DEEP Learning R18 Jntuh LAB Manual

Deep Leaning (Jawaharlal Nehru Technological University, Hyderabad)

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

AIM: Setting up the Spyder IDE Environment and Executing a Python Program

DESCRIPTION:

To set up the Spyder IDE (Integrated Development Environment) and execute a Python program, follow these steps:

1. Install Python	Download and install Python from official website: https://www.python.org/downloads/
2. Install Spyder	Anaconda, a popular Python distribution for data science, includes Spyder and pre-installed packages for scientific computing and analysis. Download from the official website: https://www.anaconda.com/products/individual and follow installation instructions.
3. Launch Spyder	Launch Spyder IDE after installing Anaconda.
4. Create or open a Python file	In Spyder, you can create a new Python file or open an existing one. To create a new file, go to "File" > "New File" or use the keyboard shortcut Ctrl+N (Command+N on Mac). Alternatively, you can open an existing Python file by going to "File" > "Open" or using the keyboard shortcut Ctrl+O (Command+O on Mac).
5. Write your Python code	Write Python code in editor window for simple program. For example, print("Hello, World!")
6. Save the Python file	Save your Python file with the ".py" extension, for example, "hello.py". You can do this by going to "File" > "Save" or using the keyboard shortcut Ctrl+S (Command+S on Mac).
7. Execute the Python program	To execute your Python program, go to "Run" > "Run" or press the F5 key. The output of your program will appear in the console pane at the bottom of the Spyder IDE.

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

AIM:

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

DESCRIPTION:

To install Keras, TensorFlow, and PyTorch libraries and make use of them, you can follow the steps below:

1. Install Python and pip (if not already installed)	Ensure Python is installed and download the latest version from the official website: https://www.python.org/downloads/ Python comes pre-installed with pip; install if unavailable via pip website: https://pip.pypa.io/en/stable/installing/
2. Install TensorFlow	Open a command prompt or terminal and run the following command to install TensorFlow using pip: pip install tensorflow
3. Install Keras	Keras integrated into TensorFlow, ensuring automatic installation during installation. However, you can explicitly install Keras using pip: pip install keras
4. Install PyTorch	Install PyTorch by visiting official website, selecting appropriate command based on system configuration: https://pytorch.org/get-started/locally/ For example, to install the CPU-only version of PyTorch using pip, you can run: pip install torch torchvision
5. Verify installations	After installing the libraries, you can verify that everything is set up correctly by launching a Python interpreter or creating a Python script and importing the libraries: import tensorflow as tf import keras import torch print("TensorFlow version:", tf.__version__) print("Keras version:", keras.__version__) print("PyTorch version:", torch.__version__) This code will output the versions of the installed libraries, confirming that everything is installed correctly.
6. Using the libraries	Install libraries, use for machine learning models training. Here's a basic example of how you can create a simple

	neural network using TensorFlow/Keras and PyTorch:

- **Using TensorFlow/Keras:**

```
import tensorflow as tf
from tensorflow.keras import layers
# Create a simple neural network
model = tf.keras.Sequential([
    layers.Dense(64, activation='relu',
input_shape=(784,)),
    layers.Dense(10, activation='softmax')
])
# Compile the model
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
# Train the model (example data used here)
# model.fit(train_data, train_labels, epochs=10,
batch_size=32)
```

- **Using PyTorch:**

```
import torch
import torch.nn as nn
import torch.optim as optim
# Create a simple neural network
class SimpleNet(nn.Module):
    def __init__(self):
        super(SimpleNet, self).__init__()
        self.fc1 = nn.Linear(784, 64)
        self.fc2 = nn.Linear(64, 10)
    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)
        return x
model = SimpleNet()
# Define loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
# Train the model (example data used here)
# for epoch in range(10):
#     running_loss = 0.0
#     for data, labels in train_loader:
#         optimizer.zero_grad()
#         outputs = model(data)
#         loss = criterion(outputs, labels)
```

	# loss.backward() # optimizer.step() # running_loss += loss.item() # print(f"Epoch {epoch+1}, Loss: {running_loss/len(train_loader)}")

3. Applying the Convolution Neural Network on computer vision problems

AIM: Applying the Convolution Neural Network on computer vision problems

DESCRIPTION:

CNNs (Convolutional Neural Networks) revolutionize computer vision by learning spatial hierarchies automatically.

