PRESIDENCY UNIVERSITY

GAIN MORE KNOWLEDGE
REACH GREATER HEIGHTS

Private University Estd. in Karnataka State by Act No. 41 of 2013

PRESIDENCY GROUP OVER 40 YEARS OF ACADEMIC WISDOM

BANGALORE

GENERATIVE AI

COURSE CODE: CSE3191

REPORT

GROUP-3

TEAM MEMBERS:

INDLA MADHANKUMAR – 20201CEI0136

PERAM MAHENDRA REDDY – 20201CEI0112

MASKANI NAVEEN YADAV – 20201CEI0025

KATIPALLY YASHWANTH REDDY – 20201CEI0039

CLASS: 8COM-1

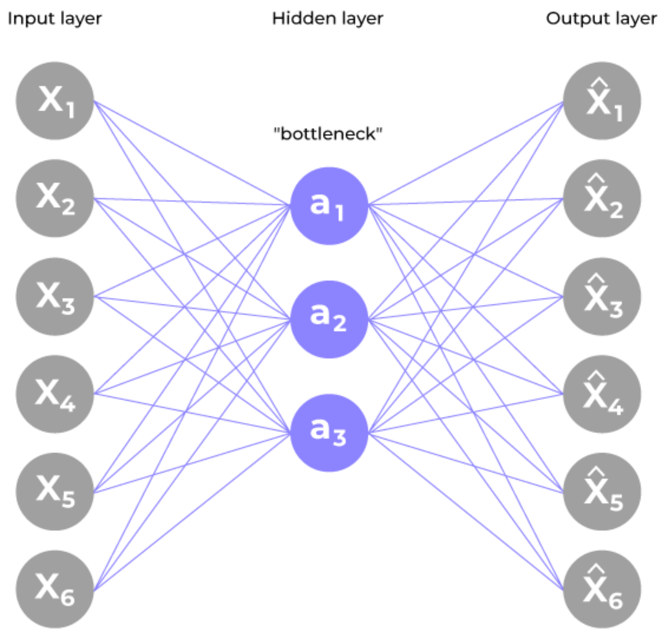
BRANCH: COMPUTER ENGINEERING

AUTOENCODERS

Autoencoders are a specialized class of algorithms that can learn efficient representations of input data with no need for labels. It is a class of artificial neural networks designed for unsupervised learning. Learning to compress and effectively represent input data without specific labels is the essential principle of an automatic decoder. This is accomplished using a two-fold structure that consists of an encoder and a decoder. The encoder transforms the input data into a reduced-dimensional representation, which is often referred to as “latent space” or “encoding”. From that representation, a decoder rebuilds the initial input. For the network to gain meaningful patterns in data, a process of encoding and decoding facilitates the definition of essential features.

Architecture of Autoencoder:

The general architecture of an autoencoder includes an encoder, decoder, and bottleneck layer.



Applications:

- Anomaly Detection
- Dimensionality Reduction
- Denoising
- Image or Audio Compression etc.

Describe the process of implementing an anomaly detection algorithm utilizing AutoEncoders specifically tailored for the analysis of ECG dataset

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl
import tensorflow as tf
from tensorflow.keras.models import Model
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler, StandardScaler
mpl.rcParams['figure.figsize'] = (10, 5)
mpl.rcParams['axes.grid'] = False
```

```
!cat "ECG5000_TRAIN.txt" "ECG5000_TEST.txt" > ecg_final.txt
```

```
df = pd.read_csv("ecg_final.txt", sep='  ', header=None)
df.shape
```

```
<ipython-input-3-4e29b172af0b>:1: ParserWarning: Falling back to the 'python' engine because the 'c' engine does not support regex s
  df = pd.read_csv("ecg_final.txt", sep='  ', header=None)
(5000, 141)
```

```
df.head()
```

	0	1	2	3	4	5	6	7	
0	1.0	-0.112522	-2.827204	-3.773897	-4.349751	-4.376041	-3.474986	-2.181408	-1.8182
1	1.0	-1.100878	-3.996840	-4.285843	-4.506579	-4.022377	-3.234368	-1.566126	-0.9922
2	1.0	-0.567088	-2.593450	-3.874230	-4.584095	-4.187449	-3.151462	-1.742940	-1.4906
3	1.0	0.490473	-1.914407	-3.616364	-4.318823	-4.268016	-3.881110	-2.993280	-1.6711
4	1.0	0.800232	-0.874252	-2.384761	-3.973292	-4.338224	-3.802422	-2.534510	-1.7834

5 rows × 141 columns

```
df=df.add_prefix("c")
df["c0"].value_counts()
```

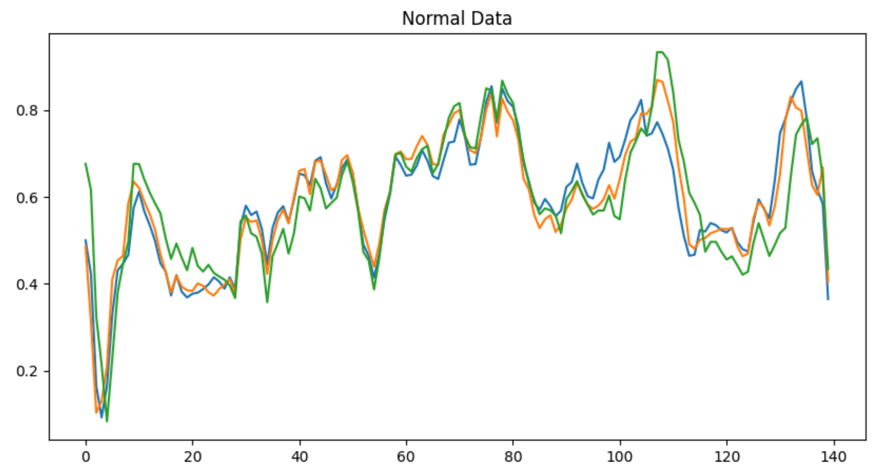
```
1.0    2919
2.0    1767
4.0     194
3.0      96
5.0      24
Name: c0, dtype: int64
```

```
X_train,X_test,y_train,y_test=train_test_split(df.values,df.values[:,0:1],test_size=0.2,random_state=111)
```

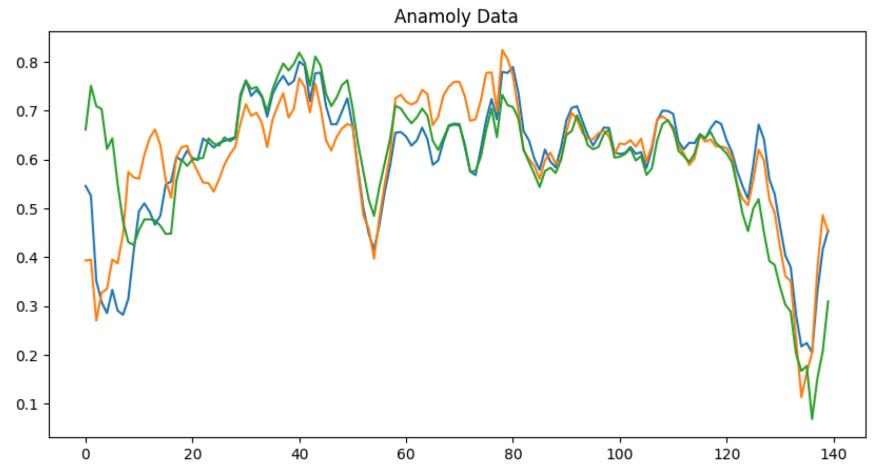
```
scaler=MinMaxScaler()
data_scaled=scaler.fit(X_train)
train_data_scaled=data_scaled.transform(X_train)
test_data_scaled=data_scaled.transform(X_test)
```

```
normal_train_data=pd.DataFrame(train_data_scaled).add_prefix("c").query("c0==0").values[:,1:]
anamoly_train_data=pd.DataFrame(train_data_scaled).add_prefix("c").query("c0>0").values[:,1:]
normal_test_data=pd.DataFrame(test_data_scaled).add_prefix("c").query("c0==0").values[:,1:]
anamoly_test_data=pd.DataFrame(test_data_scaled).add_prefix("c").query("c0>0").values[:,1:]
```

```
plt.plot(normal_train_data[0])
plt.plot(normal_train_data[1])
plt.plot(normal_train_data[2])
plt.title("Normal Data")
plt.show()
```



```
plt.plot(anamoly_train_data[0])
plt.plot(anamoly_train_data[1])
plt.plot(anamoly_train_data[2])
plt.title("Anamoly Data")
plt.show()
```



```
model = tf.keras.Sequential()
model.add(tf.keras.layers.Dense(64, activation="relu"))
model.add(tf.keras.layers.Dense(32, activation="relu"))
model.add(tf.keras.layers.Dense(16, activation="relu"))
model.add(tf.keras.layers.Dense(8, activation="relu"))
model.add(tf.keras.layers.Dense(16, activation="relu"))
model.add(tf.keras.layers.Dense(32, activation="relu"))
model.add(tf.keras.layers.Dense(64, activation="relu"))
model.add(tf.keras.layers.Dense(140, activation="sigmoid"))
```

