Practical No: 01

Aim: 1 A.) Implement advanced deep learning algorithms such as CNN.

Code:

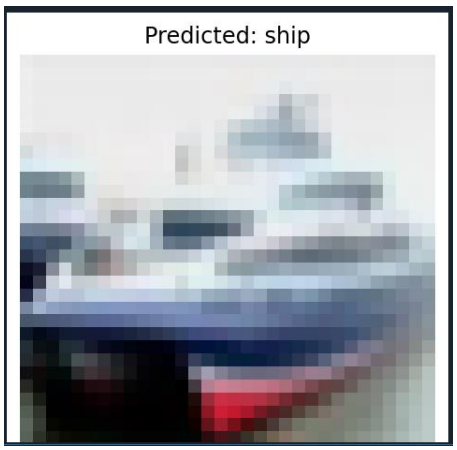
```
import tensorflow as tf
from tensorflow.keras import layers, models
import matplotlib.pyplot as plt
import numpy as np

#load CIFAR-10 dataset
(x_train, y_train),(x_test, y_test) = tf.keras.datasets.cifar10.load_data()

#normalize pixel values to [0,1]
x_train, x_test = x_train/255.0, x_test/255.0
class_names= ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']

# Display first 25 images from the training set
plt.figure(figsize=(10,10))
for i in range(25):
    plt.subplot(5,5,i+1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(x_train[i])

    #theCIFAR labels happen to be arrays,
    #which is why you need the extra index
    plt.xlabel(class_names[y_train[i][0]])
    plt.show()
    # Build the CNN model
    model= models.Sequential([
        layers.Conv2D(64,(3,3), activation='relu', input_shape=(32,32,3)),
```

```
        layers.MaxPooling2D((2,2)),
        layers.Conv2D(128, (3,3), activation='relu'),
        layers.MaxPooling2D((2,2)),
        layers.Conv2D(128, (3,3), activation='relu'),
        layers.Flatten(),
        layers.Dense(64, activation='relu'),
        layers.Dense(10, activation='softmax')
        ])

    #compile the model
    model.compile(optimizer='adam',
            loss='sparse_categorical_crossentropy',
            metrics=['accuracy'])

#train the model
model.fit(x_train, y_train, epochs=10, batch_size=64, validation_split=0.2)

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

#pick one test image. Make a prediction on a single image
img = x_test[1]

#add batch dimension for prediction
img_batch = np.expand_dims(img, axis=0)

#predict class probabilities
pred_probs = model.predict(img_batch)
#get predicted class index
predicted_class = np.argmax(pred_probs)
```

```
#CIFAR-10 class names
class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']

#plot the image with predicted label
plt.imshow(img)
plt.title(f"Predicted: {class_names[predicted_class]}")
plt.axis('off')
plt.show()
model.summary()
```

Output:


```
Epoch 1/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 15s 23ms/step - accuracy: 0.4093 - loss: 1.6137 - val_accuracy: 0.5083 - val_loss: 1.3644
Epoch 2/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 14s 23ms/step - accuracy: 0.5605 - loss: 1.2353 - val_accuracy: 0.5868 - val_loss: 1.1724
Epoch 3/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 15s 24ms/step - accuracy: 0.6201 - loss: 1.0792 - val_accuracy: 0.6540 - val_loss: 0.9959
Epoch 4/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 15s 25ms/step - accuracy: 0.6616 - loss: 0.9649 - val_accuracy: 0.6604 - val_loss: 0.9828
Epoch 5/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 16s 25ms/step - accuracy: 0.6973 - loss: 0.8704 - val_accuracy: 0.6825 - val_loss: 0.9205
Epoch 6/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 15s 23ms/step - accuracy: 0.7222 - loss: 0.8014 - val_accuracy: 0.6848 - val_loss: 0.9012
Epoch 7/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 15s 24ms/step - accuracy: 0.7423 - loss: 0.7361 - val_accuracy: 0.7052 - val_loss: 0.8618
Epoch 8/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 16s 26ms/step - accuracy: 0.7632 - loss: 0.6828 - val_accuracy: 0.7139 - val_loss: 0.8352
Epoch 9/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 15s 25ms/step - accuracy: 0.7778 - loss: 0.6320 - val_accuracy: 0.7130 - val_loss: 0.8593
Epoch 10/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 16s 25ms/step - accuracy: 0.7950 - loss: 0.5876 - val_accuracy: 0.7081 - val_loss: 0.8694
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.7034 - loss: 0.8927
Test accuracy: 0.703400
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 63ms/step
```



Model: "sequential_49"

Layer (type)	Output Shape	Param #
conv2d_147 (Conv2D)	(None, 30, 30, 64)	1,792
max_pooling2d_98 (MaxPooling2D)	(None, 15, 15, 64)	0
conv2d_148 (Conv2D)	(None, 13, 13, 128)	73,856
max_pooling2d_99 (MaxPooling2D)	(None, 6, 6, 128)	0
conv2d_149 (Conv2D)	(None, 4, 4, 128)	147,584
flatten_49 (Flatten)	(None, 2048)	0
dense_98 (Dense)	(None, 64)	131,136
dense_99 (Dense)	(None, 10)	650

Total params: 1,065,056 (4.06 MB)
Trainable params: 355,018 (1.35 MB)
Non-trainable params: 0 (0.00 B)
Optimizer params: 710,038 (2.71 MB)

Variable Explorer:

Name	Type	Size	Value
class_names	list	10	['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'hors ...
i	int	1	24
img	Array of float64	[32, 32, 3]	[[[0.92156863 0.92156863 0.92156863] [0.90588235 0.90588235 0.905882 ...
img_batch	Array of float64	[1, 32, 32, 3]	[[[[0.92156863 0.92156863 0.92156863] [0.90588235 0.90588235 0.9058 ...
pred_probs	Array of float32	[1, 10]	[[4.7200164e-03 2.3733644e-02 2.6748961e-07 9.0459807e-08 1.0671085e-0 ...
predicted_class	int64	1	np.int64(8)
test_acc	float	1	0.7034000158309937
test_loss	float	1	0.8927410840988159
x_test	Array of float64	[10000, 32, 32, 3]	[[[[0.61960784 0.43921569 0.19215686] [0.62352941 0.43529412 0.1843 ...
x_train	Array of float64	[50000, 32, 32, 3]	[[[[0.23137255 0.24313725 0.24705882] [0.16862745 0.18039216 0.1764 ...
y_test	Array of uint8	[10000, 1]	[[3] [8]
y_train	Array of uint8	[50000, 1]	[[6] [9]

Aim: 1B.) Implement advanced deep learning algorithms such as RNN.

