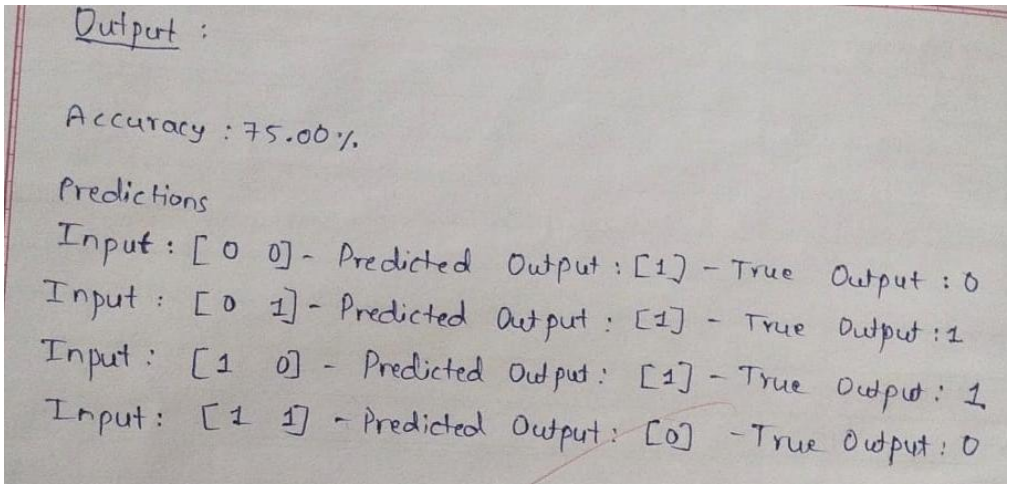
DEEP LEARNING LABORATORY

1. Solve XOR problem using multilayer perceptron

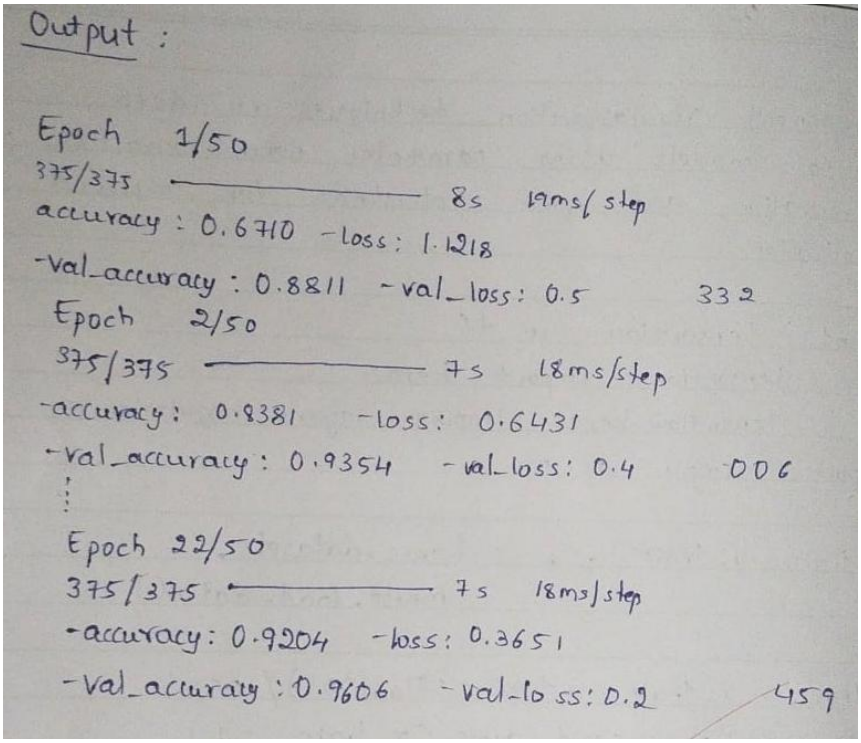
```
import numpy as np
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense,Input
from tensorflow.keras.optimizers import Adam
X=np.array([[0,0],[0,1],[1,0],[1,1]])
y=np.array([0,1,1,0])
model=Sequential()
model.add(Input(shape=(2,)))
model.add(Dense(2,activation='relu'))
model.add(Dense(1,activation='sigmoid'))
model.compile(loss='binary_crossentropy',optimizer=Adam(learning_rate=0.01),metrics=['accur
acy'])
model.fit(X,y,epochs=1000,verbose=0)
loss,accuracy=model.evaluate(X,y)
print(f"Accuracy:{accuracy*100:.2f}%")
predictions=model.predict(X)
predictions=(predictions>0.5).astype(int)
print("\nPredictions")
for i,prediction in enumerate(predictions):
    print(f"Input:{X[i]}-Predicted Output:{prediction}-True Output:{y[i]}")
```



---

2. Implement resularization technique in deep learning models using parameter norm penalties, augmentation, and noise robustness for improved generalization.