```
class AutoEncoder(Model):
  def __init__(self):
    super(AutoEncoder,self).__init__()
    self.encoder=tf.keras.Sequential([
        tf.keras.layers.Dense(64,activation="relu"),
        tf.keras.layers.Dense(32,activation="relu"),
        tf.keras.layers.Dense(16,activation="relu"),
        tf.keras.layers.Dense(8,activation="relu")
    ])
    self.decoder=tf.keras.Sequential([
        tf.keras.layers.Dense(16,activation="relu"),
        tf.keras.layers.Dense(32,activation="relu"),
        tf.keras.layers.Dense(64,activation="relu"),
        tf.keras.layers.Dense(140,activation="sigmoid")
    ])
  def call(self,x):
    encoded=self.encoder(x)
    decoded=self.decoder(encoded)
    return decoded
```

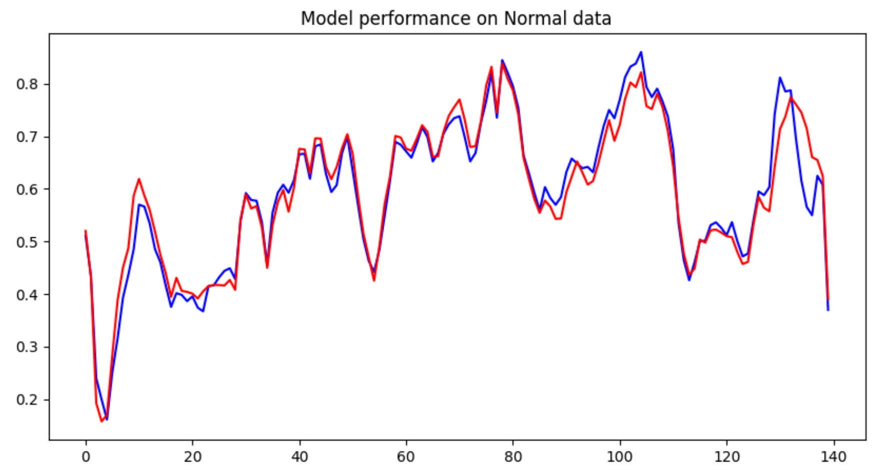
```
model = AutoEncoder()
early_stopping = tf.keras.callbacks.EarlyStopping(monitor="val_loss", patience=2, mode="min")
model.compile(optimizer='adam', loss="mae")
```

```
history=model.fit(normal_train_data, normal_train_data, epochs=50, batch_size=120, validation_data=(train_data_scaled[:,1:], train_data_
```

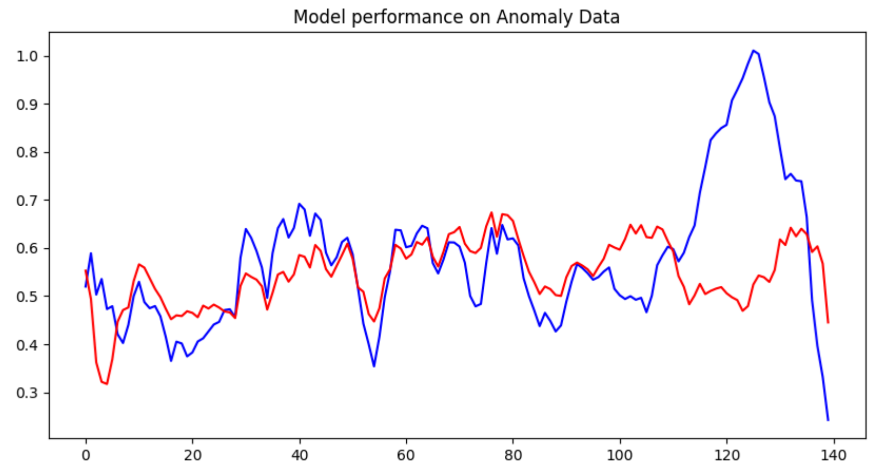
```
Epoch 1/50
20/20 [==============================] - 3s 23ms/step - loss: 0.1194 - val_loss: 0.1005
Epoch 2/50
20/20 [==============================] - 0s 11ms/step - loss: 0.0674 - val_loss: 0.0784
Epoch 3/50
20/20 [==============================] - 0s 13ms/step - loss: 0.0500 - val_loss: 0.0762
Epoch 4/50
20/20 [==============================] - 0s 11ms/step - loss: 0.0481 - val_loss: 0.0754
Epoch 5/50
20/20 [==============================] - 0s 10ms/step - loss: 0.0478 - val_loss: 0.0747
Epoch 6/50
20/20 [==============================] - 0s 13ms/step - loss: 0.0476 - val_loss: 0.0741
Epoch 7/50
20/20 [==============================] - 0s 14ms/step - loss: 0.0474 - val_loss: 0.0738
Epoch 8/50
20/20 [==============================] - 0s 10ms/step - loss: 0.0472 - val_loss: 0.0736
Epoch 9/50
20/20 [==============================] - 0s 13ms/step - loss: 0.0470 - val_loss: 0.0732
Epoch 10/50
20/20 [==============================] - 0s 12ms/step - loss: 0.0470 - val_loss: 0.0732
Epoch 11/50
20/20 [==============================] - 0s 8ms/step - loss: 0.0468 - val_loss: 0.0729
Epoch 12/50
20/20 [==============================] - 0s 7ms/step - loss: 0.0468 - val_loss: 0.0728
Epoch 13/50
20/20 [==============================] - 0s 7ms/step - loss: 0.0466 - val_loss: 0.0715
Epoch 14/50
20/20 [==============================] - 0s 8ms/step - loss: 0.0446 - val_loss: 0.0652
Epoch 15/50
20/20 [==============================] - 0s 14ms/step - loss: 0.0401 - val_loss: 0.0623
Epoch 16/50
20/20 [==============================] - 0s 11ms/step - loss: 0.0374 - val_loss: 0.0619
Epoch 17/50
20/20 [==============================] - 0s 14ms/step - loss: 0.0363 - val_loss: 0.0604
Epoch 18/50
20/20 [==============================] - 0s 13ms/step - loss: 0.0358 - val_loss: 0.0596
Epoch 19/50
20/20 [==============================] - 0s 7ms/step - loss: 0.0354 - val_loss: 0.0593
Epoch 20/50
20/20 [==============================] - 0s 7ms/step - loss: 0.0351 - val_loss: 0.0594
Epoch 21/50
20/20 [==============================] - 0s 8ms/step - loss: 0.0350 - val_loss: 0.0594
```

```
encoder_out = model.encoder(normal_test_data).numpy()
decoder_out = model.decoder(encoder_out).numpy()
```

```
plt.plot(normal_test_data[0], 'b')
plt.plot(decoder_out[0], 'r')
plt.title("Model performance on Normal data")
plt.show()
```

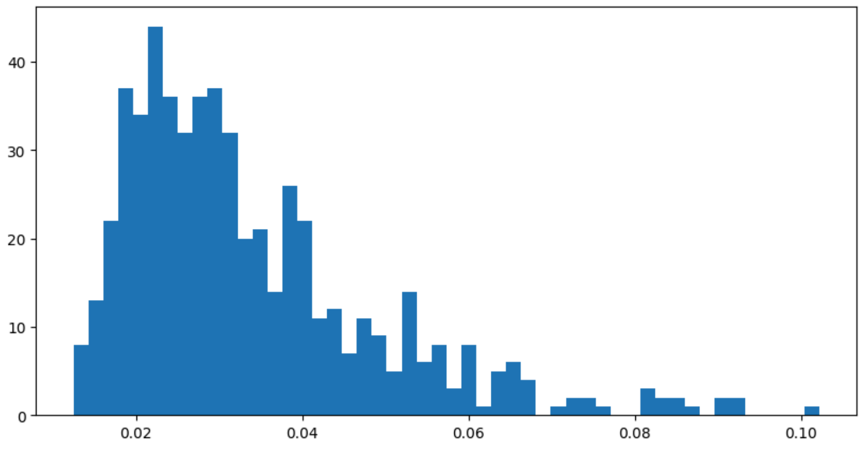


```
encoder_out_a = model.encoder(anamoly_test_data).numpy()
decoder_out_a = model.decoder(encoder_out_a).numpy()
plt.plot(anamoly_test_data[0], 'b')
plt.plot(decoder_out_a[0], 'r')
plt.title("Model performance on Anomaly Data")
plt.show()
```



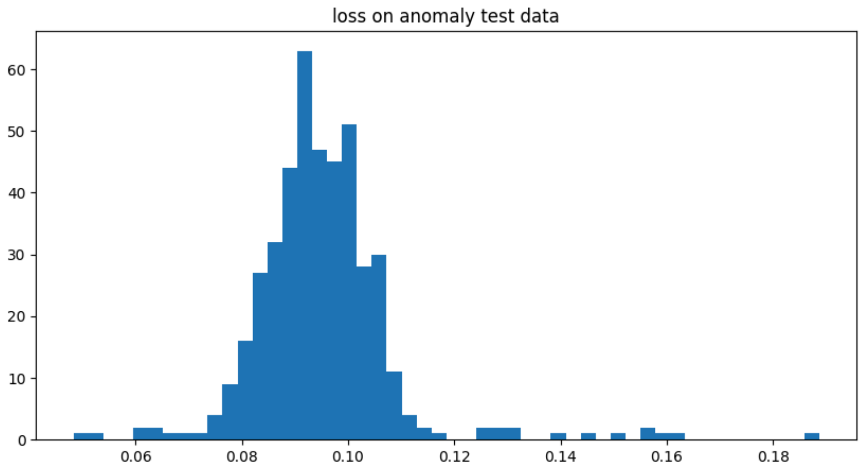
```
reconstruction = model.predict(normal_test_data)
train_loss = tf.keras.losses.mae(reconstruction, normal_test_data)
plt.hist(train_loss, bins=50)
```