Code:

```
# Simple RNN Sentiment Analysis

# Sample data (tiny dataset for demo)

reviews = [

    "I loved the movie, it was fantastic!",       # positive

    "Absolutely terrible, worst film ever.",      # negative

    "Great acting and wonderful story.",          # positive

    "The movie was boring and too long.",         # negative

    "An excellent and emotional performance.",    # positive (fixed typo)

    "I hated it, very disappointing."             # negative

]

labels = [1, 0, 1, 0, 1, 0]  # 1: positive, 0: negative


import torch

from torch.nn.utils.rnn import pad_sequence

from torch.utils.data import DataLoader, TensorDataset

from collections import Counter

import re


# Tokenize and clean

def preprocess(text):

    text = text.lower()

    text = re.sub(r"[^\w\s]", "", text)

    return text.split()


tokenized_reviews = [preprocess(review) for review in reviews]


# Build vocab

all_words = [word for review in tokenized_reviews for word in review]

word_counts = Counter(all_words)
```

```
vocab = {word: i+1 for i, (word, _) in enumerate(word_counts.most_common())}  # start indexing from 1
vocab['<PAD>'] = 0  # padding token
vocab['<UNK>'] = len(vocab)  # unknown token

# Encode reviews
encoded_reviews = [[vocab.get(word, vocab['<UNK>']) for word in review] for review in tokenized_reviews]

# Pad sequences
padded_reviews = pad_sequence([torch.tensor(seq) for seq in encoded_reviews], batch_first=True)

labels_tensor = torch.tensor(labels)
dataset = TensorDataset(padded_reviews, labels_tensor)
dataloader = DataLoader(dataset, batch_size=2, shuffle=True)

# Model
import torch.nn as nn
class ReviewRNN(nn.Module):
    def __init__(self, vocab_size, embed_size, hidden_size, num_classes):
        super(ReviewRNN, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embed_size, padding_idx=0)
        self.rnn = nn.RNN(embed_size, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, num_classes)

    def forward(self, x):
        x = self.embedding(x)
        output, _ = self.rnn(x)
        out = self.fc(output[:, -1, :])  # last timestep hidden state
        return out

# Training setup
```

```
vocab_size = len(vocab)
embed_size = 32
hidden_size = 64
num_classes = 2

model = ReviewRNN(vocab_size, embed_size, hidden_size, num_classes)
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)

# Train loop
num_epochs = 10
for epoch in range(num_epochs):
    for batch_x, batch_y in dataloader:
        outputs = model(batch_x)
        loss = criterion(outputs, batch_y)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
    print(f"Epoch [{epoch+1}/{num_epochs}], Loss: {loss.item():.4f}")

# Prediction
def predict_sentiment(text):
    model.eval()
    with torch.no_grad():
        tokens = preprocess(text)
        encoded = [vocab.get(word, vocab['<UNK>']) for word in tokens]
        tensor = torch.tensor(encoded).unsqueeze(0)
        tensor = pad_sequence([tensor.squeeze()], batch_first=True)  # ensure padding shape
        output = model(tensor)
        pred = torch.argmax(output, dim=1).item()
```

```
    return "Positive" if pred == 1 else "Negative"


# Test prediction
print(predict_sentiment("I really enjoyed the movie"))
print(predict_sentiment("It was the worst movie ever"))
print(predict_sentiment("An excellent and emotional performance."))
print(predict_sentiment("Amazing movie!."))
```

Output:

```
In [13]: %runfile C:/Users/Admin/untitled2.py --wdir
Epoch [1/10], Loss: 0.7138
Epoch [2/10], Loss: 0.0634
Epoch [3/10], Loss: 0.0075
Epoch [4/10], Loss: 0.0016
Epoch [5/10], Loss: 0.0007
Epoch [6/10], Loss: 0.0002
Epoch [7/10], Loss: 0.0001
Epoch [8/10], Loss: 0.0001
Epoch [9/10], Loss: 0.0001
Epoch [10/10], Loss: 0.0001
Positive
Negative
Positive
Positive
```

Aim: 1 C.) Implement advanced deep learning algorithms such as CNN using Pytorch.

Code:

```
import torch
import torch.nn as nn
import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
import numpy as np

# Hyperparameters
batch_size = 64
num_classes = 10
learning_rate = 0.001
num_epochs = 20

# Set device (use GPU if available)
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# Data transformations for CIFAR-10
transform = transforms.Compose([
    transforms.Resize((32, 32)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.4914, 0.4822, 0.4465],
                         std=[0.2023, 0.1994, 0.2010])
])

# Load CIFAR-10 training and test datasets
train_dataset = torchvision.datasets.CIFAR10(
    root='./data', train=True, transform=transform, download=True)

test_dataset = torchvision.datasets.CIFAR10(
```

```
    root='./data', train=False, transform=transform, download=True)

train_loader = torch.utils.data.DataLoader(
    dataset=train_dataset, batch_size=batch_size, shuffle=True)

test_loader = torch.utils.data.DataLoader(
    dataset=test_dataset, batch_size=batch_size, shuffle=True)

# Define the convolutional neural network
class ConvNeuralNet(nn.Module):
    def __init__(self, num_classes):
        super(ConvNeuralNet, self).__init__()
        self.conv_layer1 = nn.Conv2d(in_channels=3, out_channels=32, kernel_size=3)
        self.conv_layer2 = nn.Conv2d(in_channels=32, out_channels=32, kernel_size=3)
        self.max_pool1 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.conv_layer3 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3)
        self.conv_layer4 = nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3)
        self.max_pool2 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.fc1 = nn.Linear(1600, 128)
        self.relu1 = nn.ReLU()
        self.fc2 = nn.Linear(128, num_classes)

    def forward(self, x):
        out = self.conv_layer1(x)
        out = self.conv_layer2(out)
        out = self.max_pool1(out)
        out = self.conv_layer3(out)
        out = self.conv_layer4(out)
        out = self.max_pool2(out)
        out = out.reshape(out.size(0), -1)
        out = self.fc1(out)
```

```
        out = self.relu1(out)
        out = self.fc2(out)
        return out

# Initialize model, loss function, and optimizer
model = ConvNeuralNet(num_classes).to(device)
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate, weight_decay=0.005,
momentum=0.9)

# Training loop
for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(train_loader):
        images = images.to(device)
        labels = labels.to(device)

        outputs = model(images)
        loss = criterion(outputs, labels)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if (i + 1) % 100 == 0 or (i + 1) == len(train_loader):
            print('Epoch [{}/{}], Step [{}/{}], Loss: {:.4f}'
                .format(epoch + 1, num_epochs, i + 1, len(train_loader), loss.item()))

# CIFAR-10 class names
classes = ['plane', 'car', 'bird', 'cat', 'deer',
           'dog', 'frog', 'horse', 'ship', 'truck']
```

```
# Helper function to unnormalize and display an image
def imshow(img):
    img = img * torch.tensor([0.2023, 0.1994, 0.2010]).view(3, 1, 1) + \
            torch.tensor([0.4914, 0.4822, 0.4465]).view(3, 1, 1)
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.axis('off')
    plt.show()

# Evaluation and prediction on test data
model.eval()

with torch.no_grad():
    correct = 0
    total = 0

    # Display a few test images with predictions
    dataiter = iter(test_loader)
    images, labels = next(dataiter)
    images_display = images[:8]
    labels_display = labels[:8]

    images_display = images_display.to(device)
    outputs = model(images_display)
    _, predicted = torch.max(outputs, 1)

    images_display = images_display.cpu()
    predicted = predicted.cpu()
    labels_display = labels_display.cpu()

    print('\nPredictions on sample test images:')
```

```
imshow(torchvision.utils.make_grid(images_display, nrow=4))
print('Predicted:', '  '.join(f'{classes[predicted[j]]}' for j in range(8)))
print('Actual:   ', '  '.join(f'{classes[labels_display[j]]}' for j in range(8)))

# Compute accuracy over the full test set
for images, labels in test_loader:
    images = images.to(device)
    labels = labels.to(device)
    outputs = model(images)
    _, predicted = torch.max(outputs.data, 1)
    total += labels.size(0)
    correct += (predicted == labels).sum().item()

print('\nAccuracy of the network on the 10000 test images: {:.2f} %'.format(100 * correct / total))
```

