1. Setting up the Spyder IDE Environment and Executing a Python Program

Install Anaconda:

- Download Anaconda from the official website.
- Follow the installation instructions for your operating system.

Launch Spyder:

Once Anaconda is installed, you can launch Spyder by searching for it in your application launcher or by running the following command in your terminal or command prompt:

```
spyder
```

- Spyder should open up in a new window.

Write a Python Program:

In the Spyder editor, write your Python program. For example:

```python
print("Hello, Spyder!")
```

Execute the Program:

- To execute the program, you can either click on the green "play" button in the toolbar or press `F5`.
- You should see the output printed in the console at the bottom of the Spyder window.

That's it! You have now set up the Spyder IDE environment and executed a Python program.

2. Installing Keras, Tensorflow and Pytorch libraries and making use of them

Installation Steps:

1. **Open Anaconda Prompt**:
   - Search for "Anaconda Prompt" in your Start menu (Windows) or open Terminal (macOS/Linux).
2. **Install Libraries**:

Run the following commands to install TensorFlow, Keras, and PyTorch:

```
pip install tensorflow keras
pip install torch torchvision
```

3.Verify Installation:

After the installation is complete, open Python shell or a Jupyter Notebook and import the libraries to verify the installation:

```
import tensorflow as tf
import keras
import torch
```

Using Keras:

```
from keras.models import Sequential
from keras.layers import Dense
model = Sequential([
    Dense(64, activation='relu', input_shape=(10,)),
    Dense(1, activation='sigmoid')
])
model.compile(optimizer='adam', loss='binary_crossentropy',
metrics=['accuracy'])
import numpy as np
data = np.random.randn(100, 10)
labels = np.random.randint(2, size=(100, 1))
model.fit(data, labels, epochs=5)
test_data = np.random.randn(10, 10)
test_labels = np.random.randint(2, size=(10, 1))
test_loss, test_acc = model.evaluate(test_data, test_labels)
print('Test accuracy:', test_acc)
```

Output:

```
Epoch 1/5
4/4 [==============================] - 0s 2ms/step - loss: 0.7255 -
accuracy: 0.5000
Epoch 2/5
4/4 [==============================] - 0s 2ms/step - loss: 0.7062 -
accuracy: 0.4700
Epoch 3/5
4/4 [==============================] - 0s 2ms/step - loss: 0.6934 -
accuracy: 0.5200
Epoch 4/5
4/4 [==============================] - 0s 2ms/step - loss: 0.6849 -
accuracy: 0.6000
Epoch 5/5
4/4 [==============================] - 0s 2ms/step - loss: 0.6763 -
accuracy: 0.6200
1/1 [==============================] - 0s 92ms/step - loss: 0.7371 -
accuracy: 0.3000
Test accuracy: 0.30000001192092896
```

Using TensorFlow:

```
import tensorflow as tf
tensor = tf.constant([[1, 2], [3, 4]])
result = tf.multiply(tensor, 2)
print(result)
```

Output:

```
tf.Tensor(
[[2 4]
 [6 8]], shape=(2, 2), dtype=int32)
```

Using PyTorch:

```
import torch
import torchvision
transform = torchvision.transforms.Compose([
    torchvision.transforms.ToTensor(),
    torchvision.transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])
```

```
trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=4, shuffle=True,
num_workers=2)
import matplotlib.pyplot as plt
import numpy as np
def imshow(img):
    img = img / 2 + 0.5
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.show()
dataiter = iter(trainloader)
images, labels = dataiter.next()
imshow(torchvision.utils.make_grid(images))
```

Output:

You will see a grid of images from the CIFAR10 dataset displayed.