```
18/18 [==============================] - 0s 3ms/step
(array([ 8., 13., 22., 37., 34., 44., 36., 32., 36., 37., 32., 20., 21.,
        14., 26., 22., 11., 12.,  7., 11.,  9.,  5., 14.,  6.,  8.,  3.,
         8.,  1.,  5.,  6.,  4.,  0.,  1.,  2.,  2.,  1.,  0.,  0.,  3.,
         2.,  2.,  1.,  0.,  2.,  2.,  0.,  0.,  0.,  0.,  1.]),
 array([0.01243564, 0.01423151, 0.01602739, 0.01782327, 0.01961915,
        0.02141503, 0.02321091, 0.02500679, 0.02680267, 0.02859855,
        0.03039443, 0.03219031, 0.03398619, 0.03578207, 0.03757795,
        0.03937383, 0.04116971, 0.04296559, 0.04476147, 0.04655735,
        0.04835323, 0.05014911, 0.05194499, 0.05374086, 0.05553674,
        0.05733262, 0.0591285 , 0.06092438, 0.06272026, 0.06451614,
        0.06631202, 0.0681079 , 0.06990378, 0.07169966, 0.07349554,
        0.07529142, 0.0770873 , 0.07888318, 0.08067906, 0.08247494,
        0.08427082, 0.0860667 , 0.08786258, 0.08965846, 0.09145434,
        0.09325021, 0.09504609, 0.09684197, 0.09863785, 0.10043373,
        0.10222961]),
 <BarContainer object of 50 artists>)
```

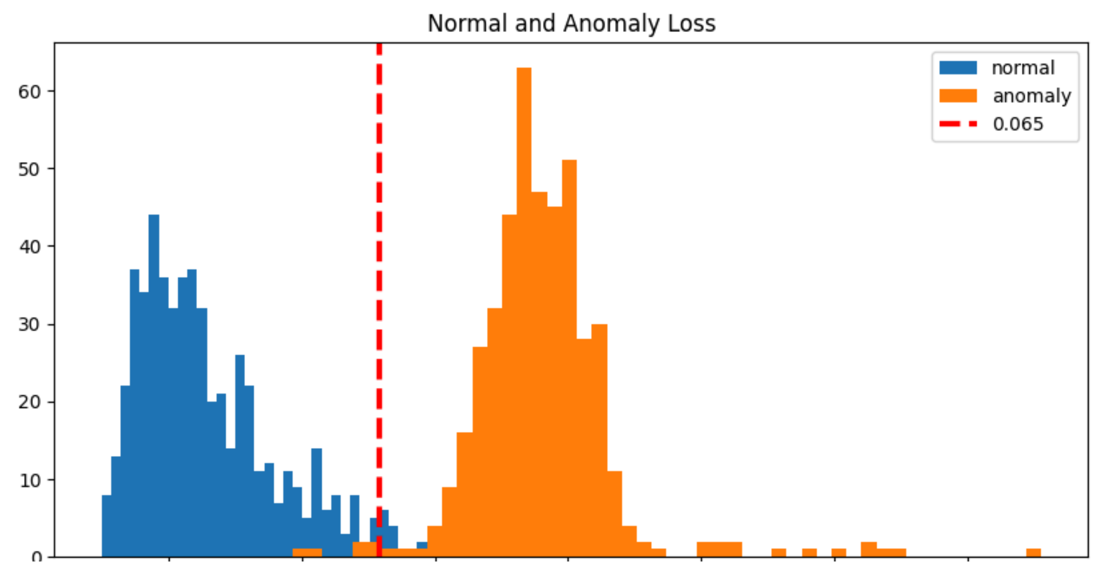


```
threshold = np.mean(train_loss) + 2*np.std(train_loss)
reconstruction_a = model.predict(anamoly_test_data)
train_loss_a = tf.keras.losses.mae(reconstruction_a, anamoly_test_data)
plt.hist(train_loss_a, bins=50)
plt.title("loss on anomaly test data")
plt.show()
```

```
14/14 [==============================] - 0s 2ms/step
```



```
plt.hist(train_loss, bins=50, label='normal')
plt.hist(train_loss_a, bins=50, label='anomaly')
plt.axvline(threshold, color='r', linewidth=3, linestyle='dashed', label='{:0.3f}'.format(threshold))
plt.legend(loc='upper right')
plt.title("Normal and Anomaly Loss")
plt.show()
```



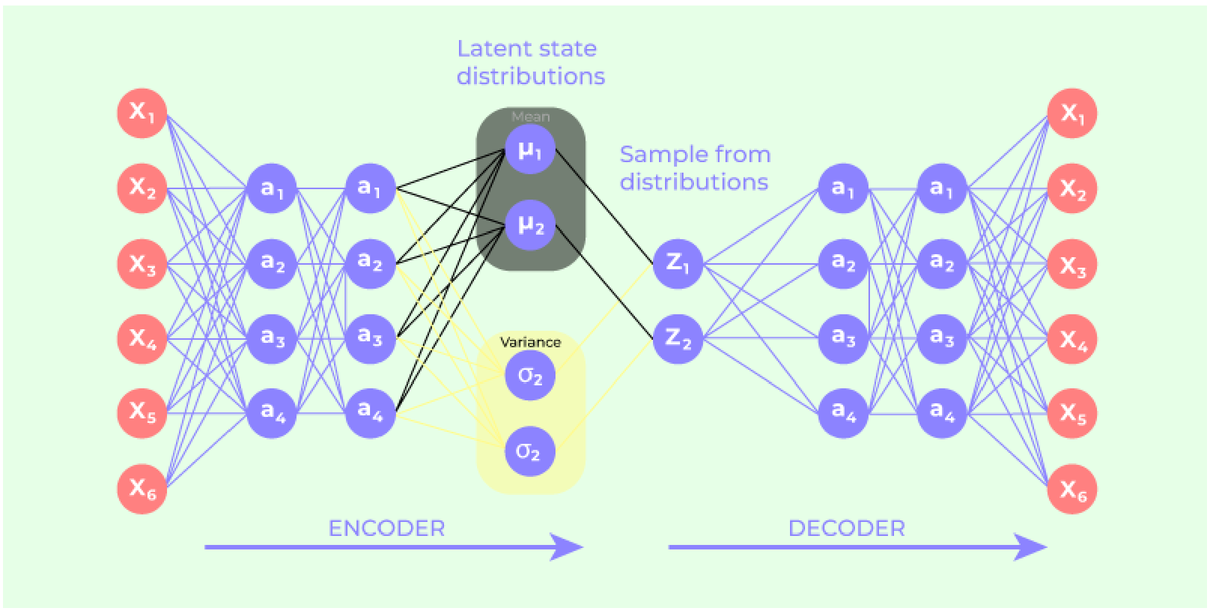
```
preds = tf.math.less(train_loss, threshold)
tf.math.count_nonzero(preds)

     <tf.Tensor: shape=(), dtype=int64, numpy=534>


preds_a = tf.math.greater(train_loss_a, threshold)
tf.math.count_nonzero(preds_a)

     <tf.Tensor: shape=(), dtype=int64, numpy=432>
```

VARIATIONAL AUTOENCODERS

Variational autoencoder was proposed in 2013 by Diederik P. Kingma and Max Welling at Google and Qualcomm. A variational autoencoder (VAE) provides a probabilistic manner for describing an observation in latent space. Thus, rather than building an encoder that outputs a single value to describe each latent state attribute, we'll formulate our encoder to describe a probability distribution for each latent attribute. It has many applications, such as data compression, synthetic data creation, etc.

Variational autoencoder is different from an autoencoder in a way that it provides a statistical manner for describing the samples of the dataset in latent space. Therefore, in the variational autoencoder, the encoder outputs a probability distribution in the bottleneck layer instead of a single output value.

Architecture of Variational Autoencoder:



Variational autoencoder uses KL-divergence as its loss function, the goal of this is to minimize the difference between a supposed distribution and original distribution of dataset.

Applications:

- Anomaly Detection
- Image Generation
- Representation Learning
- Image or Audio Compression etc.

How would you go about implementing Variational AutoEncoders to detect anomalies within the MNIST handwritten digits dataset

```
#1.VAE-Hand Written Digits
import numpy as np
import tensorflow as tf
from tensorflow.keras import layers, Model
from tensorflow.keras.datasets import mnist
import matplotlib.pyplot as plt

# Load and preprocess MNIST dataset
(x_train, _), _ = mnist.load_data()
x_train = x_train.astype('float32') / 255.0
x_train = np.expand_dims(x_train, -1)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 [==============================] - 0s 0us/step
```

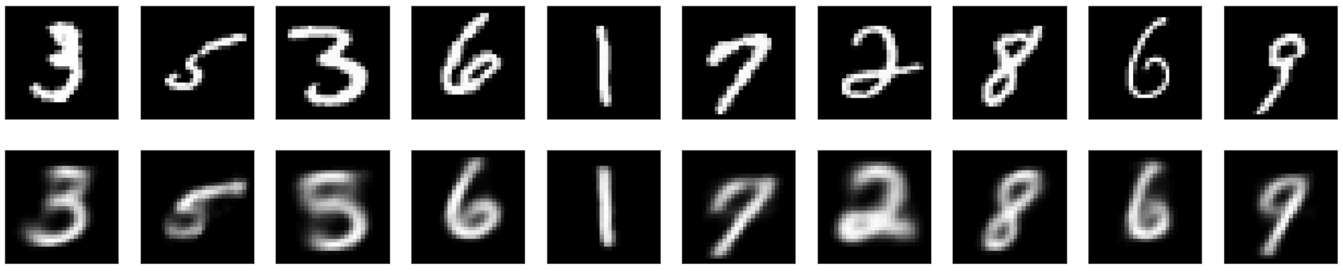
```
# Define the Variational Autoencoder (VAE) model
class VAE(Model):
    def __init__(self, latent_dim):
        super(VAE, self).__init__()
        self.latent_dim = latent_dim

        # Encoder
        self.encoder = tf.keras.Sequential([
            layers.Input(shape=(28, 28, 1)),
            layers.Flatten(),
            layers.Dense(256, activation='relu'),
            layers.Dense(128, activation='relu'),
            layers.Dense(latent_dim * 2)  # Outputting mean and log variance
        ])

        # Decoder
        self.decoder = tf.keras.Sequential([
            layers.Input(shape=(latent_dim,)),
            layers.Dense(128, activation='relu'),
            layers.Dense(256, activation='relu'),
            layers.Dense(28 * 28, activation='sigmoid'),
            layers.Reshape((28, 28, 1))
        ])

    def reparameterize(self, mean, logvar):
        eps = tf.random.normal(shape=tf.shape(mean))
        return eps * tf.exp(logvar * 0.5) + mean

    def call(self, x):
        # Encoding
        latent_params = self.encoder(x)
        mean, logvar = latent_params[:, :self.latent_dim], latent_params[:, self.latent_dim:]
        z = self.reparameterize(mean, logvar)