Output:

```
Predictions on sample test images:
```

```
Predicted: horse  dog  dog  frog  dog  ship  car  deer
Actual:    horse  car  bird  frog  plane  ship  car  deer

Accuracy of the network on the 10000 test images: 69.61 %
```

Practical No: 02

Aim: 2 A.) Build a NLP model for sentiment analysis

Code:

```
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

# Load IMDB Dataset
num_words = 10000
(x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=num_words)

# Set parameters
max_len = 200
x_train = pad_sequences(x_train, maxlen=max_len)
x_test = pad_sequences(x_test, maxlen=max_len)

# Build the model
model = Sequential([
    Embedding(input_dim=num_words, output_dim=128, input_length=max_len),
    LSTM(64, dropout=0.2, recurrent_dropout=0.2),
    Dense(1, activation='sigmoid')
])

# Compile the model
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])

# Train the model
model.fit(x_train, y_train, batch_size=64, epochs=3, validation_split=0.2)
```

```
# Evaluate the model
loss, accuracy = model.evaluate(x_test, y_test)
print(f"Test Accuracy: {accuracy:.4f}")

# Decode a review example
word_index = imdb.get_word_index()
index_offset = 3
word_index = {word: (index + index_offset) for word, index in word_index.items()}
word_index["<PAD>"] = 0
word_index["<START>"] = 1
word_index["<UNK>"] = 2

reverse_word_index = {index: word for word, index in word_index.items()}
decoded_review = " ".join([reverse_word_index.get(i, "?") for i in x_train[0]])

# Print decoded review and label
print("Decoded Review Example:\n", decoded_review)
print("Label:", y_train[0])
```

Output:

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/imdb.npz
17464789/17464789 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
Epoch 1/3
/usr/local/lib/python3.12/dist-packages/keras/src/layers/core/embedding.py:97: UserWarning: Argument `input_length` is deprecated. Just remove it.
  warnings.warn(
313/313 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 147s 453ms/step - accuracy: 0.6705 - loss: 0.5710 - val_accuracy: 0.8080 - val_loss: 0.4691
Epoch 2/3
313/313 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 182s 391ms/step - accuracy: 0.7610 - loss: 0.5175 - val_accuracy: 0.8034 - val_loss: 0.4300
Epoch 3/3
```

313/313 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **147s** 409ms/step - accuracy: 0.8832 - loss: 0.2934 - val_accuracy: 0.8132 - val_loss: 0.4455

782/782 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **48s** 62ms/step - accuracy: 0.8196 - loss: 0.4381

Test Accuracy: 0.8224

Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/imdb_word_index.json

1641221/1641221 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **0s** 0us/step

Decoded Review Example:

and you could just imagine being there robert <UNK> is an amazing actor and now the same being director <UNK> father came from the same scottish island as myself so i loved the fact there was a real connection with this film the witty remarks throughout the film were great it was just brilliant so much that i bought the film as soon as it was released for <UNK> and would recommend it to everyone to watch and the fly fishing was amazing really cried at the end it was so sad and you know what they say if you cry at a film it must have been good and this definitely was also <UNK> to the two little boy's that played the <UNK> of norman and paul they were just brilliant children are often left out of the <UNK> list i think because the stars that play them all grown up are such a big profile for the whole film but these children are amazing and should be praised for what they have done don't you think the whole story was so lovely because it was true and was someone's life after all that was shared with us all

Label: 1

Aim: 2 B.) Build a NLP model for text classification.

Code:

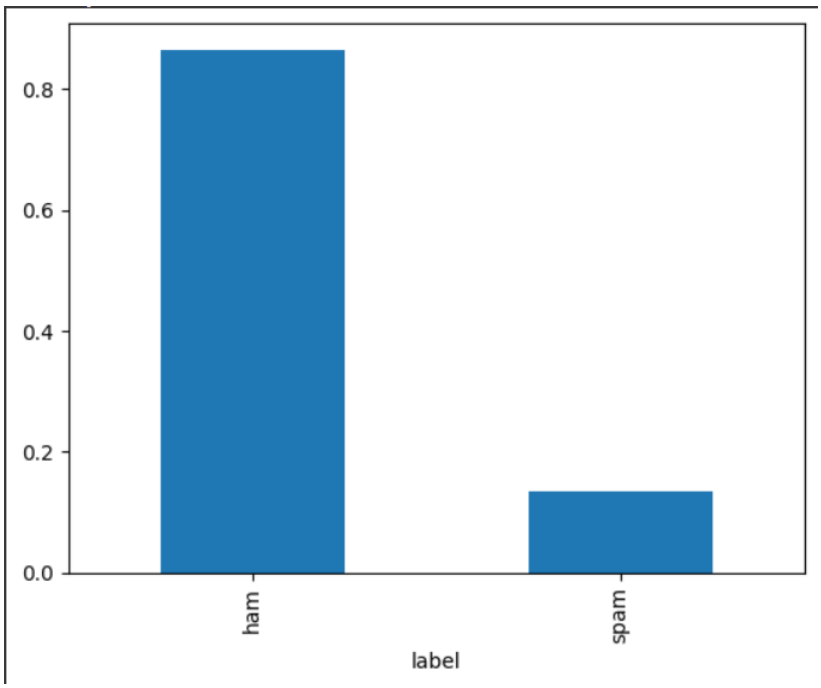
```
import pandas as pd

# Load dataset

data = pd.read_csv('https://raw.githubusercontent.com/mohitgupta-omg/Kaggle-SMS-Spam-Collection-Dataset-/master/spam.csv', encoding='latin-1')

# Check the first 5 rows

data.head()

# Drop unnecessary columns

data.drop(['Unnamed: 2', 'Unnamed: 3', 'Unnamed: 4'], axis=1, inplace=True)

data.columns = ['label', 'text']

# Check for missing values and dataset shape

data.head()

data.isna().sum()

data.shape

# Plot class distribution

data['label'].value_counts(normalize=True).plot.bar()

# Import necessary libraries

import nltk

nltk.download('all')

# Create a list of texts

text = list(data['text'])

# Preprocessing text data

import re

from nltk.corpus import stopwords

from nltk.stem import WordNetLemmatizer

lemmatizer = WordNetLemmatizer()

corpus = []