Here are some common computer vision problems where CNNs are frequently used:

Image Classification	Image classification involves assigning a label or category to an input image from a predefined set of classes. CNNs excel at this task due to their ability to learn hierarchical features and capture patterns in images.
Object Detection	Object detection involves identifying and localizing multiple objects of interest within an image. CNNs can be combined with techniques like Region Proposal Networks (RPNs) or Single Shot Multibox Detector (SSD) to accomplish object detection tasks.
Semantic Segmentation	Semantic segmentation involves classifying each pixel of an image into a specific category. CNNs with fully convolutional architectures, such as U-Net or DeepLab, are commonly used for semantic segmentation.
Instance Segmentation	Instance segmentation goes beyond semantic segmentation by not only classifying each pixel but also distinguishing individual object instances. CNNs, combined with methods like Mask R-CNN, are often used for instance segmentation tasks.
Image Generation	CNNs can be used for image generation tasks, such as generating realistic images from scratch. Generative Adversarial Networks (GANs) and Variational Autoencoders (VAEs) are popular architectures for image synthesis.
Style Transfer	Style transfer involves transferring the visual style of one image to another while preserving the content. CNNs are used to separate and manipulate the content and style representations of the images.
Image Captioning	Image captioning combines computer vision and natural language processing to generate textual descriptions for

	input images. CNNs are used to extract visual features, which are then combined with recurrent neural networks (RNNs) to generate captions.
Super-Resolution	Super-resolution aims to upscale low-resolution images to higher resolutions. CNNs, particularly architectures like SRGAN (Super-Resolution Generative Adversarial Network), are commonly used for super-resolution tasks.
Face Recognition	CNNs have demonstrated remarkable performance in face recognition tasks, where the goal is to identify individuals based on their facial features.
Image Denoising and Restoration	CNNs can be applied to remove noise, artifacts, or restore damaged images, making them useful in image denoising and restoration tasks.

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

AIM: Image classification on MNIST dataset (CNN model with Fully connected layer)

DESCRIPTION:

Image classification on the MNIST dataset is a classic example and a great starting point for understanding how to build a Convolutional Neural Network (CNN) with a Fully Connected Layer for image recognition. The MNIST dataset consists of 28x28 grayscale images of handwritten digits from 0 to 9. Each image is associated with a label representing the digit it represents.

Here's a step-by-step guide to building a CNN model with a Fully Connected Layer for image classification on the MNIST dataset using TensorFlow and Keras:

Step 1: Import the necessary libraries

```
import numpy as np

import tensorflow as tf

from tensorflow.keras import layers, models, datasets
```

Step 2: Load and preprocess the MNIST dataset

```
# Load MNIST dataset

(train_images, train_labels), (test_images, test_labels) =
datasets.mnist.load_data()

# Normalize pixel values to range [0, 1]

train_images, test_images = train_images / 255.0, test_images / 255.0

# Expand dimensions to add channel dimen

sion (for grayscale images)

train_images = np.expand_dims(train_images, axis=-1)

test_images = np.expand_dims(test_images, axis=-1)
```

Step 3: Build the CNN model

```
model = models.Sequential()
# Convolutional layers
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
# Flatten the 3D output to 1D
model.add(layers.Flatten())
# Fully Connected layers
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))  # 10 output classes (0-9 digits)
```

Step 4: Compile the model

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

Step 5: Train the model

```
model.fit(train_images, train_labels, epochs=5, batch_size=64, validation_split=0.1)
```

Step 6: Evaluate the model on the test set

```
test_loss, test_accuracy = model.evaluate(test_images, test_labels)
print("Test accuracy:", test_accuracy)
```

Step 7: Make predictions on new data

```
predictions = model.predict(test_images[:5])
predicted_labels = np.argmax(predictions, axis=1)
print("Predicted labels:", predicted_labels)
print("True labels:", test_labels[:5])
```

This code will create a CNN model with three convolutional layers followed by a fully connected layer. It will then train the model on the MNIST dataset and evaluate its accuracy on the test set. Finally, it will make predictions on five test images and display the predicted labels along with the true labels.