        # Decoding
        reconstructed_x = self.decoder(z)

        return reconstructed_x, mean, logvar


# Define the loss function for VAE
def vae_loss(recon_x, x, mean, logvar):
    # Reconstruction loss (binary cross entropy)
    reconstruction_loss = tf.keras.losses.binary_crossentropy(x, recon_x)
    reconstruction_loss = tf.reduce_sum(reconstruction_loss, axis=(1, 2))

    # KL divergence loss
    kl_divergence_loss = -0.5 * tf.reduce_sum(1 + logvar - tf.square(mean) - tf.exp(logvar), axis=1)

    return tf.reduce_mean(reconstruction_loss + kl_divergence_loss)
```

```
# Initialize parameters
latent_dim = 2  # Dimension of latent space
lr = 1e-3  # Learning rate
epochs = 20  # Number of training epochs
batch_size = 128

# Create VAE model
vae = VAE(latent_dim)
optimizer = tf.keras.optimizers.Adam(lr)


# Training loop
for epoch in range(epochs):
    total_loss = 0
    num_batches = x_train.shape[0] // batch_size
    for i in range(num_batches):
        start_idx = i * batch_size
        end_idx = (i + 1) * batch_size
        batch_x = x_train[start_idx:end_idx]

        with tf.GradientTape() as tape:
            recon_batch, mean, logvar = vae(batch_x)
            loss = vae_loss(recon_batch, batch_x, mean, logvar)
        gradients = tape.gradient(loss, vae.trainable_variables)
        optimizer.apply_gradients(zip(gradients, vae.trainable_variables))

        total_loss += loss

    print(f"Epoch {epoch+1}, Loss: {total_loss / num_batches}")
```

```
Epoch 1, Loss: 196.12351989746094
Epoch 2, Loss: 170.01319885253906
Epoch 3, Loss: 165.23135375976562
Epoch 4, Loss: 161.5552978515625
Epoch 5, Loss: 158.65994262695312
Epoch 6, Loss: 156.51344299316406
Epoch 7, Loss: 154.7811279296875
Epoch 8, Loss: 153.07965087890625
Epoch 9, Loss: 151.554443359375
Epoch 10, Loss: 150.358642578125
Epoch 11, Loss: 149.4595184326172
Epoch 12, Loss: 148.47784423828125
Epoch 13, Loss: 147.72445678710938
Epoch 14, Loss: 146.98626708984375
Epoch 15, Loss: 146.39459228515625
Epoch 16, Loss: 145.9656982421875
Epoch 17, Loss: 145.46951293945312
Epoch 18, Loss: 144.8653564453125
Epoch 19, Loss: 144.5868377685547
Epoch 20, Loss: 144.12124633789062
```

```
# Visualize reconstructed digits
means=[]
n = 10  # Number of digits to visualize
plt.figure(figsize=(20, 4))
for i in range(n):
    original = np.expand_dims(x_train[i+10], axis=0)
    reconstructed_data, mean, logvar = vae(original)
    means.append(mean)
    loss = vae_loss(reconstructed_data, original, mean, logvar)
    print("Reconstruction loss :", loss.numpy())
    # Original images
    ax = plt.subplot(2, n, i + 1)
    plt.imshow(x_train[i+10].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

    # Reconstructed images
    ax = plt.subplot(2, n, i + 1 + n)
    plt.imshow(reconstructed_data.numpy().reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
plt.show()
```

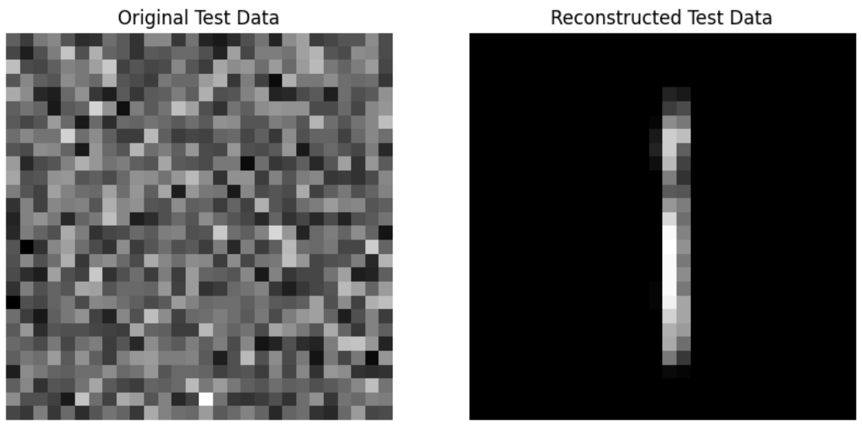
```
Reconstruction loss : 135.99326
Reconstruction loss : 106.716446
Reconstruction loss : 187.46005
Reconstruction loss : 148.9425
Reconstruction loss : 54.061985
Reconstruction loss : 107.838554
Reconstruction loss : 199.43967
Reconstruction loss : 122.752975
Reconstruction loss : 122.21537
Reconstruction loss : 104.15463
```



```
# Anomaly detection
test_data = np.random.randn(1, 28, 28, 1).astype('float32')  # Generate random digit (anomaly)
reconstructed_test_data, mean, logvar = vae(test_data)
loss = vae_loss(reconstructed_test_data, test_data, mean, logvar)
print("Reconstruction loss for anomaly:", loss.numpy())
```

```
Reconstruction loss for anomaly: -317.16595
```

```
# Visualize reconstructed test data
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.title('Original Test Data')
plt.imshow(test_data.squeeze(), cmap='gray')
plt.axis('off')

plt.subplot(1, 2, 2)
plt.title('Reconstructed Test Data')
plt.imshow(reconstructed_test_data.numpy().squeeze(), cmap='gray')
plt.axis('off')
plt.show()
```



```
x_train[0].shape
```

```
(28, 28, 1)
```

```
test_data.shape
```

```
(1, 28, 28, 1)
```

GENERATIVE ADVERSARIAL NETWORK

Generative Adversarial Networks (GANs) are a powerful class of neural networks that are used for an unsupervised learning. GANs are made up of two neural networks, a discriminator and a generator. They use adversarial training to produce artificial data that is identical to actual data.

The Generator attempts to fool the Discriminator, which is tasked with accurately distinguishing between produced and genuine data, by producing random noise samples.

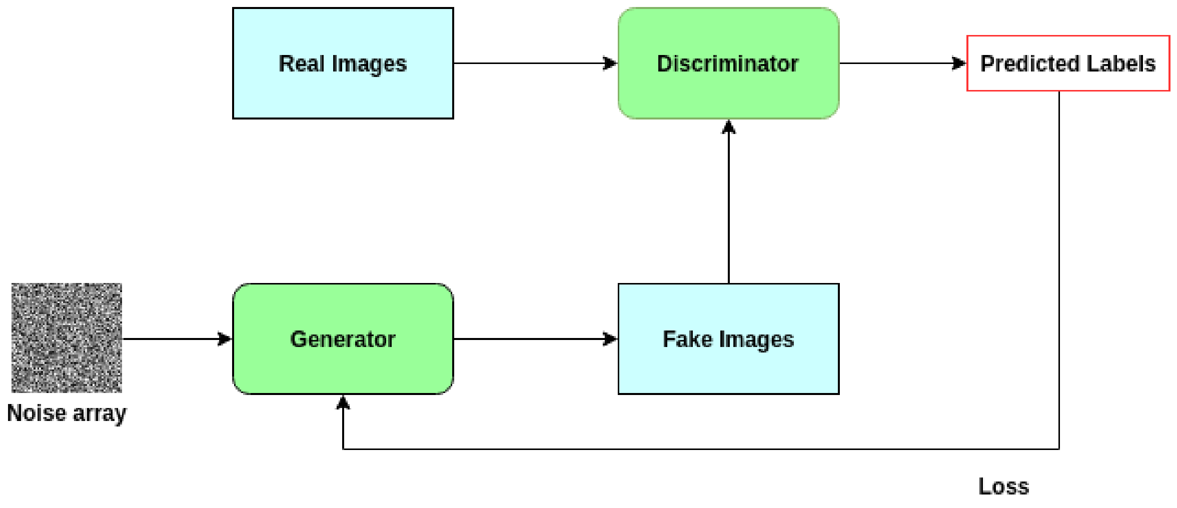
Realistic, high-quality samples are produced as a result of this competitive interaction, which drives both networks toward advancement.

GANs are proving to be highly versatile artificial intelligence tools, as evidenced by their extensive use in image synthesis, style transfer, and text-to-image synthesis.

They have also revolutionized generative modelling.

Through adversarial training, these models engage in a competitive interplay until the generator becomes adept at creating realistic samples, fooling the discriminator approximately half the time.

Architecture of GAN:



Applications:

- Image-Image Translation
- Image Generation
- Text-Image Synthesis
- Anomaly Detection etc.

How would you go about implementing a Generative Adversarial Network (GAN) to generate new images using the MNIST dataset?

