

Ex.No:01

Setting up the spyder IDE Environment and executing a python program

Aim:

To install and configure the Spyder IDE environment, and to execute a Python program.

Procedure:

Step 1: Installing Anaconda Distribution

Why Anaconda? Anaconda is a Python distribution that comes with many packages pre-installed (such as NumPy, SciPy, Pandas, Matplotlib, etc.), and it includes the Spyder IDE by default.

1. Download Anaconda:

- Go to the official Anaconda website: <https://www.anaconda.com/products/individual>
- Click on the “Download” button and choose the version suitable for your operating system (Windows, macOS, Linux).

2. Install Anaconda:

- Open the downloaded installer file.
- Follow the installation wizard instructions:
 - Select whether you want to install Anaconda for all users or just yourself.
 - Choose the installation location.
 - Add Anaconda to the system PATH (optional, but recommended for ease of use).
- Click "Install" to proceed, and wait for the installation to finish.

3. Verify Installation:

- Open **Anaconda Navigator** from your start menu (Windows) or applications folder (macOS).
- Alternatively, you can verify via the command line by typing:

conda --version

Step 2: Launching Spyder IDE

1. Open Anaconda Navigator:

- Once installed, search for "Anaconda Navigator" in your system and open it.
- Anaconda Navigator will provide a graphical user interface (GUI) for managing environments and applications.

2. Launch Spyder:

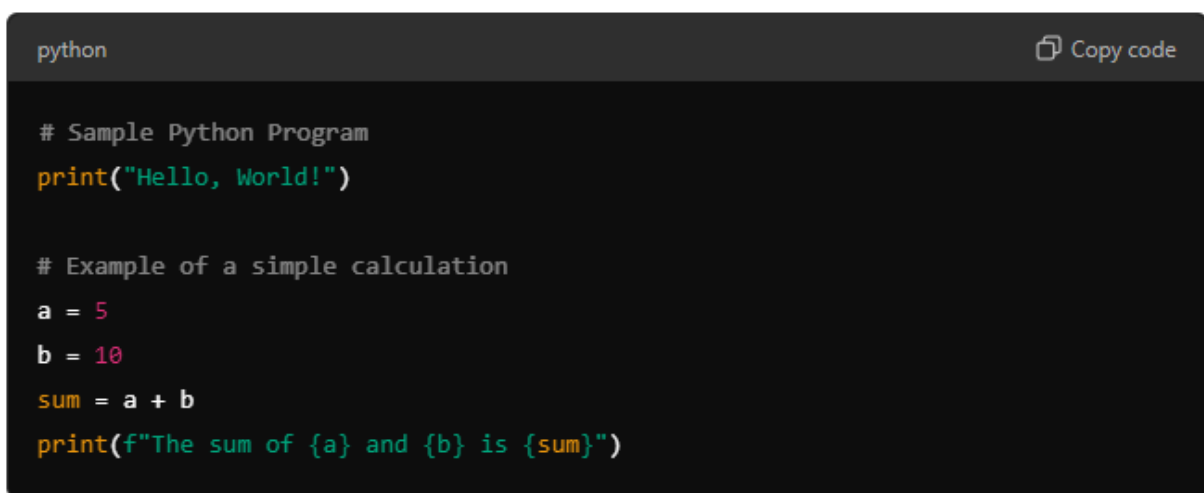
- In Anaconda Navigator, locate the Spyder icon.
- Click on “Launch” next to the Spyder icon to start the Spyder IDE.

Step 3: Writing and Executing a Python Program

Now that the environment is set up, let's write and execute a Python program.