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers, regularizers
import numpy as np


(x_train, y_train), _ = keras.datasets.mnist.load_data()

x_train = x_train.astype('float32') / 255
x_train = np.expand_dims(x_train, -1)
y_train = tf.keras.utils.to_categorical(y_train, 10)

# Noise robustness
noise = 0.05 * np.random.normal(size=x_train.shape)
x_train = np.clip(x_train + noise, 0., 1.)

# Augmentation
datagen = keras.preprocessing.image.ImageDataGenerator(
rotation_range=10,     width_shift_range=0.1,
height_shift_range=0.1,
 validation_split=0.2
 )

datagen.fit(x_train)

model = keras.Sequential([
   layers.Flatten(input_shape=(28, 28, 1)),
layers.Dense(256, activation='relu',
            kernel_regularizer=regularizers.l1_l2(l1=1e-5,l2=1e-4)),
   layers.Dropout(0.5),
   layers.Dense(128, activation='relu',
            kernel_regularizer=regularizers.l2(1e-4)),
   layers.Dropout(0.3),
   layers.Dense(10, activation='softmax')
])

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


early_stop = keras.callbacks.EarlyStopping(monitor='val_loss',
patience=3,restore_best_weights=True)

history = model.fit(
datagen.flow(x_train, y_train, batch_size=128, subset='training'),
validation_data=datagen.flow(x_train, y_train, batch_size=128, subset='validation'),
 epochs=50,
 callbacks=[early_stop]
 )
```

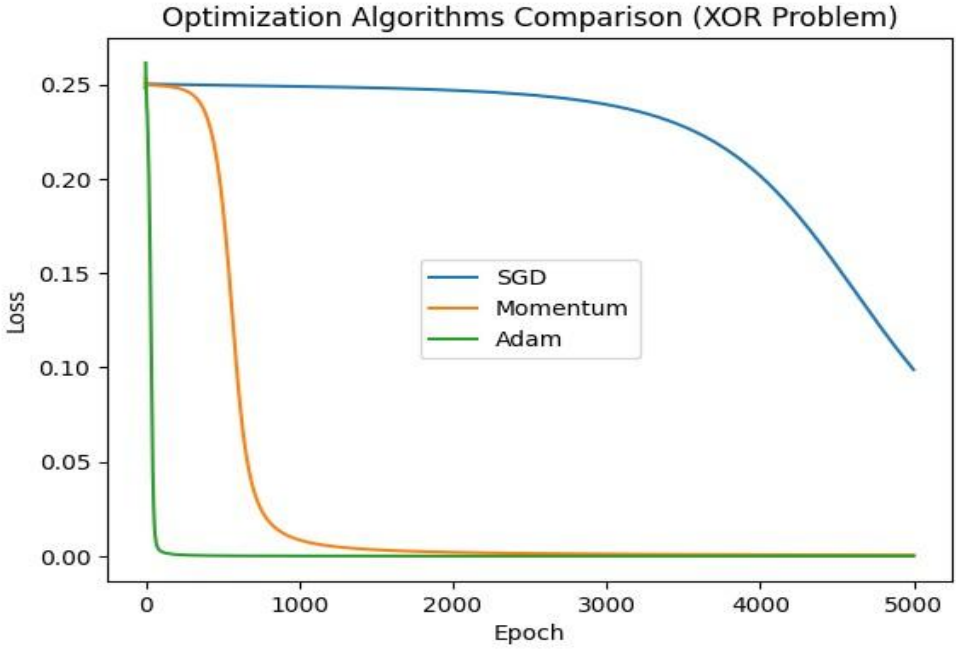


---

3. Implement and compare different optimization algorithms (SGD, Momentum, Adam) for training a simple neural network on a toy dataset.

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers, optimizers, losses
import numpy as np
import matplotlib.pyplot as plt

X = np.array([[0,0],[0,1],[1,0],[1,1]], dtype=np.float32)
y = np.array([[0],[1],[1],[0]], dtype=np.float32)
def  create_model():
   model = keras.Sequential([
      layers.Dense(4, input_dim=2, activation='sigmoid'),
      layers.Dense(1, activation='sigmoid')
   ])
  return model
def train_model(optimizer_name, X, y, epochs=5000, lr=0.1):
   model = create_model()
   if optimizer_name == 'SGD':
      optimizer = optimizers.SGD(learning_rate=lr)
```

```
    elif optimizer_name == 'Momentum':
      optimizer = optimizers.SGD(learning_rate=lr, momentum=0.9)
    elif optimizer_name == 'Adam':
      optimizer = optimizers.Adam(learning_rate=lr)
    model.compile(optimizer=optimizer, loss=losses.MeanSquaredError())
    history = model.fit(X, y, epochs=epochs, verbose=0)
    return history.history['loss']
losses_sgd = train_model('SGD', X, y)
losses_momentum = train_model('Momentum', X, y)
losses_adam = train_model('Adam', X, y)

plt.plot(losses_sgd, label='SGD')
plt.plot(losses_momentum, label='Momentum')
plt.plot(losses_adam, label='Adam')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.title('Optimization Algorithms Comparison (XOR Problem)')
plt.show()
print(f"Final Loss (SGD): {losses_sgd[-1]:.6f}")
print(f"Final Loss (Momentum): {losses_momentum[-1]:.6f}")
print(f"Final Loss (Adam): {losses_adam[-1]:.6f}")
```