```
import tensorflow as tf
```

```
import glob
import imageio
import matplotlib.pyplot as plt
import numpy as np
import os
import PIL
from tensorflow.keras import layers
import time

from IPython import display
```

```
(train_images, train_labels), (_, _) = tf.keras.datasets.mnist.load_data()
```

```
    Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
    11490434/11490434 [==============================] - 0s 0us/step
```

```
train_images = train_images.reshape(train_images.shape[0], 28, 28, 1).astype('float32')
train_images = (train_images - 127.5) / 127.5  # Normalize the images to [-1, 1]
```

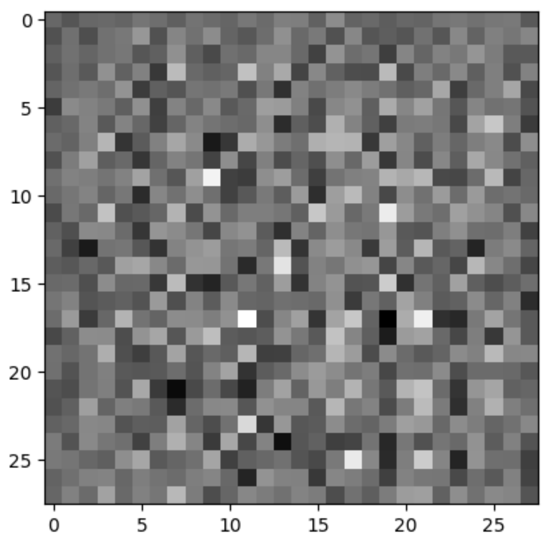
```
BUFFER_SIZE = 60000
BATCH_SIZE = 256
```

```
train_dataset = tf.data.Dataset.from_tensor_slices(train_images).shuffle(BUFFER_SIZE).batch(BATCH_SIZE)
```

```
def make_generator_model():
    model = tf.keras.Sequential()
    model.add(layers.Dense(7*7*256, use_bias=False, input_shape=(100,)))
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Reshape((7, 7, 256)))
    assert model.output_shape == (None, 7, 7, 256)  # Note: None is the batch size

    model.add(layers.Conv2DTranspose(128, (5, 5), strides=(1, 1), padding='same', use_bias=False))
    assert model.output_shape == (None, 7, 7, 128)
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Conv2DTranspose(64, (5, 5), strides=(2, 2), padding='same', use_bias=False))
    assert model.output_shape == (None, 14, 14, 64)
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Conv2DTranspose(1, (5, 5), strides=(2, 2), padding='same', use_bias=False, activation='tanh'))
    assert model.output_shape == (None, 28, 28, 1)

    return model
```

```
generator = make_generator_model()

noise = tf.random.normal([1, 100])
generated_image = generator(noise, training=False)

plt.imshow(generated_image[0, :, :, 0], cmap='gray')
```

```
<matplotlib.image.AxesImage at 0x7f0e3bfe9cc0>
```



```
def make_discriminator_model():
    model = tf.keras.Sequential()
    model.add(layers.Conv2D(64, (5, 5), strides=(2, 2), padding='same',
                                     input_shape=[28, 28, 1]))
    model.add(layers.LeakyReLU())
    model.add(layers.Dropout(0.3))

    model.add(layers.Conv2D(128, (5, 5), strides=(2, 2), padding='same'))
    model.add(layers.LeakyReLU())
    model.add(layers.Dropout(0.3))

    model.add(layers.Flatten())
    model.add(layers.Dense(1))

    return model
```

```
discriminator = make_discriminator_model()
decision = discriminator(generated_image)
print (decision)
```

```
tf.Tensor([[0.00400666]], shape=(1, 1), dtype=float32)
```

```
cross_entropy = tf.keras.losses.BinaryCrossentropy(from_logits=True)
```

```
def discriminator_loss(real_output, fake_output):
    real_loss = cross_entropy(tf.ones_like(real_output), real_output)
    fake_loss = cross_entropy(tf.zeros_like(fake_output), fake_output)
    total_loss = real_loss + fake_loss
    return total_loss
```

```
def generator_loss(fake_output):
    return cross_entropy(tf.ones_like(fake_output), fake_output)
```

```
generator_optimizer = tf.keras.optimizers.Adam(1e-4)
discriminator_optimizer = tf.keras.optimizers.Adam(1e-4)
```

```
checkpoint_dir = './training_checkpoints'
checkpoint_prefix = os.path.join(checkpoint_dir, "ckpt")
checkpoint = tf.train.Checkpoint(generator_optimizer=generator_optimizer,
                                 discriminator_optimizer=discriminator_optimizer,
                                 generator=generator,
                                 discriminator=discriminator)
```

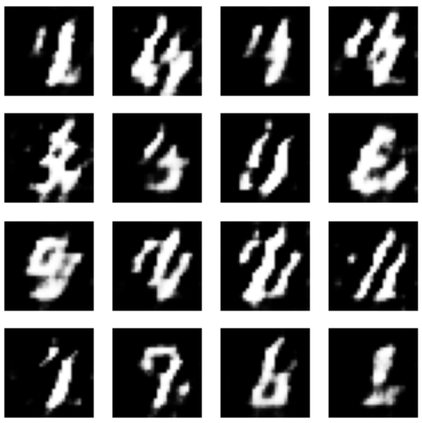
```
EPOCHS = 50
noise_dim = 100
num_examples_to_generate = 16
```

```
seed = tf.random.normal([num_examples_to_generate, noise_dim])
```

```
@tf.function
def train_step(images):
    noise = tf.random.normal([BATCH_SIZE, noise_dim])

    with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
      generated_images = generator(noise, training=True)

      real_output = discriminator(images, training=True)
      fake_output = discriminator(generated_images, training=True)

      gen_loss = generator_loss(fake_output)
      disc_loss = discriminator_loss(real_output, fake_output)

    gradients_of_generator = gen_tape.gradient(gen_loss, generator.trainable_variables)
    gradients_of_discriminator = disc_tape.gradient(disc_loss, discriminator.trainable_variables)

    generator_optimizer.apply_gradients(zip(gradients_of_generator, generator.trainable_variables))
    discriminator_optimizer.apply_gradients(zip(gradients_of_discriminator, discriminator.trainable_variables))


def train(dataset, epochs):
  for epoch in range(epochs):
    start = time.time()

    for image_batch in dataset:
      train_step(image_batch)

    # Produce images for the GIF as you go
    display.clear_output(wait=True)
    generate_and_save_images(generator,
                             epoch + 1,
                             seed)

    # Save the model every 15 epochs
    if (epoch + 1) % 15 == 0:
      checkpoint.save(file_prefix = checkpoint_prefix)

    print ('Time for epoch {} is {} sec'.format(epoch + 1, time.time()-start))

  # Generate after the final epoch
  display.clear_output(wait=True)
  generate_and_save_images(generator,
                           epochs,
                           seed)


def generate_and_save_images(model, epoch, test_input):
  # Notice `training` is set to False.
  # This is so all layers run in inference mode (batchnorm).
  predictions = model(test_input, training=False)

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

  for i in range(predictions.shape[0]):
      plt.subplot(4, 4, i+1)
      plt.imshow(predictions[i, :, :, 0] * 127.5 + 127.5, cmap='gray')
      plt.axis('off')

  plt.savefig('image_at_epoch_{:04d}.png'.format(epoch))
  plt.show()


# To generate GIFs
!pip install imageio
!pip install git+https://github.com/tensorflow/docs
```

```
Requirement already satisfied: imageio in /usr/local/lib/python3.10/dist-packages (2.31.6)
Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-packages (from imageio) (1.25.2)
Requirement already satisfied: pillow<10.1.0,>=8.3.2 in /usr/local/lib/python3.10/dist-packages (from imageio) (9.4.0)
Collecting git+https://github.com/tensorflow/docs
  Cloning https://github.com/tensorflow/docs to /tmp/pip-req-build-dazxwk09
  Running command git clone --filter=blob:none --quiet https://github.com/tensorflow/docs /tmp/pip-req-build-dazxwk09
  Resolved https://github.com/tensorflow/docs to commit 8b36191001b53bfce4fe15b77e243fbd7f382e41
  Preparing metadata (setup.py) ... done
Collecting astor (from tensorflow-docs==2024.2.5.73858)
  Downloading astor-0.8.1-py2.py3-none-any.whl (27 kB)
Requirement already satisfied: absl-py in /usr/local/lib/python3.10/dist-packages (from tensorflow-docs==2024.2.5.73858) (1.4.0)
Requirement already satisfied: jinja2 in /usr/local/lib/python3.10/dist-packages (from tensorflow-docs==2024.2.5.73858) (3.1.3)
Requirement already satisfied: nbformat in /usr/local/lib/python3.10/dist-packages (from tensorflow-docs==2024.2.5.73858) (5.9.2)
Requirement already satisfied: protobuf>=3.12 in /usr/local/lib/python3.10/dist-packages (from tensorflow-docs==2024.2.5.73858) (3.2
Requirement already satisfied: pyyaml in /usr/local/lib/python3.10/dist-packages (from tensorflow-docs==2024.2.5.73858) (6.0.1)
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.10/dist-packages (from jinja2->tensorflow-docs==2024.2.5.7
Requirement already satisfied: fastjsonschema in /usr/local/lib/python3.10/dist-packages (from nbformat->tensorflow-docs==2024.2.5.
Requirement already satisfied: jsonschema>=2.6 in /usr/local/lib/python3.10/dist-packages (from nbformat->tensorflow-docs==2024.2.5
Requirement already satisfied: jupyter-core in /usr/local/lib/python3.10/dist-packages (from nbformat->tensorflow-docs==2024.2.5.73
Requirement already satisfied: traitlets>=5.1 in /usr/local/lib/python3.10/dist-packages (from nbformat->tensorflow-docs==2024.2.5.7
Requirement already satisfied: attrs>=22.2.0 in /usr/local/lib/python3.10/dist-packages (from jsonschema>=2.6->nbformat->tensorflow
Requirement already satisfied: jsonschema-specifications>=2023.03.6 in /usr/local/lib/python3.10/dist-packages (from jsonschema>=2.6
Requirement already satisfied: referencing>=0.28.4 in /usr/local/lib/python3.10/dist-packages (from jsonschema>=2.6->nbformat->tenso
Requirement already satisfied: rpds-py>=0.7.1 in /usr/local/lib/python3.10/dist-packages (from jsonschema>=2.6->nbformat->tensorflow
Requirement already satisfied: platformdirs>=2.5 in /usr/local/lib/python3.10/dist-packages (from jupyter-core->nbformat->tensorflow
Building wheels for collected packages: tensorflow-docs
  Building wheel for tensorflow-docs (setup.py) ... done
  Created wheel for tensorflow-docs: filename=tensorflow_docs-2024.2.5.73858-py3-none-any.whl size=182442 sha256=e35d04579d193d7918
  Stored in directory: /tmp/pip-ephem-wheel-cache-03a0jonl/wheels/86/0f/1e/3b62293c8ffd0fd5a49508e6871cdb7554abe9c62afd35ec53
Successfully built tensorflow-docs
Installing collected packages: astor, tensorflow-docs
Successfully installed astor-0.8.1 tensorflow-docs-2024.2.5.73858
```