1. Write a Python Program:

- Once Spyder is open, you will see the code editor in the center.
- Type the following sample Python code:

A screenshot of the Spyder Python IDE's code editor. The editor has a dark background with syntax-highlighted Python code. At the top left, the word 'python' is written in a small font. At the top right, there is a 'Copy code' button with a copy icon. The code itself consists of two parts: a simple print statement and a small calculation. The first part is a comment followed by a print statement that outputs 'Hello, World!'. The second part is a comment followed by three lines of code that assign values to variables 'a' and 'b', calculate their sum, and then print the result using an f-string.

```
python Copy code

# Sample Python Program
print("Hello, World!")

# Example of a simple calculation
a = 5
b = 10
sum = a + b
print(f"The sum of {a} and {b} is {sum}")
```

2. Save the Program:

- Click on the "Save" button in the toolbar or press Ctrl + S (Windows/Linux) or Cmd + S (Mac).
- Save the file with a .py extension, e.g., sample_program.py.

3. Execute the Python Program:

- To execute the program, click on the "Run" button in the toolbar or press F5.
- The console (typically at the bottom of the window) will display the output:

Hello, World!

The sum of 5 and 10 is 15

Step 4: Exploring Spyder Features

1. Variable Explorer:

- You can monitor the values of variables in your program using the **Variable Explorer**.

- It is located on the right side of the Spyder window and will show you the variables (e.g., a, b, sum) and their values.

2. **Help Pane:**

- Spyder provides an integrated help pane that gives access to documentation.
- Simply type a function or module name in the help pane to get quick documentation without leaving the IDE.

3. **Plots (Optional):**

- If your program involves plotting, Spyder can display plots directly in the interface.
- Try adding a simple plot using Matplotlib:

```
import matplotlib.pyplot as plt
x = [1, 2, 3, 4]
y = [10, 20, 25, 30]
plt.plot(x, y)
plt.show()
```

4. **Debugging:**

- Spyder has a built-in debugger that allows you to step through code, inspect variables, and fix issues.
- You can set breakpoints by clicking next to the line numbers and then running the script in debugging mode (press Ctrl + F5).

Step 5: Additional Configurations (Optional)

1. **Package Management:**

- You can install additional Python packages using Anaconda Navigator or via the terminal using:

```
conda install <package_name>
```

- Alternatively, use pip within the terminal:

```
pip install <package_name>
```

2. **Creating Virtual Environments (Optional but Recommended):**

- You can create isolated environments for different projects to avoid conflicts between packages.
- To create a new environment:

```
conda create --name myenv
```

- To activate it:

```
conda activate myenv
```

Output:

```
python Copy code
```

```
Hello, World!  
The sum of 5 and 10 is 15
```

Result:

This lab experiment helped you to install and set up the Spyder IDE using Anaconda, write and run a simple Python program, and explore some key features of the IDE like the Variable Explorer, console, and debugging tools. With the IDE set up, you can now proceed with more complex Python programming tasks efficiently.

Ex.no:8

Generative Adversarial Networks for image generation

Aim:

To implement a Generative Adversarial Network (GAN) for image generation, where the model learns to generate realistic images by training two neural networks: a generator and a discriminator, in an adversarial setting.

Procedure:

1. Install Required Libraries:

- Ensure the necessary libraries such as Keras, TensorFlow, or PyTorch are installed:
 - `pip install keras`
 - `pip install tensorflow`
- These will be used to create and train the GAN model.

2. GAN Architecture Overview:

- The GAN model consists of two neural networks:
 - **Generator:** Takes random noise as input and generates images.
 - **Discriminator:** Takes an image as input and classifies it as real or fake.
- Both networks are trained in an adversarial setup: the generator improves in creating realistic images, while the discriminator improves in detecting generated images.

3. Load and Preprocess Dataset:

- Use an image dataset like MNIST (for generating handwritten digits) or CIFAR-10 (for generating more complex images).
- Preprocess the dataset by normalizing the pixel values between -1 and 1 to match the generator's output.

```
from keras.datasets import mnist
(X_train, _), (_, _) = mnist.load_data()
X_train = X_train.astype('float32') / 127.5 - 1
X_train = np.expand_dims(X_train, axis=-1)
```