Final Loss (SGD): 0.098835
Final Loss (Momentum): 0.000564
Final Loss (Adam): 0.000002

---

4. Build a CNN on MNIST dataset to demonstrate convolution, pooling and classification.

```
 import tensorflow as tf
 from tensorflow.keras import layers,models
 from tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import to_categorical
import matplotlib.pyplot as plt

(x_train,y_train),(x_test,y_test)=mnist.load_data)

x_train=x_train.reshape((x_train.shape[0],28,28,1)).astype('float32')/255
x_test=x_test.reshape((x_test.shape[0],28,28,1)).astype('float32')/255

y_train=to_categorical(y_train)
y_test=to_categorical(y_test)

 model=models.Sequential([
   layers.Conv2D(32,(3,3),activation='relu',input_shape=(28,28,1)),
layers.MaxPooling2D((2,2)),     layers.Conv2D(64,(3,3),activation='relu'),
layers.MaxPooling2D((2,2)),
  layers.Flatten(),
layers.Dense(64,activation='relu'),
  layers.Dense(10,activation='softmax')
])



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

history=model.fit(x_train,y_train,epochs=5,batch_size=64,validation_split=0.1)

test_loss,test_acc=model.evaluate(x_test,y_test)
print(f"Test accuracy:{test_acc:.4f}")

plt.plot(history.history['accuracy'],label='Train accuracy')
plt.plot(history.history['val_accuracy'],label='Val accuracy')
plt.title('Training and Validation acuuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.grid(True)
plt.show()
```

OUTPUT:

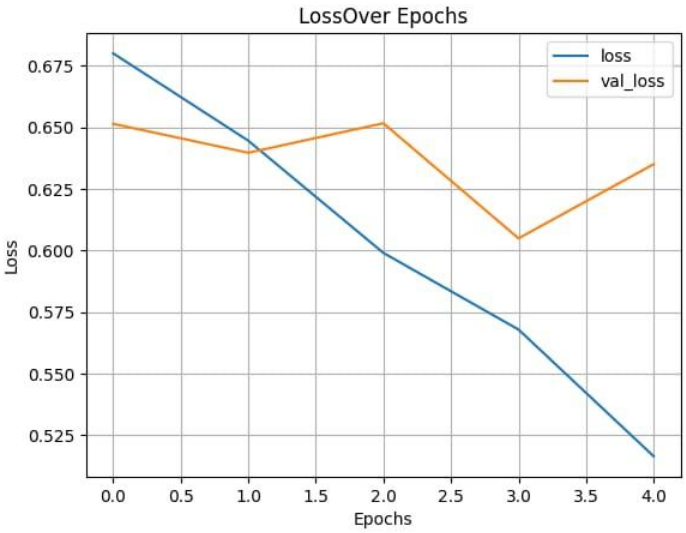
```
Epoch 1/5
844/844 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 40s 45ms/step - accuracy: 0.8742
- loss: 0.4310 - val_accuracy: 0.9823 - val_loss: 0.0622
Epoch 2/5
844/844 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 38s 45ms/step - accuracy: 0.9820
- loss: 0.0583 - val_accuracy: 0.9880 - val_loss: 0.0449
Epoch 3/5
844/844 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 41s 45ms/step - accuracy: 0.9885
- loss: 0.0368 - val_accuracy: 0.9890 - val_loss: 0.0400 Epoch 4/5
844/844 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 42s 46ms/step - accuracy: 0.9920
- loss: 0.0256 - val_accuracy: 0.9903 - val_loss: 0.0344
Epoch 5/5
844/844 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 39s 47ms/step - accuracy: 0.9946
- loss: 0.0187 - val_accuracy: 0.9902 - val_loss: 0.0320
313/313 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.9864 -
loss: 0.0441 Test accuracy:0.9888
```