# Loop through each text and clean it

for i in range(len(text)):

    r = re.sub('[^a-zA-Z]', ' ', text[i])  # Remove non-alphabetic characters
```

```
    r = r.lower()  # Convert to lowercase
    r = r.split()  # Split into words
    r = [word for word in r if word not in stopwords.words('english')]  # Remove stopwords
    r = [lemmatizer.lemmatize(word) for word in r]  # Lemmatize words
    r = ' '.join(r)  # Join words back into a single string
    corpus.append(r)
# Update text column with cleaned data
data['text'] = corpus
data.head()
# Separate features and target variable
X = data['text']
y = data['label']
# Split data into training and testing sets
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=123)
# Print training and testing data shapes
print('Training Data:', X_train.shape)
print('Testing Data:', X_test.shape)
# Vectorize text data using CountVectorizer
from sklearn.feature_extraction.text import CountVectorizer
cv = CountVectorizer()
X_train_cv = cv.fit_transform(X_train)
print(X_train_cv.shape)
# Train Logistic Regression model
from sklearn.linear_model import LogisticRegression
lr = LogisticRegression()
lr.fit(X_train_cv, y_train)
# Transform test data and make predictions
X_test_cv = cv.transform(X_test)
predictions = lr.predict(X_test_cv)
# Import metrics and evaluate the model
```

```
import pandas as pd

from sklearn import metrics

df = pd.DataFrame(metrics.confusion_matrix(y_test, predictions), index=['ham', 'spam'], columns=['ham', 'spam'])

print(df)

# Print accuracy score

print("Accuracy:", metrics.accuracy_score(y_test, predictions))
```

Output:

```
[nltk_data]  Done downloading collection all
Training Data: (4457,)
Testing Data: (1115,)
(4457, 6214)
      ham  spam
ham   980     2
spam   16   117
Accuracy: 0.9838565022421525
```



Practical No: 03

Aim: Create a chatbot using advanced techniques like transformer models.

Code:

```
import torch

from transformers import GPT2LMHeadModel, GPT2Tokenizer

#Load the pre-trained GPT-2 model and tokenizer

model_name= "gpt2"

tokenizer = GPT2Tokenizer.from_pretrained(model_name)

model = GPT2LMHeadModel.from_pretrained(model_name)

#Set the model to evaluate mode

model.eval()

#Defines a function to generate a response given an input prompt, with a default maximun response length of 50 tokens

def generate_response(prompt, max_length=50):

   #Encodes the input prompt into token IDs and converts it into Pytorch tensor for processing by thr model.

   input_ids = tokenizer.encode(prompt, return_tensors = "pt")

   #Generate response

#Disables gradient calculation to save memory and computational resources during inference.

   with torch.no_grad():

 #Generates a response using the model, limiting the output to max_length tokens andhandling padding with the specified

 #token ID

      output = model.generate(input_ids, max_length=max_length, num_return_sequences=1, pad_token_id=50256)

#Decodes the generated token IDs back into a human-readable string, skipping any special tokens

   response = tokenizer.decode(output[0], skip_special_tokens= True)

   return response

print("Chatbot: Hi there! How can I help you?")

while True:

   user_input = input("You:")

   if user_input.lower() == "exit":
```

```
        print("Chatbot: Goodbye!!!")
        break
    response = generate_response(user_input)
    print("Chatbot:", response)
```

Output:

```
In [4]: %runfile C:/Users/Admin/untitled0.py --wdir
Chatbot: Hi there! How can I help you?
You:Hello, How are you?
Chatbot: Hello, How are you?

I'm a little bit nervous. I'm not sure if I'm going to be able to do this. I'm not sure if I'm going to be able to do this. I'm not
sure if
You:Tell me a joke
Chatbot: Tell me a joke about it.

I'm not sure if you're aware of it, but I'm not sure if you're aware of it.

I'm not sure if you're aware of it.

I'm not
You:Why is the sky blue?
Chatbot: Why is the sky blue?

The sky blue is a color that is often associated with the sky. It is a color that is often associated with the sky. It is a color
that is often associated with the sky. It is a color
You:exit
Chatbot: Goodbye!!!
```

Practical No: 04

Aim: Develop a recommendation system using collaborative filtering.

Code:

```
# Importing Libraries

import numpy as np

import pandas as pd

import sklearn

import matplotlib.pyplot as plt

import seaborn as sns

import warnings

warnings.simplefilter(action='ignore', category=FutureWarning)

# Loading rating dataset

ratings = pd.read_csv("ratings.csv")

print(ratings.head())

# Loading movie dataset

movies = pd.read_csv("movies.csv")

print(movies.head())

# Print the basic data from the dataset

n_ratings = len(ratings)

n_movies = len(ratings['movie_id'].unique())

n_users = len(ratings['user_id'].unique())

print(f"Number of ratings: {n_ratings}")

print(f"Number of unique movie_id's: {n_movies}")

print(f"Number of unique users: {n_users}")

print(f"Average ratings per user: {round(n_ratings / n_users, 2)}")

print(f"Average ratings per movie: {round(n_ratings / n_movies, 2)}")


# Group the ratings DataFrame by userId, count the number of movieId entries for each user, and reset the index

user_freq = ratings[['user_id', 'movie_id']].groupby('user_id').count().reset_index()
```

```
# Rename columns to userId and n_ratings to indicate number of ratings each user has provided
user_freq.columns = ['user_id', 'n_ratings']
print(user_freq.head())
# Find Lowest and Highest rated movies:
mean_rating = ratings.groupby('movie_id')[['rating']].mean()
# Lowest rated movie
lowest_rated = mean_rating['rating'].idxmin()
print("Lowest rated movie:")
print(movies.loc[movies['movie_id'] == lowest_rated])
# Highest rated movie
highest_rated = mean_rating['rating'].idxmax()
print("Highest rated movie:")
print(movies.loc[movies['movie_id'] == highest_rated])
# Show number of people who rated highest rated movie
print(ratings[ratings['movie_id'] == highest_rated])
# Show number of people who rated lowest rated movie
print(ratings[ratings['movie_id'] == lowest_rated])
# The above movies have very low dataset counts, we will use Bayesian average
movie_stats = ratings.groupby('movie_id')[['rating']].agg(['count', 'mean'])
movie_stats.columns = movie_stats.columns.droplevel()
# Now, we create user-item matrix using scipy csr matrix
from scipy.sparse import csr_matrix
def create_matrix(df):
    N = len(df['user_id'].unique())
    M = len(df['movie_id'].unique())
    # Map Ids to indices
    user_mapper = dict(zip(np.unique(df["user_id"]), list(range(N))))
    movie_mapper = dict(zip(np.unique(df["movie_id"]), list(range(M))))
    # Map indices to IDs
    user_inv_mapper = dict(zip(list(range(N)), np.unique(df["user_id"])))
    movie_inv_mapper = dict(zip(list(range(M)), np.unique(df["movie_id"])))
```