4. Build the Generator Network:

- The generator starts with a random noise vector and produces an image. Use dense and transposed convolution layers to upsample the noise into a meaningful image.

```
from keras.models import Sequential
```

```
from keras.layers import Dense, Reshape, Conv2DTranspose, BatchNormalization, ReLU
```

```
generator = Sequential()
```

```
generator.add(Dense(256, input_dim=100))
```

```
generator.add(Reshape((7, 7, 256)))
```

```
generator.add(Conv2DTranspose(128, (3, 3), strides=(2, 2), padding='same'))
```

```
generator.add(BatchNormalization())
```

```
generator.add(ReLU())
```

```
generator.add(Conv2DTranspose(1, (3, 3), strides=(2, 2), padding='same', activation='tanh'))
```

5. Build the Discriminator Network:

- The discriminator takes an image as input and outputs a probability (real or fake). Use convolutional layers to downsample the image and extract features.

```
from keras.layers import Conv2D, LeakyReLU, Flatten
```

```
discriminator = Sequential()
```

```
discriminator.add(Conv2D(64, (3, 3), padding='same', input_shape=(28, 28, 1)))
```

```
discriminator.add(LeakyReLU(alpha=0.2))
```

```
discriminator.add(Conv2D(128, (3, 3), strides=(2, 2)))
```

```
discriminator.add(LeakyReLU(alpha=0.2))
```

```
discriminator.add(Flatten())
```

```
discriminator.add(Dense(1, activation='sigmoid'))
```

6. Compile the Discriminator:

- Compile the discriminator using binary crossentropy loss, as it performs binary classification (real vs. fake).

```
discriminator.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
```

7. Build and Compile the GAN:

- The GAN combines the generator and discriminator. When training the generator, the discriminator's weights are frozen. The goal is for the generator to fool the discriminator.

```
from keras.models import Model
from keras.layers import Input

discriminator.trainable = False
gan_input = Input(shape=(100,))
generated_image = generator(gan_input)
gan_output = discriminator(generated_image)
gan = Model(gan_input, gan_output)
gan.compile(optimizer='adam', loss='binary_crossentropy')
```

8. Train the GAN:

- Train the GAN by alternating between training the discriminator and the generator:
 - First, train the discriminator on real and fake images.
 - Then, train the generator to create images that can fool the discriminator.

```
import numpy as np

for epoch in range(epochs):
    # Train discriminator on real and fake images
    real_images = X_train[np.random.randint(0, X_train.shape[0], batch_size)]
    noise = np.random.normal(0, 1, (batch_size, 100))
    fake_images = generator.predict(noise)

    real_labels = np.ones((batch_size, 1))
    fake_labels = np.zeros((batch_size, 1))

    discriminator.train_on_batch(real_images, real_labels)
    discriminator.train_on_batch(fake_images, fake_labels)

    # Train generator
    noise = np.random.normal(0, 1, (batch_size, 100))
    gan.train_on_batch(noise, real_labels)
```

9. Generate and Visualize Images:

- After training, generate new images from random noise and visualize them to evaluate how well the generator has learned to create realistic images.

program:

```
import torch

from torch import nn

import math
import matplotlib.pyplot as plt
import torchvision
import torchvision.transforms as transforms

torch.manual_seed(111)
device = ""
if torch.cuda.is_available():
    device = torch.device("cuda")
else:
    device = torch.device("cpu")

transform = transforms.Compose(
    [transforms.ToTensor(), transforms.Normalize((0.5,), (0.5,))]
)

train_set = torchvision.datasets.MNIST(
    root=".", train=True, download=True, transform=transform
)

batch_size = 32

train_loader = torch.utils.data.DataLoader(
    train_set, batch_size=batch_size, shuffle=True
)
```



```

real_samples, mnist_labels = next(iter(train_loader))
for i in range(16):
    ax = plt.subplot(4, 4, i + 1)
    plt.imshow(real_samples[i].reshape(28, 28),
               cmap="gray_r")
    plt.xticks([])
    plt.yticks([])