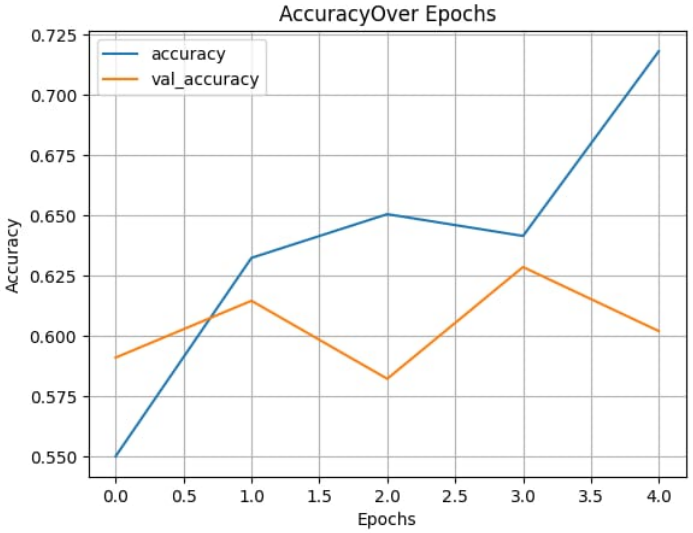
5.Implement an LSTM-based RNN for text classification to demonstrate sequence modelling,unfolding computational graphs ,and handling long-term dependencies.

```
import tensorflow as tf
from tensorflow.keras.datasets import imdb
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding,LSTM,Dense,Dropout
import matplotlib.pyplot as plt


vocab_size=10000
max_sequence_length=200


(x_train,y_train),(x_test,y_test)=imdb.load_data(num_words=vocab_size)
x_train=pad_sequences(x_train,maxlen=max_sequence_length,padding='post')
x_test=pad_sequences(x_test,maxlen=max_sequence_length,padding='post')

model=Sequential()
model.add(Embedding(input_dim=vocab_size,output_dim=64,input_length=max_sequence_len
gth))
model.add(LSTM(units=64,return_sequences=False))
model.add(Dropout(0.5))
model.add(Dense(1,activation='sigmoid'))
model.compile(optimizer='adam',loss='binary_crossentropy',metrics=['accuracy'])
print("\nTraining...")
history=model.fit(x_train,y_train,epochs=5,batch_size=64,validation_split=0.2)
print("\n Evaluating on test set...")
loss,accuracy=model.evaluate(x_test,y_test)
print(f"\nTest Accuracy:{accuracy:.4f}")


def plot_graphs(history,string):
  plt.plot(history.history[string])
  plt.plot(history.history['val_'+string])
  plt.xlabel("Epochs")
  plt.ylabel(string.capitalize())
```

```
    plt.title(f"{string.capitalize()}Over Epochs")
    plt.legend([string,'val_'+string])
    plt.grid(True)
    plt.show()

plot_graphs(history,"accuracy")
plot_graphs(history,"loss")
```

OUTPUT:





6.Construct a Bidirectional RNN and compare its performance with a standard RNN on sequence prediction tasks.

```
import tensorflow as tf
import numpy as np
#Synthetic dataset:predict next number in a sequence
seq_length=10
num_samples=1000

X=[]
y=[]

for _ in range(num_samples):
   start=np.random.randint(0,100)
   seq=np.arange(start,start+seq_length)
   X.append(seq[:-1])
```

```
    y.append(seq[1:])
X=np.array(X,dtype=np.float32)[...,np.newaxis]#(batch,seq_len-1,1)
y=np.array(y,dtype=np.float32)[...,np.newaxis]#(batch,seq_len-1,1)

#Define simple RNN model
def build_simple_rnn(input_size=1,hidden_size=32,output_size=1):
    model=tf.keras.Sequential([
        tf.keras.layers.SimpleRNN(hidden_size,return_sequences=True,
                                  input_shape=(seq_length-1,input_size)),
        tf.keras.layers.Dense(output_size)
    ])
    model.compile(optimizer=tf.keras.optimizers.Adam(0.01),loss='mse')
    return model

#Define Bidirectional RNN model
def build_birnn(input_size=1,hidden_size=32,output_size=1):
    model=tf.keras.Sequential([
        tf.keras.layers.Bidirectional(
        tf.keras.layers.SimpleRNN(hidden_size,return_sequences=True),
            input_shape=(seq_length-1,input_size)),
        tf.keras.layers.Dense(output_size)
    ])
    model.compile(optimizer=tf.keras.optimizers.Adam(0.01),loss='mse')
    return model

#Train models
def train_model(model,X,y,epochs=50):
    history=model.fit(X,y,epochs=epochs,verbose=0)
    return history.history['loss'][-1]

rnn_model=build_simple_rnn()
birnn_model=build_birnn()

rnn_loss=train_model(rnn_model,X,y)
birnn_loss=train_model(birnn_model,X,y)

print(f"Final Loss(RNN):{rnn_loss:.4f}")
print(f"Final Loss(BiRNN):{birnn_loss:.4f}")
```

OUTPUT:

Final Loss(RNN):0.7277

Final Loss(BiRNN):1.1334

7.Implement an encoder-decoder architecture with LSTMs for sequence-to-sequence learning.

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

#Sample data:input->output(reverse sequence)
num_samples=1000
timesteps=5
input_dim=1
latent_dim=32

#create toy data

X=np.random.rand(num_samples,timesteps,input_dim)
Y=np.flip(X,axis=1)#reversed sequences

#Encoder
encoder_inputs=Input(shape=(timesteps,input_dim))
encoder_outputs,state_h,state_c=LSTM(latent_dim,return_state=True)(encoder_inputs)
encoder_states=[state_h,state_c]

#Decoder
decoder_inputs=Input(shape=(timesteps,input_dim))
decoder_lstm=LSTM(latent_dim,return_sequences=True,return_state=True)
decoder_outputs,_,_=decoder_lstm(decoder_inputs,initial_state=encoder_states)
decoder_dense=Dense(input_dim)
decoder_outputs=decoder_dense(decoder_outputs)

#Full model
model=Model([encoder_inputs,decoder_inputs],decoder_outputs)
model.compile(optimizer='adam',loss='mse')

#Train(teacher forcing-decoder gets previous true sequence)
model.fit([X,Y],Y,epochs=10,batch_size=32)

#Test
pred=model.predict([X[:1],Y[:1]])
print("Input:\n",X[0].squeeze())
print("Predicted reversed:\n",pred[0].squeeze())
print("True reversed:\n",Y[0].squeeze())
```

OUTPUT:

```
Epoch 1/10
32/32 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - loss: 0.1562
Epoch 2/10
32/32 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - loss: 0.0648
Epoch 3/10
```