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

AIM: Applying the Deep Learning Models in the field of Natural Language Processing

DESCRIPTION:

Deep learning models have made significant contributions to the field of Natural Language Processing (NLP), enabling the development of powerful language models and applications. Some of the key deep learning models used in NLP include:

Recurrent Neural Networks (RNNs):
RNNs are designed to handle sequential data, making them well-suited for natural language processing tasks where the order of words matters. They process input data step-by-step while maintaining hidden states to capture context. However, traditional RNNs suffer from vanishing gradient problems. Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) are popular variants of RNNs that address this issue.

Applications:

Text classification	Sentiment analysis	Named Entity Recognition (NER)	Language modeling

Transformers:
Transformers introduced the attention mechanism, enabling more efficient and parallelized processing of sequential data. They have revolutionized NLP by capturing long-range dependencies effectively and have become the backbone of modern language models.

Applications:

Machine Translation (e.g., Google's Transformer-based model "BERT" for NMT)	Text generation (e.g., OpenAI's GPT series)	Question-Answering (e.g., Google's BERT-based model "BERT" for QA)

Bidirectional Encoder Representations from Transformers (BERT):
BERT is a transformer-based language model pre-trained on a large corpus of text data. It learns contextualized word representations, allowing it to understand the context in which a word appears in a sentence. BERT's pre-

trained representations can be fine-tuned for a wide range of NLP tasks, making it a versatile and powerful model.

Applications:

Text classification	Named Entity Recognition (NER)	Sentiment analysis	Question-Answering

Generative Pre-trained Transformer (GPT):
GPT is a family of transformer-based language models developed by OpenAI. GPT-3, in particular, is one of the largest language models ever created, with 175 billion parameters. It demonstrates impressive capabilities in natural language understanding and generation.

Applications:

Text completion	Text generation (e.g., creative writing, code generation)	Language translation

Convolutional Neural Networks for NLP:
Although more commonly used for computer vision, CNNs can be adapted to NLP tasks. They are often employed for text classification and sentiment analysis by treating text as a 1D sequence of tokens.

Applications:

Text classification	Sentiment analysis

Sequence-to-Sequence Models:
Sequence-to-sequence models use encoder-decoder architectures to handle tasks that involve transforming one sequence into another, such as machine translation and summarization.

Applications:

Machine translation	Text summarization

Attention Mechanisms:
Attention mechanisms are not models themselves, but they have been instrumental in improving the performance of various NLP models. They allow models to focus on specific parts of the input during processing, enhancing their understanding and performance.

Applications:

Language translation	Text generation

These deep learning models, along with their variants and combinations, have driven significant advancements in the field of Natural Language Processing, enabling the development of more sophisticated and context-aware language models and applications. As research progresses, we can expect further improvements and innovations in NLP, ultimately leading to more accurate and human-like language processing systems.

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

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

DESCRIPTION:

To train a sentiment analysis model on the IMDB dataset using RNN layers with LSTM/GRU units, we'll use TensorFlow and Keras. The IMDB dataset contains movie reviews labeled as positive or negative sentiments.

Step 1: Import the necessary libraries.

```
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, GRU, Dense
```

Step 2: Load and preprocess the IMDB dataset.

```
# Load the IMDB dataset, keeping only the top 10,000 most frequent words
num_words = 10000
(train_data, train_labels), (test_data, test_labels) =
imdb.load_data(num_words=num_words)
# Pad the sequences to ensure they all have the same length
max_length = 250
train_data = pad_sequences(train_data, maxlen=max_length)
test_data = pad_sequences(test_data, maxlen=max_length)
```