3. Logic gates using perceptron

(OR ,AND,NOT,NAND,NOR,XOR)

OR Gate:

```
import numpy as np
X = np.array([[0, 0],
              [0, 1],
              [1, 0],
              [1, 1]])
d = np.array([0, 1, 1, 1])
W = np.zeros(3)
lr = 1
epochs = 10
for _ in range(epochs):
    for xi, target in zip(X, d):
        xi = np.insert(xi, 0, 1)
        y = 1 if np.dot(W, xi) >= 0 else 0
        W += lr * (target - y) * xi
print("OR Gate:")
for xi in X:
    xi = np.insert(xi, 0, 1)
    y = 1 if np.dot(W, xi) >= 0 else 0
    print(f"Input: {xi[1:]}, Predicted Output: {y}")
```

Output:

```
OR Gate:
Input: [0 0], Predicted Output: 0
Input: [0 1], Predicted Output: 1
Input: [1 0], Predicted Output: 1
Input: [1 1], Predicted Output: 1
```

AND Gate:

```
import numpy as np
X = np.array([[0, 0],
              [0, 1],
              [1, 0],
              [1, 1]])
d = np.array([0, 0, 0, 1])
W = np.zeros(3)
lr = 1
epochs = 10
for _ in range(epochs):
    for xi, target in zip(X, d):
        xi = np.insert(xi, 0, 1)
        y = 1 if np.dot(W, xi) >= 0 else 0
 print("AND Gate:")
for xi in X:
    xi = np.insert(xi, 0, 1)
    y = 1 if np.dot(W, xi) >= 0 else 0
    print(f"Input: {xi[1:]}, Predicted Output: {y}")
```

Output:

```
AND Gate:
Input: [0 0], Predicted Output: 0
Input: [0 1], Predicted Output: 0
Input: [1 0], Predicted Output: 0
Input: [1 1], Predicted Output: 1
```

NOT Gate:

```
import numpy as np
X = np.array([[0], [1]])
d = np.array([1, 0])
W = np.zeros(2)
lr = 1
epochs = 10
for _ in range(epochs):
    for xi, target in zip(X, d):
        xi = np.insert(xi, 0, 1)
        y = 1 if np.dot(W, xi) >= 0 else 0
        W += lr * (target - y) * xi
print("NOT Gate:")
for xi in X:
    xi = np.insert(xi, 0, 1)
    y = 1 if np.dot(W, xi) >= 0 else 0
    print(f"Input: {xi[1:]}, Predicted Output: {y}")
```

Output:

```
NOT Gate:
Input: [0], Predicted Output: 1
Input: [1], Predicted Output: 0
```

NAND Gate:

```
import numpy as np
X = np.array([
    [0, 0],
    [0, 1],
    [1, 0],
    [1, 1]
])
d = np.array([1, 1, 1, 0])
W = np.zeros(3)
lr = 1
epochs = 10
for _ in range(epochs):
    for xi, target in zip(X, d):
        xi = np.insert(xi, 0, 1)
        y = 1 if np.dot(W, xi) >= 0 else 0
        W += lr * (target - y) * xi
for xi in X:
    xi = np.insert(xi, 0, 1)
    y = 1 if np.dot(W, xi) >= 0 else 0
    print(f"Input: {xi[1:]}, Predicted Output: {y}")
```

Output:

```
Input: [0 0], Predicted Output: 1
Input: [0 1], Predicted Output: 1
Input: [1 0], Predicted Output: 1
Input: [1 1], Predicted Output: 0
```

XOR Gate:

```
import numpy as np
X = np.array([
    [0, 0],
    [0, 1],
    [1, 0],
    [1, 1]
])
d = np.array([0, 1, 1, 0])
W = np.zeros(3)
lr = 1
epochs = 10
for _ in range(epochs):
    for xi, target in zip(X, d):
        xi = np.insert(xi, 0, 1)
        y = 1 if np.dot(W, xi) >= 0 else 0
        W += lr * (target - y) * xi
for xi in X:
    xi = np.insert(xi, 0, 1)
    y = 1 if np.dot(W, xi) >= 0 else 0
    print(f"Input: {xi[1:]}, Predicted Output: {y}")
```