```
    user_index = [user_mapper[i] for i in df['user_id']]
    movie_index = [movie_mapper[i] for i in df['movie_id']]
    X = csr_matrix((df["rating"], (movie_index, user_index)), shape=(M, N))
    return X, user_mapper, movie_mapper, user_inv_mapper, movie_inv_mapper
X, user_mapper, movie_mapper, user_inv_mapper, movie_inv_mapper = create_matrix(ratings)
ratings_matrix = pd.DataFrame(X.toarray(), index=movie_inv_mapper.values(),
columns=user_inv_mapper.values())
print(ratings_matrix.head())
# Find similar movies using KNN
from sklearn.neighbors import NearestNeighbors
def find_similar_movies(movie_id, X, k, metric='cosine', show_distance=False):
    neighbour_ids = []
    # Get the index of the movie in the dataset from its movie ID
    movie_ind = movie_mapper[movie_id]
    movie_vec = X[movie_ind]  # Retrieve the feature vector of the selected movie
    # Initialize KNN model with specified params
    knn = NearestNeighbors(n_neighbors=k + 1, algorithm="brute", metric=metric)
    knn.fit(X)
    movie_vec = movie_vec.reshape(1, -1)
    neighbour = knn.kneighbors(movie_vec, return_distance=show_distance)
    # neighbor indices are in neighbour[0] when show_distance is False
    for i in range(k + 1):
        n = neighbour[0][i]
        neighbour_ids.append(movie_inv_mapper[n])
    neighbour_ids.pop(0)  # Remove the movie itself
    return neighbour_ids
# Create a dictionary mapping movie IDs to their titles
movie_titles = dict(zip(movies['movie_id'], movies['title']))
movies_id = 3
similar_ids = find_similar_movies(movies_id, X, k=10)
```

```
movie_title = movie_titles[movies_id]
print(f"Since you watched {movie_title}, you might also like:")
for i in similar_ids:
    print(movie_titles.get(i, "Movie not found"))
# Function to recommend movies for a user based on their highest-rated movie
def recommend_movies_for_user(users_id, X, user_mapper, movie_mapper, movie_inv_mapper, k=10):
    # Filter ratings for the user
    df1 = ratings[ratings['user_id'] == users_id]
    if df1.empty:
        print(f"User with ID {users_id} does not exist.")
        return
    # Movie ID of highest-rated movie by user
    movies_id = df1[df1['rating'] == df1['rating'].max()]['movie_id'].iloc[0]
    movie_titles = dict(zip(movies['movie_id'], movies['title']))
    similar_ids = find_similar_movies(movies_id, X, k)
    movie_title = movie_titles.get(movies_id, "Movie not found")
    if movie_title == "Movie not found":
        print(f"Movie with ID {movies_id} not found.")
        return
    print(f"Since you watched {movie_title}, you might also like:")
    for i in similar_ids:
        print(movie_titles.get(i, "Movie not found"))
users_id = 69  # Replace with desired user ID
recommend_movies_for_user(users_id, X, user_mapper, movie_mapper, movie_inv_mapper, k=10)
```

Output:

```
   rating_id  movie_id  user_id  rating            timestamp
0          1         1      101       9  2025-09-01 14:00:00
1          2         2      102       8  2025-09-02 15:00:00
2          3         3      103      10  2025-09-01 16:30:00
3          4         4      104       7  2025-09-01 17:45:00
4          5         5      105       9  2025-09-03 10:15:00
   movie_id            title   genre  release_year  duration_minutes
0         1        Inception  Sci-Fi          2010               148
1         2  The Dark Knight  Action          2008               152
2         3     Forrest Gump   Drama          1994               142
3         4       The Matrix  Sci-Fi          1999               136
4         5    The Godfather   Crime          1972               175
Number of ratings: 50
Number of unique movie_id's: 50
Number of unique users: 50
Average ratings per user: 1.0
Average ratings per movie: 1.0
   user_id  n_ratings
0      101          1
1      102          1
2      103          1
3      104          1
4      105          1
```

```
Lowest rated movie:
   movie_id       title   genre  release_year  duration_minutes
3         4  The Matrix  Sci-Fi          1999               136
Highest rated movie:
   movie_id         title  genre  release_year  duration_minutes
2         3  Forrest Gump  Drama          1994               142
   rating_id  movie_id  user_id  rating            timestamp
2          3         3      103      10  2025-09-01 16:30:00
   rating_id  movie_id  user_id  rating            timestamp
3          4         4      104       7  2025-09-01 17:45:00
   101  102  103  104  105  106  107  108  109  110  ...  141  142  143  144  \
1    9    0    0    0    0    0    0    0    0    0  ...    0    0    0    0
2    0    8    0    0    0    0    0    0    0    0  ...    0    0    0    0
3    0    0   10    0    0    0    0    0    0    0  ...    0    0    0    0
4    0    0    0    7    0    0    0    0    0    0  ...    0    0    0    0
5    0    0    0    0    9    0    0    0    0    0  ...    0    0    0    0

   145  146  147  148  149  150
1    0    0    0    0    0    0
2    0    0    0    0    0    0
3    0    0    0    0    0    0
4    0    0    0    0    0    0
5    0    0    0    0    0    0

[5 rows x 50 columns]
```

```
Since you watched Forrest Gump, you might also like:
Inception
The Dark Knight
The Matrix
The Godfather
The Shawshank Redemption
Pulp Fiction
The Lord of the Rings: The Return of the King
Star Wars: A New Hope
The Empire Strikes Back
The Dark Knight Rises
User with ID 69 does not exist.
```

Practical No: 06

Aim: Train a GAN for generalistic realistic images.

Code:

```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
from torchvision import datasets, transforms
import matplotlib.pyplot as plt
import numpy as np
# Device configuration
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
# Hyperparameters
latent_dim = 100
lr = 0.0002
beta1 = 0.5
beta2 = 0.999
num_epochs = 10
batch_size = 32
# Transforms for CIFAR-10
transform = transforms.Compose([
   transforms.ToTensor(),
   transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))  # normalize to [-1, 1]
])
# Load dataset
train_dataset = datasets.CIFAR10(root='./data', train=True,
                                 download=True, transform=transform)
dataloader = torch.utils.data.DataLoader(train_dataset, batch_size=batch_size, shuffle=True)


# --------------------------
# Generator Model (outputs 32x32)
```

```
# --------------------------
class Generator(nn.Module):
    def __init__(self, latent_dim):
        super(Generator, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(latent_dim, 256 * 4 * 4),
            nn.LeakyReLU(0.2),
            nn.Unflatten(1, (256, 4, 4)),  # Output: (256, 4, 4)

            nn.ConvTranspose2d(256, 128, kernel_size=4, stride=2, padding=1),  # (128, 8, 8)
            nn.BatchNorm2d(128),
            nn.LeakyReLU(0.2),

            nn.ConvTranspose2d(128, 64, kernel_size=4, stride=2, padding=1),   # (64, 16, 16)
            nn.BatchNorm2d(64),
            nn.LeakyReLU(0.2),