```

```

class Discriminator(nn.Module):
    def __init__(self):
        super().__init__()
        self.model = nn.Sequential(
            nn.Linear(784, 1024),
            nn.ReLU(),
            nn.Dropout(0.3),
            nn.Linear(1024, 512),
            nn.ReLU(),
            nn.Dropout(0.3),
            nn.Linear(512, 256),
            nn.ReLU(),
            nn.Dropout(0.3),
            nn.Linear(256, 1),
            nn.Sigmoid(),
        )

    def forward(self, x):
        x = x.view(x.size(0), 784)
        output = self.model(x)
        return output

discriminator = Discriminator().to(device=device)

class Generator(nn.Module):
    def __init__(self):
        super().__init__()
        self.model = nn.Sequential(
            nn.Linear(100, 256),
            nn.ReLU(),

```

```

nn.Linear(256, 512),

nn.ReLU(),

nn.Linear(512, 1024),

nn.ReLU(),


nn.Linear(1024, 784),
nn.Tanh(),
)

def forward(self, x):

output = self.model(x)

output = output.view(x.size(0), 1, 28, 28)
return output

generator = Generator().to(device=device)
lr = 0.0001
num_epochs = 50
loss_function = nn.BCELoss()

optimizer_discriminator = torch.optim.Adam(discriminator.parameters(), lr=lr)
optimizer_generator = torch.optim.Adam(generator.parameters(), lr=lr)

for epoch in range(num_epochs):

for n, (real_samples, mnist_labels) in enumerate(train_loader): # Data for
training the discriminator
real_samples = real_samples.to(device=device)
real_samples_labels = torch.ones((batch_size, 1)).to(
device=device
)

latent_space_samples = torch.randn((batch_size, 100)).to(
device=device

```

```

)

generated_samples = generator(latent_space_samples)
generated_samples_labels = torch.zeros((batch_size, 1)).to(
device=device
)

all_samples = torch.cat((real_samples, generated_samples))
all_samples_labels = torch.cat(
(real_samples_labels, generated_samples_labels)

)

# Training the discriminator
discriminator.zero_grad()
output_discriminator = discriminator(all_samples)
loss_discriminator = loss_function( output_discriminator,
all_samples_labels
)

loss_discriminator.backward()
optimizer_discriminator.step()


# Data for training the generator

latent_space_samples = torch.randn((batch_size, 100)).to(
device=device
)

# Training the generator
generator.zero_grad()
generated_samples = generator(latent_space_samples)
output_discriminator_generated = discriminator(generated_samples)
loss_generator = loss_function(

```

```
output_discriminator_generated, real_samples_labels
```

```
)
```

```
loss_generator.backward()
```

```
optimizer_generator.step()#
```

```
Show loss
```

```
if n == batch_size - 1:
```

```
print(f'Epoch: {epoch} Loss D.: {loss_discriminator}')
```

```
print(f'Epoch: {epoch} Loss G.: {loss_generator}')
```

```
latent_space_samples = torch.randn(batch_size, 100).to(device=device)
```

```
generated_samples = generator(latent_space_samples)
```

```
generated_samples = generated_samples.cpu().detach()for i in
```

```
range(16):
```

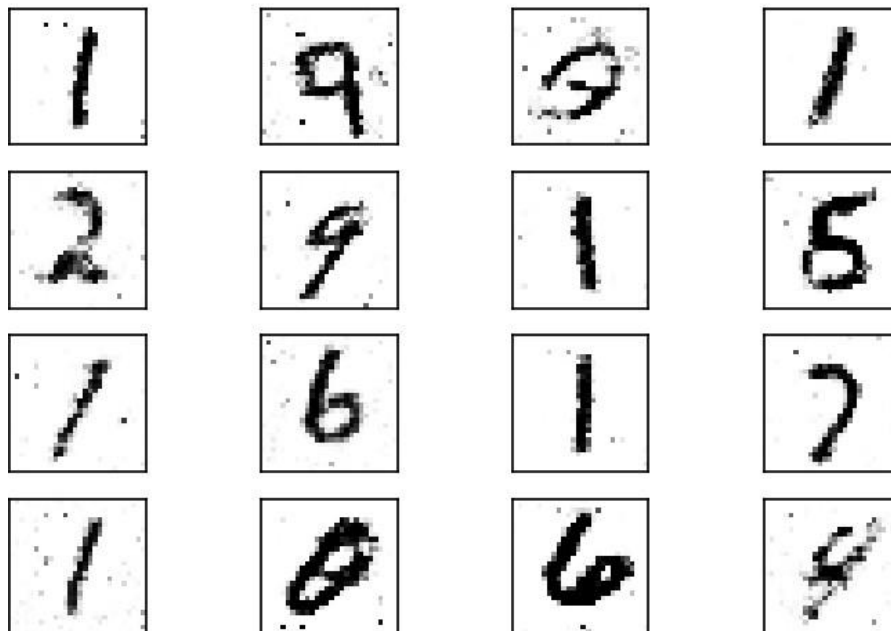
```
ax = plt.subplot(4, 4, i + 1) plt.imshow(generated_samples[i].reshape(28, 28),
```