Step 3: Build the RNN model with LSTM or GRU layers.

```
embedding_dim = 100
model = Sequential()
model.add(Embedding(input_dim=num_words, output_dim=embedding_dim,
input_length=max_length))
# Choose LSTM or GRU layer
# LSTM Layer
# model.add(LSTM(units=64, dropout=0.2, recurrent_dropout=0.2))
# GRU Layer
model.add(GRU(units=64, dropout=0.2, recurrent_dropout=0.2))
model.add(Dense(units=1, activation='sigmoid'))
```

Step 4: Compile and train the model.

```
model.compile(optimizer='adam', loss='binary_crossentropy',
metrics=['accuracy'])
batch_size = 128
```

```
epochs = 5
model.fit(train_data, train_labels, batch_size=batch_size, epochs=epochs,
validation_split=0.2)
```

Step 5: Evaluate the model on the test set.

```
test_loss, test_accuracy = model.evaluate(test_data, test_labels)
print("Test accuracy:", test_accuracy)
```

That's it! With these steps, you have built and trained a sentiment analysis model on the IMDB dataset using RNN layers with LSTM/GRU units. The model will learn to predict whether movie reviews are positive or negative based on the given text. You can experiment with different hyperparameters, such as the number of LSTM/GRU units, the embedding dimension, or the number of epochs, to see how they affect the model's performance.

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

AIM: Applying the Autoencoder algorithms for encoding the real-world data

DESCRIPTION:

Autoencoders are a class of neural network architectures used for unsupervised learning. They are particularly useful for encoding real-world data into a lower-dimensional representation, also known as the "latent space" or "encoding space." Autoencoders consist of two main parts: the encoder, which maps the input data to the latent space, and the decoder, which reconstructs the data from the encoded representation. The objective is to minimize the difference between the original input and the reconstructed output, encouraging the model to learn a compact representation of the data.

Applying Autoencoders for encoding real-world data involves the following steps:

Step 1: Data Preparation

Collect and preprocess the real-world data you want to encode. Ensure that the data is in a suitable format and normalize it if necessary.

Step 2: Build the Autoencoder Model

Create the Autoencoder model using a neural network library like TensorFlow or Keras.

```
import tensorflow as tf
from tensorflow.keras.layers import Input, Dense
from tensorflow.keras.models import Model
# Define the input size
input_size = <your_input_size>
# Define the size of the latent space (encoded representation)
latent_dim = <your_desired_latent_dimension>
# Encoder
input_data = Input(shape=(input_size,))
encoded = Dense(latent_dim, activation='relu')(input_data)
# Decoder
decoded = Dense(input_size, activation='sigmoid')(encoded)
# Autoencoder model
autoencoder = Model(input_data, decoded)
```

Step 3: Compile and Train the Autoencoder

Compile the model with an appropriate optimizer and loss function, and then train the Autoencoder on your real-world data.

```
autoencoder.compile(optimizer='adam', loss='mean_squared_error')
# Assuming you have your data loaded into 'data'
```

autoencoder.fit(data, data, epochs=50, batch_size=32)

Step 4: Encode Real-World Data
Once the Autoencoder is trained, you can use the encoder part of the model to encode your real-world data into the latent space.
encoder = Model(input_data, encoded)
encoded_data = encoder.predict(data)
The encoded_data variable now contains the encoded representation of your real-world data in the lower-dimensional latent space.

Step 5: Decode Encoded Data (Optional)
If desired, you can use the decoder part of the model to reconstruct the data from the encoded representation.
# Assuming you have the encoder part of the model stored in 'encoder'
decoder_input = Input(shape=(latent_dim,))
decoded_output = autoencoder.layers[-1](decoder_input) # Get the last layer of the decoder
decoder = Model(decoder_input, decoded_output)
# Reconstruct data from encoded representation
reconstructed_data = decoder.predict(encoded_data)

Autoencoders are powerful tools for dimensionality reduction and feature learning in real-world data. They are widely used in various applications, including anomaly detection, data compression, and denoising. By training an Autoencoder, you can obtain a more compact representation of your data, which can be useful for downstream tasks or visualizations.

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

AIM: Applying Generative Adversial Networks for image generation and unsupervised tasks.