            nn.ConvTranspose2d(64, 3, kernel_size=4, stride=2, padding=1),     # (3, 32, 32)
            nn.Tanh()
        )
    def forward(self, z):
        return self.model(z)
# --------------------------
# Discriminator Model
# --------------------------
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.conv_layers = nn.Sequential(
            nn.Conv2d(3, 32, kernel_size=3, stride=2, padding=1),  # (32, 16, 16)
            nn.LeakyReLU(0.2),
```

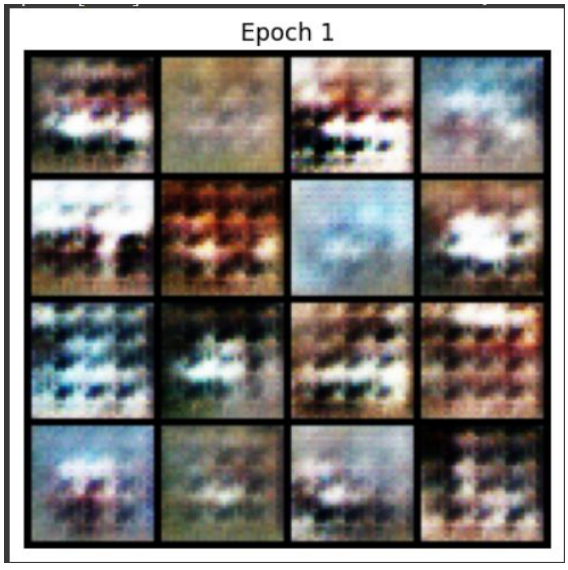
```
            nn.Dropout(0.25),
            nn.Conv2d(32, 64, kernel_size=3, stride=2, padding=1),  # (64, 8, 8)
            nn.BatchNorm2d(64, momentum=0.8),
            nn.LeakyReLU(0.2),
            nn.Dropout(0.25),
            nn.Conv2d(64, 128, kernel_size=3, stride=2, padding=1),  # (128, 4, 4)
            nn.BatchNorm2d(128, momentum=0.8),
            nn.LeakyReLU(0.2),
            nn.Dropout(0.25),
            nn.Conv2d(128, 256, kernel_size=3, stride=2, padding=1),  # (256, 2, 2)
            nn.BatchNorm2d(256, momentum=0.8),
            nn.LeakyReLU(0.2),
            nn.Dropout(0.25),
            nn.Flatten()
        )
        self.output_layer = nn.Sequential(
            nn.Linear(256 * 2 * 2, 1),
            nn.Sigmoid()
        )
    def forward(self, img):
        x = self.conv_layers(img)
        return self.output_layer(x)
# Initialize models
generator = Generator(latent_dim).to(device)
discriminator = Discriminator().to(device)
# Loss and optimizers
adversarial_loss = nn.BCELoss()
optimizer_G = optim.Adam(generator.parameters(), lr=lr, betas=(beta1, beta2))
optimizer_D = optim.Adam(discriminator.parameters(), lr=lr, betas=(beta1, beta2))
# --------------------------
# Training Loop
```

```
# --------------------------
for epoch in range(num_epochs):
    for i, (real_images, _) in enumerate(dataloader):
        real_images = real_images.to(device)
        batch_size = real_images.size(0)
        # Labels
        valid = torch.ones(batch_size, 1, device=device)
        fake = torch.zeros(batch_size, 1, device=device)
        # ---------------------
        #  Train Discriminator
        # ---------------------
        optimizer_D.zero_grad()
        # Generate fake images
        z = torch.randn(batch_size, latent_dim, device=device)
        fake_images = generator(z)
        # Discriminator loss
        real_loss = adversarial_loss(discriminator(real_images), valid)
        fake_loss = adversarial_loss(discriminator(fake_images.detach()), fake)
        d_loss = (real_loss + fake_loss) / 2
        d_loss.backward()
        optimizer_D.step()
        # -----------------
        #  Train Generator
        # -----------------
        optimizer_G.zero_grad()
        # Generate images again for generator update
        gen_images = generator(z)
        g_loss = adversarial_loss(discriminator(gen_images), valid)
        g_loss.backward()
        optimizer_G.step()
```

```
        # Logging
        if (i + 1) % 100 == 0:
            print(f"Epoch [{epoch+1}/{num_epochs}] Batch {i+1}/{len(dataloader)} "
                  f"D Loss: {d_loss.item():.4f}, G Loss: {g_loss.item():.4f}")

    # Show generated images at the end of each epoch
    with torch.no_grad():
        z = torch.randn(16, latent_dim, device=device)
        generated = generator(z).detach().cpu()
        grid = torchvision.utils.make_grid(generated, nrow=4, normalize=True)
        plt.imshow(np.transpose(grid, (1, 2, 0)))
        plt.title(f"Epoch {epoch+1}")
        plt.axis("off")
        plt.show()
```

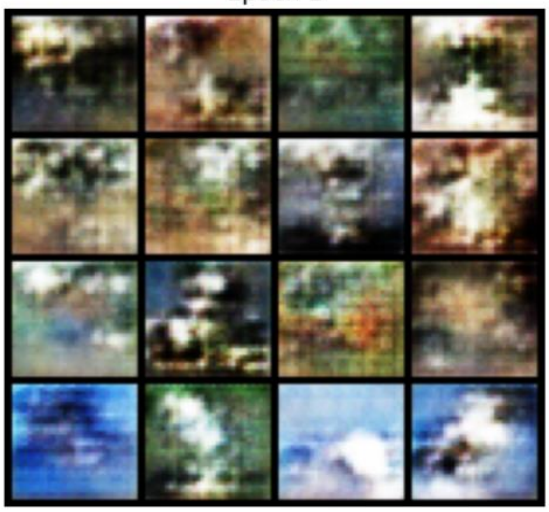
Output:

```
Epoch [1/10] Batch 100/1563 D Loss: 0.1698, G Loss: 2.8074
Epoch [1/10] Batch 200/1563 D Loss: 0.1348, G Loss: 3.9064
Epoch [1/10] Batch 300/1563 D Loss: 0.3971, G Loss: 2.0801
Epoch [1/10] Batch 400/1563 D Loss: 0.5394, G Loss: 1.6982
Epoch [1/10] Batch 500/1563 D Loss: 0.5161, G Loss: 1.3079
Epoch [1/10] Batch 600/1563 D Loss: 0.5959, G Loss: 1.3965
Epoch [1/10] Batch 700/1563 D Loss: 0.4265, G Loss: 1.7928
Epoch [1/10] Batch 800/1563 D Loss: 0.5240, G Loss: 1.5957
Epoch [1/10] Batch 900/1563 D Loss: 0.5112, G Loss: 1.5496
Epoch [1/10] Batch 1000/1563 D Loss: 0.5483, G Loss: 1.3919
Epoch [1/10] Batch 1100/1563 D Loss: 0.5170, G Loss: 1.7339
Epoch [1/10] Batch 1200/1563 D Loss: 0.4751, G Loss: 1.6253
Epoch [1/10] Batch 1300/1563 D Loss: 0.4237, G Loss: 2.0265
Epoch [1/10] Batch 1400/1563 D Loss: 0.3231, G Loss: 1.9558
Epoch [1/10] Batch 1500/1563 D Loss: 0.4115, G Loss: 1.9041
```