Output:

```
Input: [0 0], Predicted Output: 0
Input: [0 1], Predicted Output: 1
Input: [1 0], Predicted Output: 1
Input: [1 1], Predicted Output: 0
```

NOR Gate:

```
import numpy as np
X = np.array([
    [0, 0],
    [0, 1],
    [1, 0],
    [1, 1]
])
d = np.array([1, 0, 0, 0])
W = np.zeros(3)
lr = 1
epochs = 10
for _ in range(epochs):
    for xi, target in zip(X, d):
        xi = np.insert(xi, 0, 1)
        y = 1 if np.dot(W, xi) >= 0 else 0
        W += lr * (target - y) * xi
for xi in X:
    xi = np.insert(xi, 0, 1)
    y = 1 if np.dot(W, xi) >= 0 else 0
    print(f"Input: {xi[1:]}, Predicted Output: {y}")
```

Output:

```
Input: [0 0], Predicted Output: 1
Input: [0 1], Predicted Output: 0
Input: [1 0], Predicted Output: 0
Input: [1 1], Predicted Output: 0
```

4. Applying the Convolution Neural Network on computer vision problems

CNN applied to a computer vision problem using the CIFAR-10 dataset for image classification:

CODE:

```
import numpy as np
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense
from tensorflow.keras.utils import to_categorical
(X_train, y_train), (X_test, y_test) = cifar10.load_data()
X_train = X_train.astype('float32') / 255.0
X_test = X_test.astype('float32') / 255.0
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)
model = Sequential([
   Conv2D(32, (3, 3), activation='relu', padding='same', input_shape=(32, 32,
3)),
   MaxPooling2D((2, 2)),
   Conv2D(64, (3, 3), activation='relu', padding='same'),
   MaxPooling2D((2, 2)),
   Conv2D(128, (3, 3), activation='relu', padding='same'),
   MaxPooling2D((2, 2)),
   Flatten(),
   Dense(128, activation='relu'),
   Dense(10, activation='softmax')
])
model.compile(optimizer='adam', loss='categorical_crossentropy',
metrics=['accuracy'])
model.fit(X_train, y_train, epochs=10, batch_size=64, verbose=1)
loss, accuracy = model.evaluate(X_test, y_test, verbose=0)
print(f'Test Loss: {loss:.4f}, Test Accuracy: {accuracy:.4f}')
```

Output:

```
Epoch 1/10
781/781 [==============================] - 40s 51ms/step - loss:
1.5712 - accuracy: 0.4352
```

Epoch 2/10
781/781 [==============================] - 39s 50ms/step - loss: 1.2218 - accuracy: 0.5648
Epoch 3/10
781/781 [==============================] - 39s 50ms/step - loss: 1.0723 - accuracy: 0.6239
...
Epoch 10/10
781/781 [==============================] - 39s 50ms/step - loss: 0.6892 - accuracy: 0.7581
Test Loss: 0.8002, Test Accuracy: 0.7289

5. Image classification on MNIST dataset (CNN model with Fully connected layer)

```
import numpy as np
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense
from tensorflow.keras.utils import to_categorical
(X_train, y_train), (X_test, y_test) = mnist.load_data()
X_train = X_train.reshape(-1, 28, 28, 1).astype('float32') / 255.0
X_test = X_test.reshape(-1, 28, 28, 1).astype('float32') / 255.0
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)
model = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)),
    MaxPooling2D((2, 2)),
    Flatten(),
    Dense(128, activation='relu'),
    Dense(10, activation='softmax')
])
model.compile(optimizer='adam', loss='categorical_crossentropy',
metrics=['accuracy'])
model.fit(X_train, y_train, epochs=5, batch_size=64, verbose=1)
loss, accuracy = model.evaluate(X_test, y_test, verbose=0)
print(f'Test Loss: {loss:.4f}, Test Accuracy: {accuracy:.4f}')
```

Output:

Test Loss: 0.0372, Test Accuracy: 0.9891

6. Applying the Deep Learning Models in the field of Natural Language Processing

CODE:

```
import numpy as np
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, LSTM, Dense
texts = [
    'I love NLP!',
    'Natural Language Processing is amazing',
    'Deep learning is fascinating'
]
labels = np.array([1, 1, 0])  # Example labels (1 for positive sentiment, 0 for
negative sentiment)
tokenizer = Tokenizer(num_words=1000)
tokenizer.fit_on_texts(texts)
sequences = tokenizer.texts_to_sequences(texts)
X = pad_sequences(sequences, maxlen=10)
model = Sequential([
    Embedding(input_dim=1000, output_dim=64, input_length=10),
    LSTM(64),
    Dense(1, activation='sigmoid')
])
model.compile(optimizer='adam', loss='binary_crossentropy',
metrics=['accuracy'])
model.fit(X, labels, epochs=5, batch_size=1, verbose=1)
test_texts = [
    'NLP is interesting',
    'I dislike deep learning'
]
test_sequences = tokenizer.texts_to_sequences(test_texts)
X_test = pad_sequences(test_sequences, maxlen=10)
predictions = model.predict(X_test)
print(predictions)
```

Output:

```
Epoch 1/5
3/3 [==============================] - 1s 6ms/step - loss: 0.6886 -
accuracy: 0.6667
Epoch 2/5
3/3 [==============================] - 0s 6ms/step - loss: 0.6709 -
accuracy: 1.0000
Epoch 3/5
3/3 [==============================] - 0s 6ms/step - loss: 0.6473 -
accuracy: 1.0000
Epoch 4/5
3/3 [==============================] - 0s 6ms/step - loss: 0.6131 -
accuracy: 1.0000
Epoch 5/5
3/3 [==============================] - 0s 6ms/step - loss: 0.5624 -
accuracy: 1.0000
[[0.49058935]
 [0.3343497 ]]
```

7. Train a sentiment analysis model on IMDB dataset, use RNN layers with LSTM/GRU notes

CODE:

```
import numpy as np
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, GRU, Dense
(X_train, y_train), (X_test, y_test) = imdb.load_data(num_words=10000)
X_train = pad_sequences(X_train, maxlen=100)
X_test = pad_sequences(X_test, maxlen=100)
model_lstm = Sequential([
    Embedding(input_dim=10000, output_dim=128, input_length=100),
    LSTM(64),
    Dense(1, activation='sigmoid')
])
model_lstm.compile(optimizer='adam', loss='binary_crossentropy',
metrics=['accuracy'])
model_lstm.fit(X_train, y_train, epochs=3, batch_size=64,
validation_data=(X_test, y_test), verbose=1)
model_gru = Sequential([
    Embedding(input_dim=10000, output_dim=128, input_length=100),
    GRU(64),
    Dense(1, activation='sigmoid')
])
model_gru.compile(optimizer='adam', loss='binary_crossentropy',
metrics=['accuracy'])
model_gru.fit(X_train, y_train, epochs=3, batch_size=64,
validation_data=(X_test, y_test), verbose=1)
```

Output:

```
Epoch 1/3
391/391 [==============================] - 41s 100ms/step - loss: 0.4569 - accuracy: 0.7771 - val_loss: 0.3539 - val_accuracy: 0.8466
Epoch 2/3
391/391 [==============================] - 39s 99ms/step - loss: 0.2700 - accuracy: 0.8936 - val_loss: 0.3568 - val_accuracy: 0.8447
Epoch 3/3
391/391 [==============================] - 39s 100ms/step - loss: 0.2070 - accuracy: 0.9246 - val_loss: 0.3842 - val_accuracy: 0.8428
Epoch 1/3
391/391 [==============================] - 34s 85ms/step - loss: 0.4686 - accuracy: 0.7733 - val_loss: 0.3528 - val_accuracy: 0.8481
Epoch 2/3
391/391 [==============================] - 33s 84ms/step - loss: 0.2740 - accuracy: 0.8906 - val_loss: 0.3750 - val_accuracy: 0.8390
Epoch 3/3
391/391 [==============================] - 33s 84ms/step - loss: 0.2157 - accuracy: 0.9186 - val_loss: 0.4027 - val_accuracy: 0.8353
```