```
32/32 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - loss: 0.0579
Epoch 4/10
32/32 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 0.0519
Epoch 5/10
32/32 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - loss: 0.0441
Epoch 6/10
32/32 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - loss: 0.0345
Epoch 7/10
32/32 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - loss: 0.0235
Epoch 8/10
32/32 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - loss: 0.0122
Epoch 9/10
32/32 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - loss: 0.0043
Epoch 10/10
32/32 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - loss: 0.0015
1/1 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 130ms/step
Input:
 [0.71156131 0.67760978 0.03392224 0.32621476 0.03627892]
Predicted reversed:
 [0.00841371 0.26170507 0.09245112 0.6876484  0.7192257 ]
True reversed:
 [0.03627892 0.32621476 0.03392224 0.67760978 0.71156131]
```

8.Implement a Restricted Boltzmann Machine (RBM) for learning binary data representations.

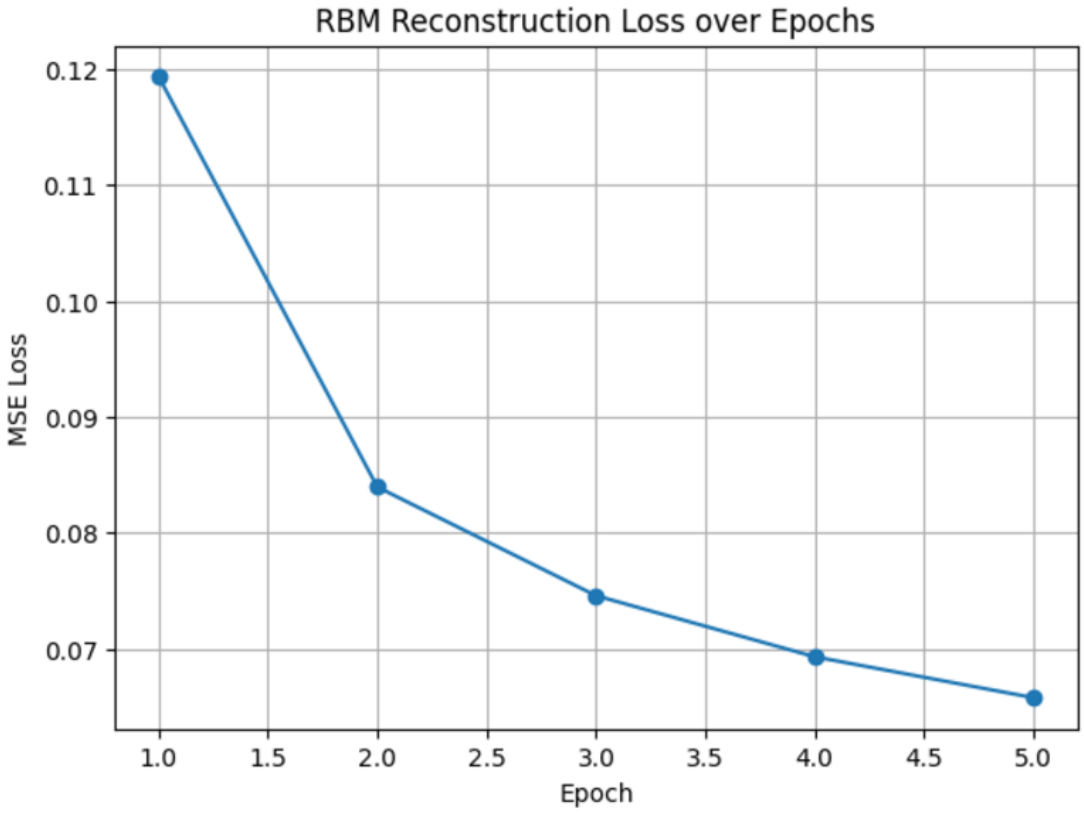
```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt

# --- Load and preprocess MNIST ---
(x_train, _), _ = tf.keras.datasets.mnist.load_data()
x_train = x_train.astype('float32') / 255.0
x_train = (x_train > 0.5).astype('float32')  # binarize
x_train = x_train.reshape(-1, 784)

batch_size = 64
train_dataset=
tf.data.Dataset.from_tensor_slices(x_train).shuffle(10000).batch(batch_size)
```

```
# --- RBM Class ---
class RBM(tf.keras.Model):
    def __init__(self, n_visible, n_hidden):
        super(RBM, self).__init__()
        self.n_visible = n_visible
        self.n_hidden = n_hidden

        # Parameters
        initializer = tf.initializers.RandomNormal(mean=0.0, stddev=0.01)
        self.W = tf.Variable(initializer([n_visible, n_hidden]), name='weights')
        self.h_bias = tf.Variable(tf.zeros([n_hidden]), name='hidden_bias')
        self.v_bias = tf.Variable(tf.zeros([n_visible]), name='visible_bias')

    def sample_prob(self, probs):
        """Sample binary values from probabilities."""
        return tf.nn.relu(tf.sign(probs - tf.random.uniform(tf.shape(probs))))

    def sample_h(self, v):
        prob_h = tf.nn.sigmoid(tf.matmul(v, self.W) + self.h_bias)
        return prob_h, self.sample_prob(prob_h)

    def sample_v(self, h):
        prob_v = tf.nn.sigmoid(tf.matmul(h, tf.transpose(self.W)) + self.v_bias)
        return prob_v, self.sample_prob(prob_v)

    def contrastive_divergence(self, v, lr=0.01):
        # Positive phase
        prob_h, h0 = self.sample_h(v)

