_\_.

|  |
| --- |
| import cv2  cam = cv2.VideoCapture(0) # you can change this based on the assigned number in Step C.2  while True:  ret, img = cam.read()  cv2.imshow("Camera", img)  if cv2.waitKey(1) == 27:  break  cam.release()  cv2.destroyAllWindows() |

Figure 5.12. Code for open\_cv\_cam.py

Figure 5.12 uses the OpenCV library to access and display a live video feed from a camera connected to the Raspberry Pi. The command cv2.VideoCapture(0) initializes the default camera device, where the number 0 represents the first detected camera. Inside the continuous loop, cam.read() captures each video frame, and cv2.imshow() displays it in a window titled “Camera.” The loop runs continuously until the Esc key (key code 27) is pressed, at which point the program releases the camera resource and closes all OpenCV windows using cam.release() and cv2.destroyAllWindows().

|  |
| --- |
| # -- YOUR CODE HERE -- #  import cv2  import time  # Open the webcam  cam = cv2.VideoCapture(0)  # Set the resolution of the webcam capture to 640x480  cam.set(cv2.CAP\_PROP\_FRAME\_WIDTH, 640)  cam.set(cv2.CAP\_PROP\_FRAME\_HEIGHT, 480)  # Define video codec and create VideoWriter object  fourcc = cv2.VideoWriter\_fourcc(\*'XVID')  fps = 60.0  frame\_width = 640 # Set frame width to 640  frame\_height = 480 # Set frame height to 480  out = cv2.VideoWriter('output.mp4', fourcc, fps, (frame\_width, frame\_height))  # Get the start time for video capture  start\_time = time.time()  duration = 20 # Set video duration to 5 seconds  while True:  ret, img = cam.read()  if not ret:  break  out.write(img) # Write frame to video file  cv2.imshow("Camera", img) # Display frame in window  # Check if the 5 seconds have passed  if time.time() - start\_time > duration:  break  # Allow user to exit by pressing 'Esc'  if cv2.waitKey(1) == 27:  break  # Release resources  cam.release()  out.release()  cv2.destroyAllWindows() |

Figure 5.13. Code for Camera Stream as Video

Figure 5.13 records a short video using the Raspberry Pi’s connected webcam and saves it as an MP4 file. It begins by importing the cv2 library for video processing and time for tracking the recording duration. The webcam is initialized using cv2.VideoCapture(0), and its resolution is set to 640×480 pixels. The VideoWriter object is then configured with the XVID codec to save the video as “output.mp4” at 60 frames per second. Inside the loop, each frame is captured from the camera, displayed in a window titled “Camera,” and simultaneously written to the output file. The recording continues until 20 seconds have passed or the user presses the Esc key, after which the program releases all resources and closes the OpenCV windows.

|  |
| --- |
| # -- YOUR CODE HERE -- #  import cv2  import RPi.GPIO as GPIO  import os  BUTTON\_PIN = 17  GPIO.setmode(GPIO.BCM)  GPIO.setup(BUTTON\_PIN, GPIO.IN, pull\_up\_down= GPIO.PUD\_UP)  cam = cv2.VideoCapture(0)  def get\_next\_filename():      i=1      while os.path.exists(f'frame\_{i}.jpg'):          i+= 1      return f'frame\_{i}.jpg'  while True:      ret, img = cam.read()      if not ret:          break      cv2.imshow("Camera", img)      if GPIO.input(BUTTON\_PIN) == GPIO.LOW:                 filename = get\_next\_filename()                 cv2.imwrite(filename, img)                 while GPIO.input(BUTTON\_PIN) == GPIO.LOW:                     pass      if cv2.waitKey(1) == 27:                 break  cam.release()  cv2.destroyAllWindows()  GPIO.cleanup() |

Figure 5.14. Code for Save Frame with Button to GPIO

Figure 5.14 captures and saves individual camera frames whenever a physical button connected to the Raspberry Pi is pressed. The RPi.GPIO library is used to configure pin 17 as an input with an internal pull-up resistor to detect button presses reliably, while OpenCV handles video capture from the connected camera. The function get\_next\_filename() automatically generates sequential filenames (e.g., frame\_1.jpg, frame\_2.jpg) to prevent overwriting previously saved images. During execution, the camera feed is displayed in a window, and when the button is pressed, the current frame is captured and saved as a JPEG file. The program continuously checks for input until the Esc key is pressed, after which it safely releases the camera, closes all OpenCV windows, and cleans up the GPIO configuration.

### Answer to Guide Questions, Insights, and Reflections

The LED flickers when the button is not pressed because the input pin is left floating, meaning it can randomly detect electrical noise as either HIGH or LOW. By adding the argument pull\_up\_down=GPIO.PUD\_UP, an internal pull-up resistor is activated, which keeps the input pin at a stable HIGH state when the button is not pressed. When the button is pressed, the circuit connects the pin to ground, changing the state to LOW in a controlled manner. This modification ensures that the input signal is stable and prevents false readings, effectively eliminating the flickering issue.