```
train(train_dataset, EPOCHS)
```



```
    Time for epoch 8 is 642.8402316570282 sec
```

```
checkpoint.restore(tf.train.latest_checkpoint(checkpoint_dir))
```

```
def display_image(epoch_no):
  return PIL.Image.open('image_at_epoch_{:04d}.png'.format(epoch_no))
```

```
display_image(EPOCHS)
```

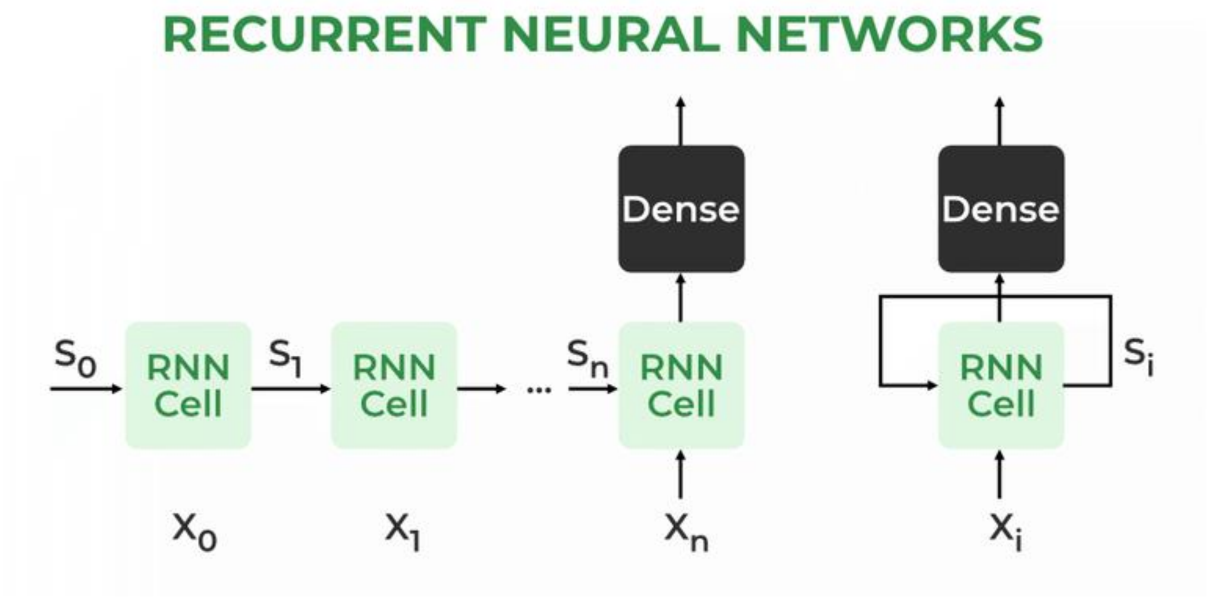
```
anim_file = 'dcgan.gif'

with imageio.get_writer(anim_file, mode='I') as writer:
  filenames = glob.glob('image*.png')
  filenames = sorted(filenames)
  for filename in filenames:
    image = imageio.imread(filename)
    writer.append_data(image)
  image = imageio.imread(filename)
  writer.append_data(image)
```

```
import tensorflow_docs.vis.embed as embed
embed.embed_file(anim_file)
```

RECURRENT NUERAL NETWORK

Recurrent Neural Network (RNN) is a type of Neural Network where the output from the previous step is fed as input to the current step. In traditional neural networks, all the inputs and outputs are independent of each other. Still, in cases when it is required to predict the next word of a sentence, the previous words are required and hence there is a need to remember the previous words. Thus, RNN came into existence, which solved this issue with the help of a Hidden Layer. The main and most important feature of RNN is its Hidden state, which remembers some information about a sequence. The state is also referred to as Memory State since it remembers the previous input to the network. It uses the same parameters for each input as it performs the same task on all the inputs or hidden layers to produce the output. This reduces the complexity of parameters, unlike other neural networks.

Architecture of GAN:



The fundamental processing unit in a Recurrent Neural Network (RNN) is a Recurrent Unit, which is not explicitly called a “Recurrent Neuron.” This unit has the unique ability to maintain a hidden state, allowing the network to capture sequential dependencies by remembering previous inputs while processing.

Applications:

- Sentiment Analysis
- Text Generation
- Machine Translation
- Anomaly Detection etc.

Construct a basic Recurrent Neural Network (RNN) model to forecast stock prices based on historical stock data

```
import yfinance as yf

# Define the ticker symbol of the company
ticker_symbol = 'AAPL'  # Example: Apple Inc.

# Download historical data
data = yf.download(ticker_symbol, start='2010-01-01', end='2022-01-01')

# Save data to a CSV file
data.to_csv(f'{ticker_symbol}_historical_data.csv')
```

```
[*********************100%%**********************]  1 of 1 completed
```

```
import numpy as np
import pandas as pd
import yfinance as yf
from sklearn.preprocessing import MinMaxScaler
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import SimpleRNN, Dense

# Download historical data
def download_historical_data(ticker_symbol, start_date, end_date):
    data = yf.download(ticker_symbol, start=start_date, end=end_date)
    return data

# Preprocess data
def preprocess_data(data):
    scaler = MinMaxScaler(feature_range=(0, 1))
    scaled_data = scaler.fit_transform(data['Close'].values.reshape(-1, 1))
    return scaled_data, scaler

# Create sequences for training
def create_sequences(data, sequence_length):
    X, y = [], []
    for i in range(len(data) - sequence_length):
        X.append(data[i:i+sequence_length])
        y.append(data[i+sequence_length])
    return np.array(X), np.array(y)

# Define model architecture
def build_model(sequence_length):
    model = Sequential([
        SimpleRNN(50, input_shape=(sequence_length, 1)),
        Dense(1)
    ])
    model.compile(optimizer='adam', loss='mean_squared_error')
    return model

# Define parameters
ticker_symbol = 'AAPL'
start_date = '2010-01-01'
end_date = '2022-01-01'
sequence_length = 10  # Number of historical data points to look back

# Download historical data
historical_data = download_historical_data(ticker_symbol, start_date, end_date)

# Preprocess data
scaled_data, scaler = preprocess_data(historical_data)

# Create sequences for training
X, y = create_sequences(scaled_data, sequence_length)

# Split data into training and testing sets
split_ratio = 0.8
split_index = int(split_ratio * len(X))
X_train, X_test = X[:split_index], X[split_index:]
y_train, y_test = y[:split_index], y[split_index:]

# Reshape data for LSTM
X_train = np.reshape(X_train, (X_train.shape[0], X_train.shape[1], 1))
X_test = np.reshape(X_test, (X_test.shape[0], X_test.shape[1], 1))
```

```
# Build and train the model
model = build_model(sequence_length)
model.fit(X_train, y_train, epochs=50, batch_size=32, validation_split=0.2)