```
Epoch [2/10] Batch 100/1563 D Loss: 0.3273, G Loss: 2.0771
Epoch [2/10] Batch 200/1563 D Loss: 0.5364, G Loss: 1.5776
Epoch [2/10] Batch 300/1563 D Loss: 0.4152, G Loss: 1.6360
Epoch [2/10] Batch 400/1563 D Loss: 0.4637, G Loss: 2.0201
Epoch [2/10] Batch 500/1563 D Loss: 0.6245, G Loss: 2.5356
Epoch [2/10] Batch 600/1563 D Loss: 0.2864, G Loss: 1.4335
Epoch [2/10] Batch 700/1563 D Loss: 0.4777, G Loss: 1.7340
Epoch [2/10] Batch 800/1563 D Loss: 0.5429, G Loss: 1.6100
Epoch [2/10] Batch 900/1563 D Loss: 0.3829, G Loss: 1.6548
Epoch [2/10] Batch 1000/1563 D Loss: 0.3736, G Loss: 1.8646
Epoch [2/10] Batch 1100/1563 D Loss: 0.4682, G Loss: 1.8537
Epoch [2/10] Batch 1200/1563 D Loss: 0.2375, G Loss: 2.0316
Epoch [2/10] Batch 1300/1563 D Loss: 0.6056, G Loss: 1.0721
Epoch [2/10] Batch 1400/1563 D Loss: 0.6005, G Loss: 1.8418
Epoch [2/10] Batch 1500/1563 D Loss: 0.4574, G Loss: 1.1520
```

Epoch 2



```
Epoch [3/10] Batch 100/1563 D Loss: 0.4362, G Loss: 2.9203
Epoch [3/10] Batch 200/1563 D Loss: 0.4245, G Loss: 1.7363
Epoch [3/10] Batch 300/1563 D Loss: 0.3080, G Loss: 1.8949
Epoch [3/10] Batch 400/1563 D Loss: 0.3951, G Loss: 2.8099
Epoch [3/10] Batch 500/1563 D Loss: 0.3501, G Loss: 2.4604
Epoch [3/10] Batch 600/1563 D Loss: 0.3361, G Loss: 2.4960
Epoch [3/10] Batch 700/1563 D Loss: 0.3773, G Loss: 2.8686
Epoch [3/10] Batch 800/1563 D Loss: 0.2202, G Loss: 2.0665
Epoch [3/10] Batch 900/1563 D Loss: 0.5867, G Loss: 1.9605
Epoch [3/10] Batch 1000/1563 D Loss: 0.4792, G Loss: 2.6434
Epoch [3/10] Batch 1100/1563 D Loss: 0.5000, G Loss: 2.6198
Epoch [3/10] Batch 1200/1563 D Loss: 0.4200, G Loss: 2.1847
Epoch [3/10] Batch 1300/1563 D Loss: 0.4266, G Loss: 1.9429
Epoch [3/10] Batch 1400/1563 D Loss: 0.1708, G Loss: 1.9142
Epoch [3/10] Batch 1500/1563 D Loss: 0.4407, G Loss: 2.0310
```

Epoch 3

```
Epoch [4/10] Batch 100/1563 D Loss: 0.2250, G Loss: 2.1803
Epoch [4/10] Batch 200/1563 D Loss: 0.4257, G Loss: 2.1484
Epoch [4/10] Batch 300/1563 D Loss: 0.3093, G Loss: 2.8053
Epoch [4/10] Batch 400/1563 D Loss: 0.5112, G Loss: 2.7399
Epoch [4/10] Batch 500/1563 D Loss: 0.4484, G Loss: 1.4436
Epoch [4/10] Batch 600/1563 D Loss: 0.5044, G Loss: 2.8906
Epoch [4/10] Batch 700/1563 D Loss: 0.2543, G Loss: 3.1233
Epoch [4/10] Batch 800/1563 D Loss: 0.3366, G Loss: 2.6895
Epoch [4/10] Batch 900/1563 D Loss: 0.3679, G Loss: 2.8753
Epoch [4/10] Batch 1000/1563 D Loss: 0.2051, G Loss: 3.2505
Epoch [4/10] Batch 1100/1563 D Loss: 0.2136, G Loss: 2.4175
Epoch [4/10] Batch 1200/1563 D Loss: 0.1400, G Loss: 3.1768
Epoch [4/10] Batch 1300/1563 D Loss: 0.1931, G Loss: 1.9609
Epoch [4/10] Batch 1400/1563 D Loss: 0.3902, G Loss: 2.0245
Epoch [4/10] Batch 1500/1563 D Loss: 0.2281, G Loss: 2.9722
```

Epoch 4

```
Epoch [5/10] Batch 100/1563 D Loss: 0.2906, G Loss: 1.3987
Epoch [5/10] Batch 200/1563 D Loss: 0.4904, G Loss: 2.2220
Epoch [5/10] Batch 300/1563 D Loss: 0.3733, G Loss: 2.1196
Epoch [5/10] Batch 400/1563 D Loss: 0.2743, G Loss: 2.6520
Epoch [5/10] Batch 500/1563 D Loss: 0.2179, G Loss: 3.0810
Epoch [5/10] Batch 600/1563 D Loss: 0.3981, G Loss: 2.4686
Epoch [5/10] Batch 700/1563 D Loss: 0.2634, G Loss: 3.8444
Epoch [5/10] Batch 800/1563 D Loss: 0.5163, G Loss: 2.7508
Epoch [5/10] Batch 900/1563 D Loss: 0.4130, G Loss: 2.3221
Epoch [5/10] Batch 1000/1563 D Loss: 0.3187, G Loss: 2.2726
Epoch [5/10] Batch 1100/1563 D Loss: 0.1154, G Loss: 2.8672
Epoch [5/10] Batch 1200/1563 D Loss: 0.1523, G Loss: 2.1208
Epoch [5/10] Batch 1300/1563 D Loss: 0.1906, G Loss: 3.3981
Epoch [5/10] Batch 1400/1563 D Loss: 0.3479, G Loss: 2.2372
Epoch [5/10] Batch 1500/1563 D Loss: 0.2492, G Loss: 2.4063
```

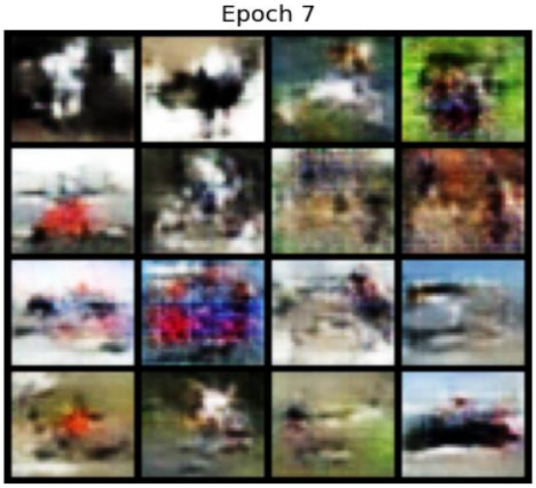
Epoch 5