In activity 4, I learned how to use the Raspberry Pi as a single-board computer for controlling hardware components and capturing real-time images and videos. I gained hands-on experience with GPIO programming, understanding how digital input and output signals work, and how to eliminate issues such as signal noise using internal pull-up resistors. I also became familiar with integrating OpenCV for camera operations, allowing the Raspberry Pi to capture and save images or record videos automatically. Overall, this experiment enhanced my understanding of how hardware and software interact in embedded systems and how these principles can be applied to real-world computer vision and automation projects.

## Activity 6: <Title>

### Objectives

<…>

### Experimental Procedure

<…>

### Results and Analysis

<…>

### Answer to Guide Questions, Insights, and Reflections

<…>

## <Add more activity sections as needed>

|  |
| --- |
| ### --YOUR CODE HERE-- ###  import your mom |

Figure 1.15. Displaying the 5 Tested Noisy to Denoised Images

Figures and Tables

A graph with a line

Description automatically generated

Figure x. Training loss for the autoencoder…

Explain and provide analysis of each plot or model illustration…

Table x. Model accuracy comparison…

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |

Explain and provide analysis of each table provided…

>

# CONCLUSION AND OVERALL UNDERSTANDING <Replace the guide text with your own conclusions. Approx. 1-2 pages>

* < Summarize your findings on how the fundamental concepts and theory apply to the results of the activity conducted.
* Mention the strengths and weaknesses of applying the concepts and theories in laboratory activities based on your actual model implementation experience.
* Discuss the most significant learning moments and challenges you faced in applying the concepts and how you overcame them.
* Briefly mention how the knowledge gained in these modules will contribute to your understanding of the advanced topics in the future.
* Provide a concluding statement summarizing your overall understanding and your recommendations.>

# REFERENCES

* Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning*. MIT Press.

**RUBRIC FOR SUMMATIVE LABORATORY REPORT ASSESSMENT:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **CRITERIA** | **EXEMPLARY**  **90-100** | **SATISFACTORY**  **80-89** | **DEVELOPING**  **70-79** | **BEGINNING**  **Below 70** | **WEIGHT** |
| Coverage of Topics | All modules are thoroughly addressed with clear explanations and relevant examples from lab work. | Most modules are addressed adequately with explanations and some examples from lab work. | Some modules are addressed superficially or key concepts are missing. Examples from lab work are limited or unclear. | Many modules are missing or poorly addressed. Understanding of key concepts is not demonstrated. Few or no examples from lab work are provided. | 20% |
| Experiment Procedure ***(SO-PI: B1)*** | Laboratory procedure is logically correct including the method used for evaluating results. | Most of the Laboratory procedure, including the method used for evaluating results, are logically correct. | Only around half of the laboratory procedure, including the method used for evaluating results, is logically correct. | Less than half of the laboratory procedure, including the method used for evaluating results, is logically correct. | 20% |
| Experimental Data  ***(SO-PI: B1)*** | Relevant code snippets from lab work are provided for most key concepts. Data gathered is complete, correct and well documented. Tables and graphs are used as required in the procedure. | Relevant code snippets from lab work are provided for most key concepts. Most of the data expected by the experiment are well documented and correct. Missing one required table or graph. | Code snippets are limited, poorly commented, or not always clearly relevant to the concepts being explained. Around 50% of the data are correct and well documented. Missing two to three required tables or graphs. | Few or no relevant code snippets are provided. The data gathered is severely lacking and many of those gathered are questionable. All the required tables and graphs are missing. | 20% |
| Analysis of Experimental Data  ***(SO-PI: B2)*** | All of the data, including errors were analyzed and interpreted correctly using appropriate theories. Clear and extensive evidence of using appropriate probabilistic, statistical, and data analysis techniques. | Most of the data were analyzed and interpreted correctly using appropriate theories. Clear evidence of using appropriate probabilistic, statistical, and data analysis techniques. | Only around 50% of the data were analyzed and interpreted. Some evidence of using appropriate probabilistic, statistical, and data analysis techniques. | Less than half of the data were analyzed and interpreted. Little or no evidence of appropriate probabilistic, statistical, and data analysis techniques. | 20% |
| Concrete understanding   - writing the conclusion  ***(SO-PI: B2)*** | Summarizes the experiment/activity, cites data or output, cites the causes of error, and suggests a recommendation. Conclusion answers all the objectives. Conclusion made is in agreement with the data gathered. | Demonstrates a good overall understanding of most concepts covered and provides some reflection on the learning experience and challenges faced. Conclusion generally answers most of the objectives. Conclusion is stated on the basis of data gathered. | Conclusion is brief and missing significant pieces of information. Conclusion answers only some of the objectives. Conclusion made has some relation with the data gathered. | Demonstrates a limited understanding of the concepts covered and provides little to no meaningful reflection on the learning experience. There is no connection between the conclusion and the objectives and/or data gathered. | 20% |
|  |  |  |  | TOTAL: | 100% |