        # Negative phase (reconstruction)
        prob_v, v1 = self.sample_v(h0)
        prob_h1, _ = self.sample_h(v1)

        # Compute gradients
```

```
        positive_grad = tf.matmul(tf.transpose(v), prob_h)
        negative_grad = tf.matmul(tf.transpose(v1), prob_h1)

        # Update weights and biases
        batch_size = tf.cast(tf.shape(v)[0], tf.float32)
        self.W.assign_add(lr * (positive_grad - negative_grad) / batch_size)
        self.v_bias.assign_add(lr * tf.reduce_mean(v - v1, axis=0))
        self.h_bias.assign_add(lr * tf.reduce_mean(prob_h - prob_h1, axis=0))

        # Compute reconstruction loss (MSE)
        loss = tf.reduce_mean(tf.square(v - v1))
        return loss

# --- Initialize and train RBM ---
n_visible = 784
n_hidden = 128
rbm = RBM(n_visible, n_hidden)

n_epochs = 5
lr = 0.05
losses = []

for epoch in range(n_epochs):
    epoch_loss = 0
    for batch in train_dataset:
        loss = rbm.contrastive_divergence(batch, lr)
        epoch_loss += loss.numpy()
    avg_loss = epoch_loss / len(list(train_dataset))
    losses.append(avg_loss)
    print(f"Epoch {epoch+1}/{n_epochs}, Reconstruction Loss: {avg_loss:.4f}")

# --- Plot training loss ---
plt.figure(figsize=(7, 5))
```

```
plt.plot(range(1, n_epochs+1), losses, marker='o')
plt.title("RBM Reconstruction Loss over Epochs")
plt.xlabel("Epoch")
plt.ylabel("MSE Loss")
plt.grid(True)
plt.show()
```

OUTPUT:

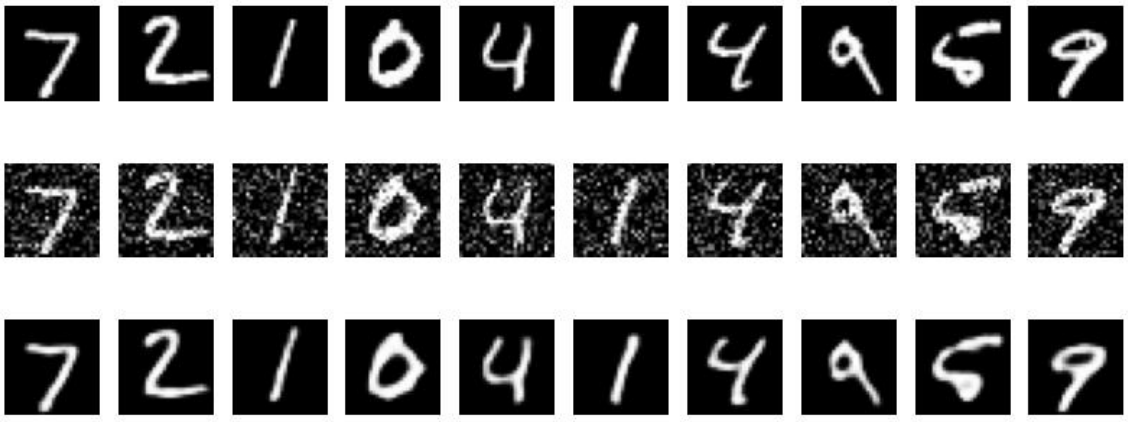


9.Develop a denoising autoencoder to reconstruct clean images from noisy input data.