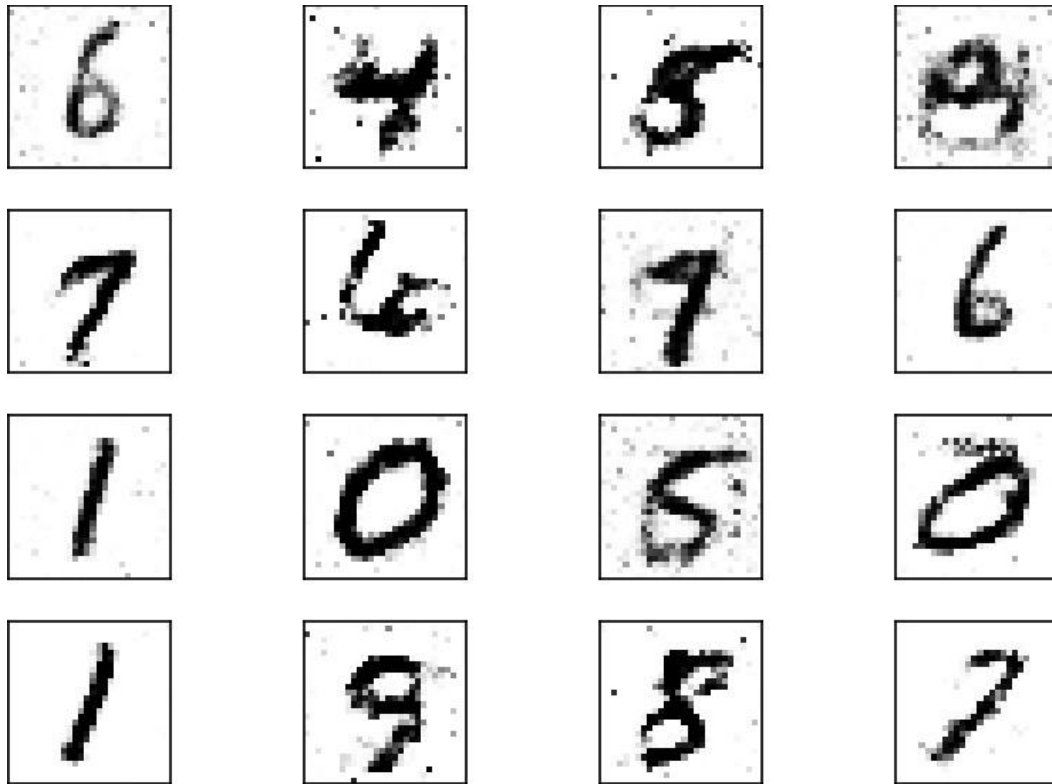
```
cmap="gray_r")plt.xticks([])
```

```
plt.yticks([])
```

OUTPUT:

After 46 epoch(s)



**Result:**

The Generative Adversarial Network (GAN) was successfully implemented for image generation. The generator learned to create realistic images through adversarial training, where it competes with the discriminator. This experiment demonstrates the effectiveness of GANs in generating synthetic images that resemble real data, showcasing the potential of GANs in applications such as image synthesis, style transfer, and data augmentation.