# Evaluate the model
loss = model.evaluate(X_test, y_test)
print('Test Loss:', loss)

# Make predictions
predictions = model.predict(X_test)
predictions = scaler.inverse_transform(predictions)

# Compare predictions with actual prices
actual_prices = scaler.inverse_transform(y_test.reshape(-1, 1))
comparison_df = pd.DataFrame({'Actual': actual_prices.flatten(), 'Predicted': predictions.flatten()})
print(comparison_df)
```

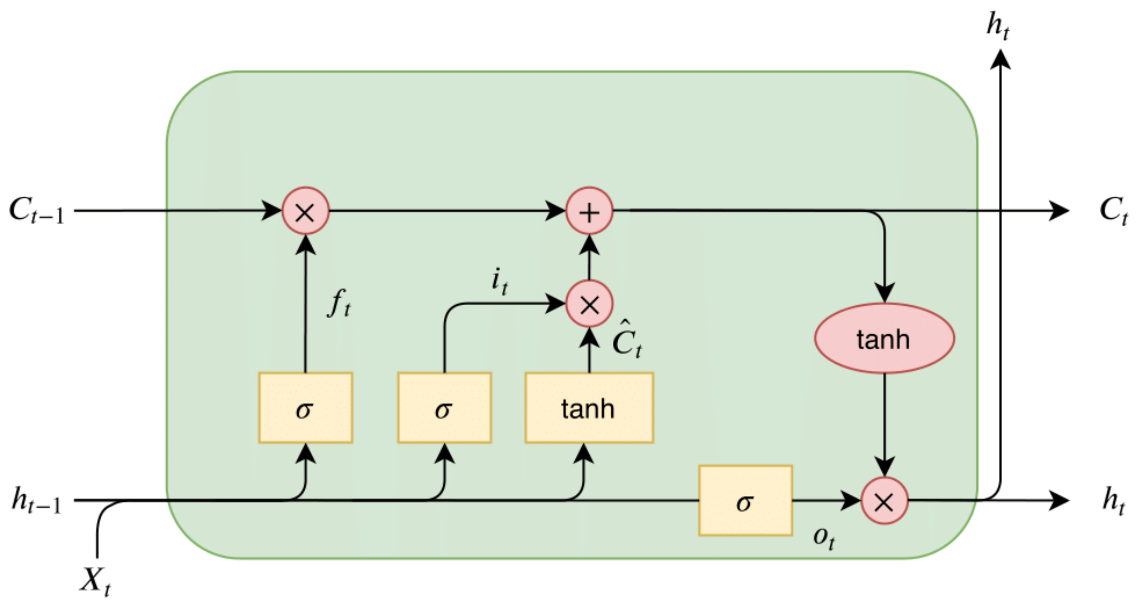
```
[*********************100%%**********************]  1 of 1 completed
Epoch 1/50
61/61 [==============================] - 2s 9ms/step - loss: 0.0012 - val_loss: 3.8546e-04
Epoch 2/50
61/61 [==============================] - 0s 6ms/step - loss: 4.5488e-05 - val_loss: 2.9210e-04
Epoch 3/50
61/61 [==============================] - 0s 6ms/step - loss: 3.8507e-05 - val_loss: 2.3714e-04
Epoch 4/50
61/61 [==============================] - 0s 7ms/step - loss: 3.4193e-05 - val_loss: 2.8078e-04
Epoch 5/50
61/61 [==============================] - 0s 7ms/step - loss: 2.9125e-05 - val_loss: 1.7012e-04
Epoch 6/50
61/61 [==============================] - 0s 7ms/step - loss: 2.3823e-05 - val_loss: 1.5562e-04
Epoch 7/50
61/61 [==============================] - 0s 7ms/step - loss: 2.0021e-05 - val_loss: 1.4795e-04
Epoch 8/50
61/61 [==============================] - 0s 6ms/step - loss: 1.8066e-05 - val_loss: 1.1065e-04
Epoch 9/50
61/61 [==============================] - 0s 6ms/step - loss: 1.5016e-05 - val_loss: 1.0072e-04
Epoch 10/50
61/61 [==============================] - 0s 6ms/step - loss: 1.3168e-05 - val_loss: 7.4579e-05
Epoch 11/50
61/61 [==============================] - 0s 7ms/step - loss: 1.1927e-05 - val_loss: 5.5885e-05
Epoch 12/50
61/61 [==============================] - 0s 8ms/step - loss: 1.3090e-05 - val_loss: 5.1345e-05
Epoch 13/50
61/61 [==============================] - 0s 7ms/step - loss: 1.1240e-05 - val_loss: 4.6895e-05
Epoch 14/50
61/61 [==============================] - 0s 7ms/step - loss: 1.0190e-05 - val_loss: 9.4728e-05
Epoch 15/50
61/61 [==============================] - 0s 7ms/step - loss: 1.0028e-05 - val_loss: 1.2240e-04
Epoch 16/50
61/61 [==============================] - 0s 7ms/step - loss: 9.6813e-06 - val_loss: 7.8087e-05
Epoch 17/50
61/61 [==============================] - 1s 9ms/step - loss: 8.0338e-06 - val_loss: 9.0063e-05
Epoch 18/50
61/61 [==============================] - 1s 9ms/step - loss: 9.0001e-06 - val_loss: 5.5214e-05
Epoch 19/50
61/61 [==============================] - 1s 9ms/step - loss: 8.3345e-06 - val_loss: 7.2945e-05
Epoch 20/50
61/61 [==============================] - 1s 9ms/step - loss: 7.5561e-06 - val_loss: 3.6442e-05
Epoch 21/50
61/61 [==============================] - 1s 9ms/step - loss: 7.9761e-06 - val_loss: 3.2679e-05
Epoch 22/50
61/61 [==============================] - 0s 6ms/step - loss: 6.5877e-06 - val_loss: 3.7000e-05
Epoch 23/50
61/61 [==============================] - 0s 6ms/step - loss: 9.1415e-06 - val_loss: 5.1531e-05
Epoch 24/50
61/61 [==============================] - 0s 6ms/step - loss: 7.1209e-06 - val_loss: 3.0517e-05
Epoch 25/50
61/61 [==============================] - 0s 5ms/step - loss: 7.3215e-06 - val_loss: 3.0134e-05
Epoch 26/50
61/61 [==============================] - 0s 6ms/step - loss: 6.4744e-06 - val_loss: 2.9631e-05
Epoch 27/50
61/61 [==============================] - 0s 6ms/step - loss: 6.3735e-06 - val_loss: 6.6620e-05
Epoch 28/50
61/61 [==============================] - 0s 6ms/step - loss: 6.0909e-06 - val_loss: 3.4219e-05
```

LONG SHORT TERM MEMORY

A traditional RNN has a single hidden state that is passed through time, which can make it difficult for the network to learn long-term dependencies. LSTMs address this problem by introducing a memory cell, which is a container that can hold information for an extended period. LSTM networks are capable of learning long-term dependencies in sequential data, which makes them well-suited for tasks such as language translation, speech recognition, and time series forecasting. LSTMs can also be used in combination with other neural network architectures, such as Convolutional Neural Networks (CNNs) for image and video analysis.

The memory cell is controlled by three gates: the input gate, the forget gate, and the output gate. These gates decide what information to add to, remove from, and output from the memory cell. The input gate controls what information is added to the memory cell. The forget gate controls what information is removed from the memory cell. And the output gate controls what information is output from the memory cell. This allows LSTM networks to selectively retain or discard information as it flows through the network, which allows them to learn long-term dependencies.

Architecture of LSTM:



Applications:

- Language Modelling
- Time Series Forecasting
- Recommender Systems
- Anomaly Detection etc.

Implement Long Short Term Memory For predicting the next sequence of characters(Text Generation) for the given sentance

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense

text = "I am A Student From Presidency University"
chars = sorted(list(set(text)))
char_to_index = {char: i for i, char in enumerate(chars)}
index_to_char = {i: char for i, char in enumerate(chars)}

seq_length = 3
sequences = []
labels = []

for i in range(len(text) - seq_length):
  seq = text[i:i+seq_length]
  label = text[i+seq_length]
  sequences.append([char_to_index[char] for char in seq])
  labels.append(char_to_index[label])

X = np.array(sequences)
y = np.array(labels)

X_one_hot = tf.one_hot(X, len(chars))
y_one_hot = tf.one_hot(y, len(chars))

model = Sequential()
model.add(LSTM(50, input_shape=(seq_length, len(chars)), activation='relu'))
model.add(Dense(len(chars), activation='softmax'))

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

model.fit(X_one_hot, y_one_hot, epochs=100)

start_seq = "I am"
generated_text = start_seq

for i in range(50):
  x = np.array([[char_to_index[char] for char in generated_text[-seq_length:]]])
  x_one_hot = tf.one_hot(x, len(chars))
  prediction = model.predict(x_one_hot)
  next_index = np.argmax(prediction)
  next_char = index_to_char[next_index]
  generated_text += next_char

print("Generated Text:")
print(generated_text)
```

```
1/1 [==============================] - 0s 24ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 21ms/step
1/1 [==============================] - 0s 21ms/step
1/1 [==============================] - 0s 21ms/step
1/1 [==============================] - 0s 21ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 22ms/step
1/1 [==============================] - 0s 21ms/step
1/1 [==============================] - 0s 29ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 20ms/step
1/1 [==============================] - 0s 21ms/step
1/1 [==============================] - 0s 21ms/step
1/1 [==============================] - 0s 22ms/step
1/1 [==============================] - 0s 22ms/step
1/1 [==============================] - 0s 21ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 24ms/step
1/1 [==============================] - 0s 21ms/step
1/1 [==============================] - 0s 21ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 21ms/step
1/1 [==============================] - 0s 24ms/step
1/1 [==============================] - 0s 21ms/step
1/1 [==============================] - 0s 21ms/step
1/1 [==============================] - 0s 22ms/step
1/1 [==============================] - 0s 22ms/step
Generated Text:
I am  rriieent ersiteesiteesiteesiteesiteesiteesiteesi
```

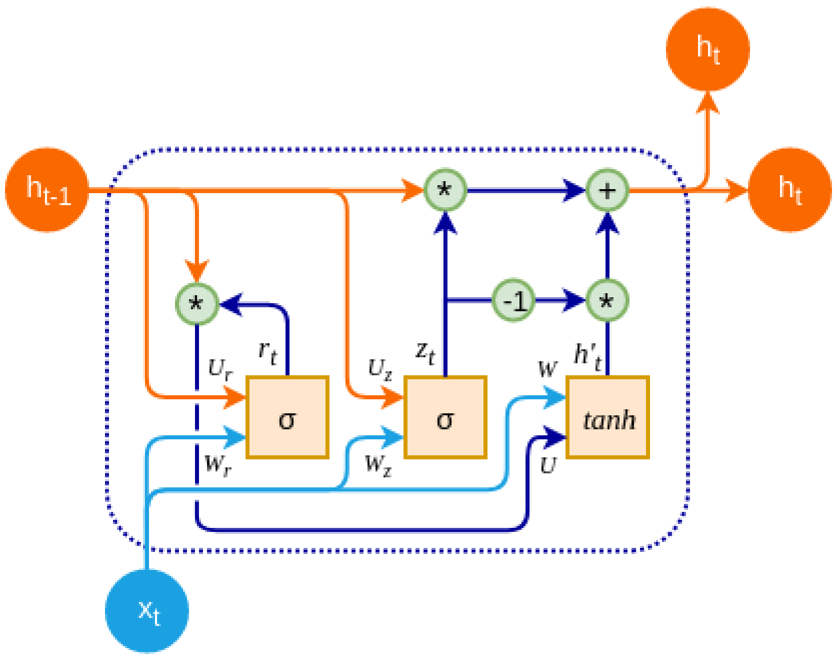
GATED RECURRENT UNIT

Gated Recurrent Unit (GRU) is a type of recurrent neural network (RNN) that was introduced by Cho et al. in 2014 as a simpler alternative to Long Short-Term Memory (LSTM) networks. Like LSTM, GRU can process sequential data such as text, speech, and time-series data.

The basic idea behind GRU is to use gating mechanisms to selectively update the hidden state of the network at each time step. The gating mechanisms are used to control the flow of information in and out of the network. The GRU has two gating mechanisms, called the reset gate and the update gate.

The reset gate determines how much of the previous hidden state should be forgotten, while the update gate determines how much of the new input should be used to update the hidden state. The output of the GRU is calculated based on the updated hidden state.

Architecture of GRU:



Applications:

- Gesture Recognition
- Speech Synthesis
- Stock Market Prediction
- Text Generation etc.