```
import tensorflow as tf
from tensorflow.keras import layers,models
import matplotlib.pyplot as plt
import numpy as np

(x_train,_),(x_test,_)=tf.keras.datasets.mnist.load_data()
x_train,x_test=x_train/255.0,x_test/255.0
x_train,x_test=np.expand_dims(x_train,-1),np.expand_dims(x_test,-1)

x_train_noisy=np.clip(x_train+0.3*np.random.normal(0,1,x_train.shape),0,1)
x_test_noisy=np.clip(x_test+0.3*np.random.normal(0,1,x_test.shape),0,1)

plt.figure(figsize=(10,4))
for i in range(10):
   plt.subplot(2,10,i+1);
   plt.imshow(x_test[i].squeeze(),cmap='gray');
   plt.axis('off')
   plt.subplot(2,10,i+11);
   plt.imshow(x_test_noisy[i].squeeze(),cmap='gray');
   plt.axis('off')
plt.show()

autoencoder=models.Sequential([
   layers.Conv2D(32,(3,3),activation='relu',padding='same',input_shape=(28,28,1)),
   layers.MaxPooling2D((2,2),padding='same'),
   layers.Conv2D(64,(3,3),activation='relu',padding='same'),
   layers.MaxPooling2D((2,2),padding='same'),
   layers.Conv2D(64,(3,3),activation='relu',padding='same'),
   layers.UpSampling2D((2,2)),
   layers.Conv2D(32,(3,3),activation='relu',padding='same'),
   layers.UpSampling2D((2,2)),
   layers.Conv2D(1,(3,3),activation='sigmoid',padding='same')
])
```

```
autoencoder.compile(optimizer='adam',loss='binary_crossentropy')
autoencoder.fit(x_train_noisy,x_train,epochs=5,batch_size=256,validation_data=(x_test_noisy,x_test))

decoded=autoencoder.predict(x_test_noisy)

plt.figure(figsize=(10,4))
for i in range(10):
   plt.subplot(3,10,i+1);
   plt.imshow(x_test[i].squeeze(),cmap='gray');
   plt.axis('off')
   plt.subplot(3,10,i+11);
   plt.imshow(x_test_noisy[i].squeeze(),cmap='gray');
   plt.axis('off')
   plt.subplot(3,10,i+21);
   plt.imshow(decoded[i].squeeze(),cmap='gray');
   plt.axis('off')
plt.show()
```

OUTPUT:



10.Implement a simple Generative Adversarial Network(GAN) on the MNIST dataset to generate newhandwritten digits.

```
import tensorflow as tf
from tensorflow.keras import layers
import numpy as np
import matplotlib.pyplot as plt

#load and normalize MNIST
(x_train,_),_=tf.keras.datasets.mnist.load_data()
x_train=(x_train.astype("float32")-127.5)/127.5
x_train=np.expand_dims(x_train,axis=-1)
batch_size=128
buffer_size=60000
train_dataset=tf.data.Dataset.from_tensor_slices(x_train).shuffle(buffer_size).batch(
batch_size)

#Generator
def build_generator():
    model=tf.keras.Sequential([
        layers.Dense(7*7*128,input_dim=100),
        layers.LeakyRelu(0.2),
        layers.Reshape((7,7,128)),
        layers.Conv2DTranspose(64,(5,5),strides=2,padding="same"),
        layers.LeakyRelu(0.2),
        layers.Conv2DTranspose(1,(5,5),strides=2,padding="same",activation="tanh")
      ])
    return model

#Discriminator
def build_discriminator():
    model=tf.keras.Sequential([
        layers.Conv2D(64,(5,5),strides=2,padding="same",input_shape=[28,28,1]),
        layers.LeakyRelu(0.2),
        layers.Dropout(0.3),
        layers.Conv2D(128,(5,5),strides=2,padding="same"),
        layers.LeakyRelu(0.2),
        layers.Dropout(0.3),
        layers.Flatten(),
        layers.Dense(1,activation="sigmoid")
      ])
    return model

#Models
generator=build_generator()
discriminator=build_discriminator()
discriminator.compile(loss="binary_crossentropy",optimizer="adam",metrics=["accur
acy"])
```

```
#Gan combined model
discriminator.trainable=False
gan_input=tf.keras.Input(shape=(100,))
gan_output=discriminator(generator(gan_input))
gan=tf.keras.Model(gan_input,gan_output)
gan.compile(loss="binary_crossentropy",optimizer="adam")

#--Training--
epochs=5000#small but enough to get clear digits
for step in range(epochs):
    #Train Discriminator
    noise=np.random.normal(0,1,(batch_size,100))
    fake=generator.predict(noise,verbose=0)
    real=x_train[np.random.randint(0,x_train.shape[0],batch_size)]
    X=np.concatenate([real,fake])
    y=np.concatenate([np.omes((batch_size,1)),np.zeros((batch_size,1))])
    d_loss,_=discriminator.train_on_batch(X,y)

    #Train Generator
```

(NOT COMPLETED YET)