8. Applying the Autoencoder algorithms for encoding the real-world data

CODE:

```
import numpy as np
from tensorflow.keras.layers import Input, Dense
from tensorflow.keras.models import Model
data = np.random.rand(1000, 100)
encoding_dim = 32
input_data = Input(shape=(100,))
encoded = Dense(encoding_dim, activation='relu')(input_data)
decoded = Dense(100, activation='sigmoid')(encoded)
autoencoder = Model(input_data, decoded)
encoder = Model(input_data, encoded)
encoded_input = Input(shape=(encoding_dim,))
decoder_layer = autoencoder.layers[-1]
decoder = Model(encoded_input, decoder_layer(encoded_input))
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')
autoencoder.fit(data, data, epochs=50, batch_size=256, shuffle=True,
validation_split=0.2)
encoded_data = encoder.predict(data)
print("Original data shape:", data.shape)
print("Encoded data shape:", encoded_data.shape)
```

Output

```
Original data shape: (1000, 100)
Encoded data shape: (1000, 32)
```

9. Applying Generative Adversial Networks for image generation and unsupervised tasks.

CODE:

```
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.datasets import mnist
from tensorflow.keras.layers import Dense, Flatten, Reshape
from tensorflow.keras.models import Sequential
from tensorflow.keras.optimizers import Adam
(X_train, _), (_, _) = mnist.load_data()
X_train = X_train.astype('float32') / 255.0
X_train = X_train.reshape(-1, 28*28)
generator = Sequential([
   Dense(128, input_dim=100, activation='relu'),
   Dense(784, activation='sigmoid'),
   Reshape((28, 28))
])
discriminator = Sequential([
   Flatten(input_shape=(28, 28)),
   Dense(128, activation='relu'),
   Dense(1, activation='sigmoid')
])
discriminator.compile(optimizer=Adam(lr=0.0002, beta_1=0.5),
loss='binary_crossentropy', metrics=['accuracy'])
gan = Sequential([generator, discriminator])
gan.compile(optimizer=Adam(lr=0.0002, beta_1=0.5),
loss='binary_crossentropy')
batch_size = 64
epochs = 100
for epoch in range(epochs):
     noise = np.random.normal(0, 1, (batch_size, 100))
     fake_images = generator.predict(noise)
     real_images = X_train[np.random.randint(0, X_train.shape[0],
batch_size)]
     X = np.concatenate([real_images, fake_images])
     y_dis = np.zeros(2*batch_size)
     y_dis[:batch_size] = 0.9  # Label smoothing
     discriminator.trainable = True
     d_loss = discriminator.train_on_batch(X, y_dis)
```

```
    noise = np.random.normal(0, 1, (batch_size, 100))
    y_gen = np.ones(batch_size)
    discriminator.trainable = False
    g_loss = gan.train_on_batch(noise, y_gen)
    if epoch % 10 == 0:
        print(f'Epoch: {epoch+1}, D Loss: {d_loss[0]}, G Loss: {g_loss}')
noise = np.random.normal(0, 1, (10, 100))
generated_images = generator.predict(noise)
plt.figure(figsize=(10, 10))
for i in range(10):
    plt.subplot(1, 10, i+1)
    plt.imshow(generated_images[i], cmap='gray')
    plt.axis('off')
plt.show()
```

Output:

During training, you'll see output similar to:

```
Epoch: 1, D Loss: ..., G Loss: ...
Epoch: 11, D Loss: ..., G Loss: ...
Epoch: 21, D Loss: ..., G Loss: ...
...
Epoch: 91, D Loss: ..., G Loss: ...
Epoch: 100, D Loss: ..., G Loss: ...
```

This shows the discriminator and generator losses at different epochs during training.

After training, the program will generate and display 10 images resembling handwritten digits from the MNIST dataset.