DESCRIPTION:

Generative Adversarial Networks (GANs) are a powerful class of deep learning models used for various tasks, including image generation and unsupervised learning. GANs consist of two neural networks: a generator and a discriminator. The generator generates fake data (e.g., images), while the discriminator tries to distinguish between real data and fake data generated by the generator. The two networks are trained simultaneously in a competition, where the generator tries to produce data that fools the discriminator, and the discriminator tries to get better at distinguishing real from fake data.

Here's how GANs can be applied for image generation and unsupervised tasks:

Image Generation with GANs:

The primary application of GANs is image generation. The generator learns to create new images that resemble the training data. The discriminator provides feedback to the generator, guiding it to produce more realistic images over time.

Step 1: Import the necessary libraries.

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.layers import Dense, LeakyReLU, BatchNormalization,
Reshape, Conv2DTranspose, Conv2D, Flatten
from tensorflow.keras.models import Sequential
```

Step 2: Build the Generator and Discriminator models.

```
def build_generator(latent_dim):
    model = Sequential()
    model.add(Dense(256, input_dim=latent_dim))
    model.add(LeakyReLU(alpha=0.01))
    model.add(BatchNormalization(momentum=0.8))
    model.add(Dense(512))
    model.add(LeakyReLU(alpha=0.01))
    model.add(BatchNormalization(momentum=0.8))
    model.add(Dense(1024))
    model.add(LeakyReLU(alpha=0.01))
    model.add(BatchNormalization(momentum=0.8))
    model.add(Dense(28 * 28 * 1, activation='tanh'))
    model.add(Reshape((28, 28, 1)))
```

```
    return model
def build_discriminator(img_shape):
    model = Sequential()
    model.add(Flatten(input_shape=img_shape))
    model.add(Dense(512))
    model.add(LeakyReLU(alpha=0.01))
    model.add(Dense(256))
    model.add(LeakyReLU(alpha=0.01))
    model.add(Dense(1, activation='sigmoid'))
    return model
```

Step 3: Define the GAN model by combining the Generator and Discriminator.

```
def build_gan(generator, discriminator):
    discriminator.trainable = False
    model = Sequential()
    model.add(generator)
    model.add(discriminator)
    return model
```

Step 4: Train the GAN model.

```
# Define hyperparameters
latent_dim = 100
img_shape = (28, 28, 1)
# Build and compile the discriminator and generator
generator = build_generator(latent_dim)
discriminator = build_discriminator(img_shape)
gan = build_gan(generator, discriminator)
discriminator.compile(loss='binary_crossentropy',
optimizer=tf.keras.optimizers.Adam(0.0002, 0.5))
gan.compile(loss='binary_crossentropy',
optimizer=tf.keras.optimizers.Adam(0.0002, 0.5))
# Load and preprocess real image data (e.g., from MNIST dataset)
# Training loop
epochs = 10000
batch_size = 64
half_batch = batch_size // 2
for epoch in range(epochs):
    # Train the discriminator
    idx = np.random.randint(0, X_train.shape[0], half_batch)
```

```
    real_images = X_train[idx]
    noise = np.random.normal(0, 1, (half_batch, latent_dim))
    generated_images = generator.predict(noise)
    d_loss_real = discriminator.train_on_batch(real_images, np.ones((half_batch,
1)))
    d_loss_fake = discriminator.train_on_batch(generated_images,
np.zeros((half_batch, 1)))
    d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)
    # Train the generator
    noise = np.random.normal(0, 1, (batch_size, latent_dim))
    valid_labels = np.ones((batch_size, 1))
    g_loss = gan.train_on_batch(noise, valid_labels)
```

Unsupervised Tasks with GANs:

GANs can be used for various unsupervised learning tasks, such as data augmentation, feature extraction, and anomaly detection. For example, a pre-trained GAN generator can be used to generate additional data for training purposes, which can improve the performance of other models. GANs can also be used for feature extraction by using the intermediate layers of the generator as feature representations for downstream tasks.

In summary, GANs are versatile models that have proven to be effective in image generation and unsupervised learning tasks. They have a wide range of applications and continue to be an active area of research in the deep learning community.