```
Epoch [6/10] Batch 100/1563 D Loss: 0.2452, G Loss: 3.6229
Epoch [6/10] Batch 200/1563 D Loss: 0.3238, G Loss: 1.3243
Epoch [6/10] Batch 300/1563 D Loss: 0.3200, G Loss: 4.2746
Epoch [6/10] Batch 400/1563 D Loss: 0.3210, G Loss: 2.7640
Epoch [6/10] Batch 500/1563 D Loss: 0.0925, G Loss: 4.8075
Epoch [6/10] Batch 600/1563 D Loss: 0.2534, G Loss: 1.6835
Epoch [6/10] Batch 700/1563 D Loss: 0.1415, G Loss: 2.5755
Epoch [6/10] Batch 800/1563 D Loss: 0.4759, G Loss: 3.7021
Epoch [6/10] Batch 900/1563 D Loss: 0.1736, G Loss: 3.1443
Epoch [6/10] Batch 1000/1563 D Loss: 0.8756, G Loss: 0.9918
Epoch [6/10] Batch 1100/1563 D Loss: 0.3888, G Loss: 2.1457
Epoch [6/10] Batch 1200/1563 D Loss: 0.0773, G Loss: 2.3410
Epoch [6/10] Batch 1300/1563 D Loss: 0.3832, G Loss: 2.8590
Epoch [6/10] Batch 1400/1563 D Loss: 0.2319, G Loss: 2.4108
Epoch [6/10] Batch 1500/1563 D Loss: 1.1700, G Loss: 1.1916
```

Epoch 6

```
Epoch [7/10] Batch 100/1563 D Loss: 0.2585, G Loss: 4.6638
Epoch [7/10] Batch 200/1563 D Loss: 0.1078, G Loss: 4.0169
Epoch [7/10] Batch 300/1563 D Loss: 0.5158, G Loss: 1.9126
Epoch [7/10] Batch 400/1563 D Loss: 0.7062, G Loss: 2.3110
Epoch [7/10] Batch 500/1563 D Loss: 0.4278, G Loss: 0.8407
Epoch [7/10] Batch 600/1563 D Loss: 0.3140, G Loss: 2.1294
Epoch [7/10] Batch 700/1563 D Loss: 0.4805, G Loss: 2.3685
Epoch [7/10] Batch 800/1563 D Loss: 0.5331, G Loss: 1.9040
Epoch [7/10] Batch 900/1563 D Loss: 0.1843, G Loss: 1.9323
Epoch [7/10] Batch 1000/1563 D Loss: 0.4010, G Loss: 4.6805
Epoch [7/10] Batch 1100/1563 D Loss: 0.2247, G Loss: 3.2621
Epoch [7/10] Batch 1200/1563 D Loss: 0.1599, G Loss: 2.3717
Epoch [7/10] Batch 1300/1563 D Loss: 0.4457, G Loss: 1.9307
Epoch [7/10] Batch 1400/1563 D Loss: 0.2926, G Loss: 2.4467
Epoch [7/10] Batch 1500/1563 D Loss: 0.4885, G Loss: 2.1548
```

Epoch 7



```
Epoch [8/10] Batch 100/1563 D Loss: 0.2886, G Loss: 2.5581
Epoch [8/10] Batch 200/1563 D Loss: 0.3287, G Loss: 2.3091
Epoch [8/10] Batch 300/1563 D Loss: 0.1910, G Loss: 4.2200
Epoch [8/10] Batch 400/1563 D Loss: 0.2808, G Loss: 2.2404
Epoch [8/10] Batch 500/1563 D Loss: 0.2139, G Loss: 3.5320
Epoch [8/10] Batch 600/1563 D Loss: 0.1042, G Loss: 3.0996
Epoch [8/10] Batch 700/1563 D Loss: 0.2958, G Loss: 2.8935
Epoch [8/10] Batch 800/1563 D Loss: 0.3451, G Loss: 2.4741
Epoch [8/10] Batch 900/1563 D Loss: 0.2125, G Loss: 4.2078
Epoch [8/10] Batch 1000/1563 D Loss: 0.4781, G Loss: 2.1386
Epoch [8/10] Batch 1100/1563 D Loss: 0.1131, G Loss: 4.6437
Epoch [8/10] Batch 1200/1563 D Loss: 0.4500, G Loss: 1.6421
Epoch [8/10] Batch 1300/1563 D Loss: 0.1107, G Loss: 3.3952
Epoch [8/10] Batch 1400/1563 D Loss: 0.4007, G Loss: 2.3017
Epoch [8/10] Batch 1500/1563 D Loss: 0.2088, G Loss: 1.7227
```

Epoch 8

```
Epoch [9/10] Batch 100/1563 D Loss: 0.5759, G Loss: 1.6715
Epoch [9/10] Batch 200/1563 D Loss: 0.4395, G Loss: 1.6614
Epoch [9/10] Batch 300/1563 D Loss: 0.2304, G Loss: 2.4918
Epoch [9/10] Batch 400/1563 D Loss: 1.6342, G Loss: 3.5813
Epoch [9/10] Batch 500/1563 D Loss: 0.2886, G Loss: 2.1133
Epoch [9/10] Batch 600/1563 D Loss: 0.0951, G Loss: 0.9324
Epoch [9/10] Batch 700/1563 D Loss: 0.3729, G Loss: 1.5654
Epoch [9/10] Batch 800/1563 D Loss: 0.0586, G Loss: 3.3419
Epoch [9/10] Batch 900/1563 D Loss: 0.4054, G Loss: 4.6372
Epoch [9/10] Batch 1000/1563 D Loss: 0.3704, G Loss: 1.5484
Epoch [9/10] Batch 1100/1563 D Loss: 0.1682, G Loss: 4.3446
Epoch [9/10] Batch 1200/1563 D Loss: 0.1060, G Loss: 3.6518
Epoch [9/10] Batch 1300/1563 D Loss: 0.1262, G Loss: 2.3220
Epoch [9/10] Batch 1400/1563 D Loss: 0.5815, G Loss: 1.5987
Epoch [9/10] Batch 1500/1563 D Loss: 0.2327, G Loss: 2.1091
```

Epoch 9

```
Epoch [10/10] Batch 100/1563 D Loss: 0.7863, G Loss: 1.8610
Epoch [10/10] Batch 200/1563 D Loss: 0.0555, G Loss: 3.3590
Epoch [10/10] Batch 300/1563 D Loss: 0.2582, G Loss: 3.8949
Epoch [10/10] Batch 400/1563 D Loss: 0.2133, G Loss: 0.2133
Epoch [10/10] Batch 500/1563 D Loss: 0.1714, G Loss: 2.6700
Epoch [10/10] Batch 600/1563 D Loss: 0.1558, G Loss: 2.4205
Epoch [10/10] Batch 700/1563 D Loss: 0.2572, G Loss: 1.5890
Epoch [10/10] Batch 800/1563 D Loss: 0.6321, G Loss: 2.7579
Epoch [10/10] Batch 900/1563 D Loss: 0.4095, G Loss: 2.3359
Epoch [10/10] Batch 1000/1563 D Loss: 0.2808, G Loss: 1.7665
Epoch [10/10] Batch 1100/1563 D Loss: 0.4985, G Loss: 3.7579
Epoch [10/10] Batch 1200/1563 D Loss: 0.3623, G Loss: 3.2118
Epoch [10/10] Batch 1300/1563 D Loss: 0.4998, G Loss: 3.7606
Epoch [10/10] Batch 1400/1563 D Loss: 0.6441, G Loss: 2.2580
Epoch [10/10] Batch 1500/1563 D Loss: 0.3637, G Loss: 1.7455
```

Epoch 10