Implement a Stock price prediction model for predicting future stocks using historical data using Gated Recurrent Unit

```
import numpy as np
import pandas as pd
import yfinance as yf
from sklearn.preprocessing import MinMaxScaler
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import GRU, Dense
import matplotlib.pyplot as plt
```

```
ticker_symbol = 'AAPL'
start_date = '2010-01-01'
end_date = '2022-01-01'
```

```
data = yf.download(ticker_symbol, start=start_date, end=end_date)
```

```
[*********************100%%**********************]  1 of 1 completed
```

```
data
```

	Open	High	Low	Close	Adj Close	Volume
Date						
2010-01-04	7.622500	7.660714	7.585000	7.643214	6.470741	493729600
2010-01-05	7.664286	7.699643	7.616071	7.656429	6.481927	601904800
2010-01-06	7.656429	7.686786	7.526786	7.534643	6.378824	552160000
2010-01-07	7.562500	7.571429	7.466071	7.520714	6.367033	477131200
2010-01-08	7.510714	7.571429	7.466429	7.570714	6.409362	447610800
...	...	...	...	...	...	...
2021-12-27	177.089996	180.419998	177.070007	180.330002	178.065674	74919600
2021-12-28	180.160004	181.330002	178.529999	179.289993	177.038696	79144300
2021-12-29	179.330002	180.630005	178.139999	179.380005	177.127594	62348900
2021-12-30	179.470001	180.570007	178.089996	178.199997	175.962402	59773000
2021-12-31	178.089996	179.229996	177.259995	177.570007	175.340302	64062300

3021 rows × 6 columns

```
close_prices = data['Close'].values.reshape(-1, 1)
```

```
scaler = MinMaxScaler(feature_range=(0, 1))
scaled_data = scaler.fit_transform(close_prices)
```

```
def create_dataset(data, time_step):
    X, y = [], []
    for i in range(len(data)-time_step-1):
        X.append(data[i:(i+time_step), 0])
        y.append(data[i + time_step, 0])
    return np.array(X), np.array(y)
```

```
time_step = 30

X, y = create_dataset(scaled_data, time_step)
```

```
train_size = int(len(X) * 0.8)
X_train, X_test = X[:train_size], X[train_size:]
y_train, y_test = y[:train_size], y[train_size:]
```

```
model = Sequential()
model.add(GRU(units=50, return_sequences=True, input_shape=(X_train.shape[1], 1)))
model.add(GRU(units=50))
model.add(Dense(units=1))
model.compile(optimizer='adam', loss='mean_squared_error',metrics='accuracy')
```

```
model.fit(X_train, y_train, epochs=100, batch_size=64)
```

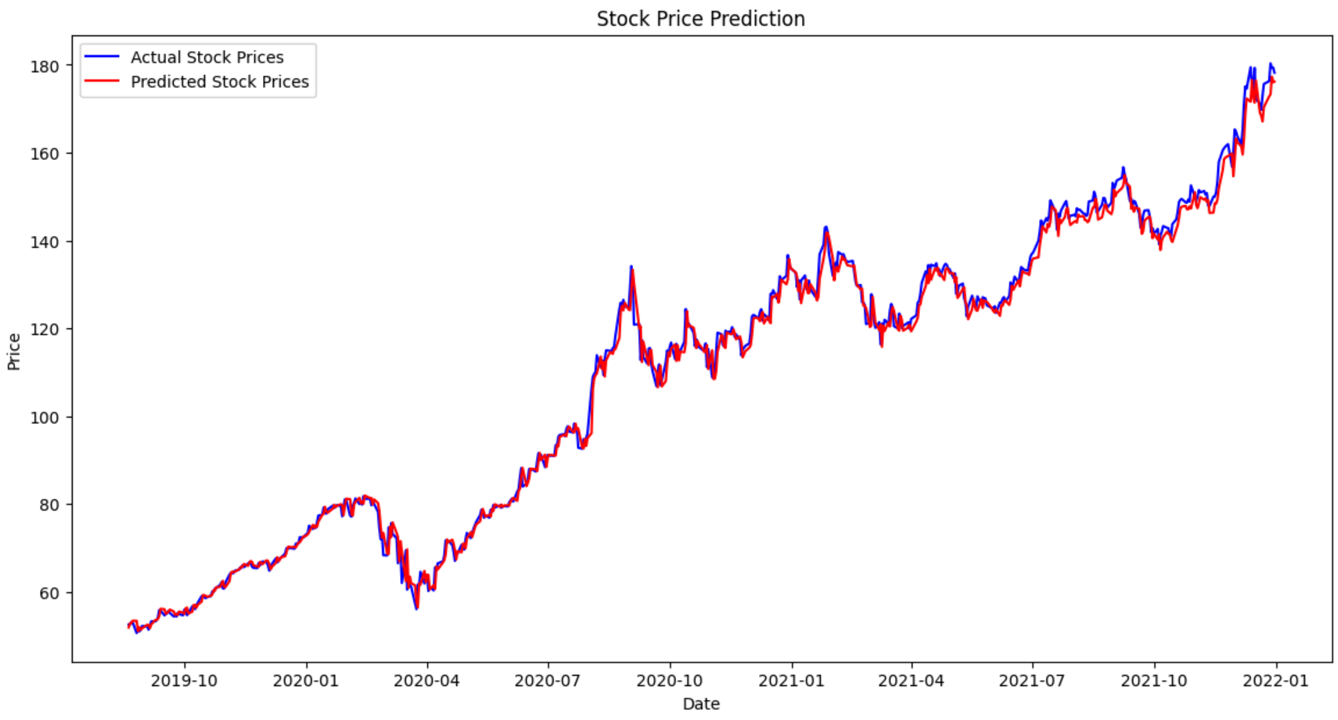
```
    Epoch 1/100
    38/38 [==============================] - 7s 42ms/step - loss: 8.0610e-04 - accuracy: 0.0000e+00
    Epoch 2/100
    38/38 [==============================] - 2s 41ms/step - loss: 2.2976e-05 - accuracy: 0.0000e+00
    Epoch 3/100
    38/38 [==============================] - 2s 40ms/step - loss: 1.7662e-05 - accuracy: 0.0000e+00
    Epoch 4/100
    38/38 [==============================] - 2s 40ms/step - loss: 1.7049e-05 - accuracy: 0.0000e+00
    Epoch 5/100
    38/38 [==============================] - 2s 41ms/step - loss: 1.6964e-05 - accuracy: 0.0000e+00
    Epoch 6/100
    38/38 [==============================] - 3s 67ms/step - loss: 1.6387e-05 - accuracy: 0.0000e+00
    Epoch 7/100
    38/38 [==============================] - 2s 49ms/step - loss: 1.6875e-05 - accuracy: 0.0000e+00
    Epoch 8/100
    38/38 [==============================] - 2s 58ms/step - loss: 1.6760e-05 - accuracy: 0.0000e+00
    Epoch 9/100
    38/38 [==============================] - 2s 40ms/step - loss: 1.7468e-05 - accuracy: 0.0000e+00
    Epoch 10/100
    38/38 [==============================] - 2s 41ms/step - loss: 1.5488e-05 - accuracy: 0.0000e+00
    Epoch 11/100
    38/38 [==============================] - 2s 40ms/step - loss: 1.5590e-05 - accuracy: 0.0000e+00
    Epoch 12/100
    38/38 [==============================] - 2s 40ms/step - loss: 1.4748e-05 - accuracy: 0.0000e+00
    Epoch 13/100
    38/38 [==============================] - 3s 67ms/step - loss: 1.4596e-05 - accuracy: 0.0000e+00
    Epoch 14/100
    38/38 [==============================] - 4s 92ms/step - loss: 1.6386e-05 - accuracy: 0.0000e+00
    Epoch 15/100
    38/38 [==============================] - 2s 51ms/step - loss: 1.4804e-05 - accuracy: 0.0000e+00
    Epoch 16/100
    38/38 [==============================] - 3s 67ms/step - loss: 1.3669e-05 - accuracy: 0.0000e+00
    Epoch 17/100
    38/38 [==============================] - 2s 56ms/step - loss: 1.3039e-05 - accuracy: 0.0000e+00
    Epoch 18/100
    38/38 [==============================] - 2s 41ms/step - loss: 1.3046e-05 - accuracy: 0.0000e+00
    Epoch 19/100
    38/38 [==============================] - 2s 40ms/step - loss: 1.2918e-05 - accuracy: 0.0000e+00
    Epoch 20/100
    38/38 [==============================] - 3s 68ms/step - loss: 1.3258e-05 - accuracy: 0.0000e+00
    Epoch 21/100
    38/38 [==============================] - 2s 52ms/step - loss: 1.3225e-05 - accuracy: 0.0000e+00
    Epoch 22/100
    38/38 [==============================] - 2s 42ms/step - loss: 1.2052e-05 - accuracy: 0.0000e+00
    Epoch 23/100
    38/38 [==============================] - 2s 40ms/step - loss: 1.1751e-05 - accuracy: 0.0000e+00
    Epoch 24/100
    38/38 [==============================] - 2s 40ms/step - loss: 1.2054e-05 - accuracy: 0.0000e+00
    Epoch 25/100
    38/38 [==============================] - 2s 41ms/step - loss: 1.1699e-05 - accuracy: 0.0000e+00
    Epoch 26/100
    38/38 [==============================] - 2s 40ms/step - loss: 1.1842e-05 - accuracy: 0.0000e+00
    Epoch 27/100
    38/38 [==============================] - 2s 45ms/step - loss: 1.2059e-05 - accuracy: 0.0000e+00
    Epoch 28/100
    38/38 [==============================] - 3s 67ms/step - loss: 1.0917e-05 - accuracy: 0.0000e+00
    Epoch 29/100
    38/38 [==============================] - 2s 40ms/step - loss: 1.1099e-05 - accuracy: 0.0000e+00
```

```
model.evaluate(X,y)
```

```
    94/94 [==============================] - 2s 10ms/step - loss: 4.8620e-05 - accuracy: 3.3445e-04
    [4.862000059802085e-05, 0.00033444815198890865]
```

```
predictions = model.predict(X_test)
predictions = scaler.inverse_transform(predictions)
y_test=scaler.inverse_transform(y_test.reshape(-1, 1))

plt.figure(figsize=(14, 7))
plt.plot(data.index[train_size + time_step + 1:], y_test, color='blue', label='Actual Stock Prices')
plt.plot(data.index[train_size + time_step + 1:], predictions, color='red', label='Predicted Stock Prices')
plt.title('Stock Price Prediction')
plt.xlabel('Date')
plt.ylabel('Price')
plt.legend()
plt.show()
```

```
19/19 [==============================] - 1s 10ms/step
```


Stock Price Prediction
Actual Stock Prices
Predicted Stock Prices
Price
Date