NAME – Mansi
Phone no - 9311453903
Email id – mansidobriyal50@gmail.com

Assignments: Deep Learning Frameworks

1. What is TensorFlow 2.0, and how is it different from TensorFlow 1.x2

   Ans - TensorFlow 2.0 is a major upgrade to Google's open-source machine learning framework, designed for simplicity and ease of use. Key differences from TensorFlow 1.x include:

   1. **Eager Execution**: Enabled by default for immediate operation evaluation, making debugging and prototyping easier.
   2. **Keras Integration**: Includes tf.keras as its high-level API for building and training models.
   3. **Simplified API**: Streamlined operations by removing redundant APIs.
   4. **Enhanced Model Building**: Improved usability with better support for custom and pre-built models.

2. How do you install TensorFlow 2.02

   Ans - To install TensorFlow 2.0:

   1. **Set Up Python Environment**: Use Python 3.7+ and create a virtual environment (optional).
   2. **Install**: Run pip install tensorflow.
   3. **Verify**: Check the version with:

```
import tensorflow as tf
print(tf.__version__)

[1]  ✓ 45.1s

...  2.18.0
```

   4. **GPU Support (Optional)**: Use pip install tensorflow-gpu and ensure CUDA/cuDNN compatibility.

```
pip install tensorflow-gpu

2]  ✓ 10.7s

Collecting tensorflow-gpu
  Downloading tensorflow-gpu-2.12.0.tar.gz (2.6 kB)
  Preparing metadata (setup.py): started
  Preparing metadata (setup.py): finished with status 'error'
```

3. What is the primary function of the tf.function in TensorFlow 2.02

   Ans - The primary function of tf.function is to convert Python functions into TensorFlow computational graphs for faster execution and portability. Example:

```python
import tensorflow as tf

@tf.function
def add(a, b):
    return a + b

x = tf.constant(2)
y = tf.constant(3)
print(add(x, y))  # Outputs: tf.Tensor(5, shape=(), dtype=int32)
```

[4] ✓ 0.7s

```
tf.Tensor(5, shape=(), dtype=int32)
```

4. What is the purpose of the Model class in TensorFlow 2.02

   Ans - The Model class in TensorFlow 2.0 (part of tf.keras) is used to create and manage neural network models. It provides methods for building, training, evaluating, and saving models.

   Key Features:

- **Model Building**: Define models using sequential or functional APIs.
- **Training**: Use model.fit() for training.
- **Evaluation**: Use model.evaluate() and model.predict() for testing and inference.
- **Save/Load**: Save models with model.save() and load with tf.keras.models.load_model().

5. How do you create a neural network using TensorFlow 2.02

   Ans - To create a neural network in TensorFlow 2.0:

1. **Import Libraries**:

```python
import tensorflow as tf
from tensorflow.keras import layers, models
```

2. **Define the Model**:

```python
model = models.Sequential([
    layers.Dense(64, activation='relu', input_shape=(input_size,)),
    layers.Dense(32, activation='relu'),
    layers.Dense(output_size, activation='softmax')
])
```

[8] ✓ 0.2s

```
d:\anaconda\Lib\site-packages\keras\src\layers\core\dense.py:87: UserWarning: Do not pa
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

3. **Compile the Model**:

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

4. **Train the Model**:

```
model.fit(x_train, y_train, epochs=10, batch_size=32)
```

6. What is the importance of Tensor Space in TensorFlow2

Ans – In TensorFlow, **Tensor Space** refers to the mathematical representation of data as multidimensional arrays (tensors). It's critical because:

1. **Unified Data Representation**: Tensors store and process data in a consistent format, enabling TensorFlow to handle complex computations efficiently.
2. **Flexibility**: Tensors can represent scalars (0D), vectors (1D), matrices (2D), and higher-dimensional data (ND), supporting various machine learning tasks.
3. **Hardware Optimization**: Tensors enable TensorFlow to optimize operations for CPUs, GPUs, and TPUs seamlessly.
4. **Efficient Computation**: Operations on tensors are vectorized, allowing parallel processing for faster execution.
5. **Core to TensorFlow**: All TensorFlow computations, from model inputs to outputs, rely on tensors as the foundational data structure.

7. How can TensorBoard be integrated with TensorFlow 2.02

Ans –

To integrate TensorBoard with TensorFlow 2.0:

1. **Import Libraries**:

```
import tensorflow as tf
from tensorflow.keras.callbacks import TensorBoard
```

2. **Set Up TensorBoard Callback**: Create a directory to store logs and set up the TensorBoard callback:

```
log_dir = "logs/fit"  # Log directory for TensorBoard
tensorboard_callback = TensorBoard(log_dir=log_dir, histogram_freq=1)
```

3. **Train the Model with TensorBoard**: Pass the tensorboard_callback during model training

```
model.fit(x_train, y_train, epochs=10, batch_size=32, callbacks=[tensorboard_callback])
```

4. **Launch TensorBoard**: After training, open TensorBoard in the terminal:

```
tensorboard --logdir=logs/fit
```

8. What is the purpose of TensorFlow Playground2

The purpose of **TensorFlow Playground** is to provide an interactive, web-based environment for experimenting with neural networks in real-time. It allows users to:

1. **Visualize Neural Networks**: See how different hyperparameters, architectures, and data affect the training process.
2. **Experiment with Model Parameters**: Adjust settings like learning rate, activation functions, and the number of layers.
3. **Understand Concepts**: Helps beginners understand the working of neural networks without needing to write code.
4. **Explore Data**: Use toy datasets for classification and regression tasks.

9. What is Netron, and how is it useful for deep learning models2

Ans - Netron is a web-based tool for visualizing and analyzing deep learning models. It supports various frameworks, including TensorFlow, Keras, PyTorch, ONNX, and more.

Key Uses:

1. **Model Visualization**: Displays the architecture of models, showing layers, their connections, and parameters.
2. **Model Debugging**: Helps in understanding model structures and troubleshooting issues by providing a clear, interactive view.
3. **Cross-framework Support**: Can visualize models from different frameworks, making it versatile for various deep learning workflows.
4. **Model Summary**: Offers insights into the model's layers, shapes, and parameters for easier analysis and optimization.

10.What is the difference between TensorFlow and PyTorch2

Ans - The key differences between **TensorFlow** and **PyTorch** are:

1. **Execution Model**:
    - **TensorFlow**: Uses a static computation graph (eager execution is available in TensorFlow 2.0).
    - **PyTorch**: Uses dynamic computation graphs (eager execution by default).
2. **API and Usability**:
    - **TensorFlow**: Initially more complex but improved in version 2.0 with Keras integration for easier use.
    - **PyTorch**: Known for its simple and intuitive API, making it more popular in research.
3. **Deployment**:
    - **TensorFlow**: Offers TensorFlow Serving and TensorFlow Lite for deployment in production and on mobile devices.
    - **PyTorch**: PyTorch's TorchServe and mobile deployment tools are evolving but not as mature as TensorFlow's.
4. **Community and Ecosystem**:
    - **TensorFlow**: Larger production-focused ecosystem, extensive documentation, and enterprise support.
    - **PyTorch**: Popular in research, with a growing ecosystem for deployment but more research-oriented.
5. **Performance**:
    - Both provide strong performance, with TensorFlow often being slightly better for production-scale applications and PyTorch excelling in flexibility for research.

11. How do you install PyTorch2

Ans –

To install PyTorch 2, follow these steps:

Step 1: Visit the Installation Page

Go to the PyTorch installation page to select the command based on your system and preferences.

Step 2: Install via Conda or Pip

Use the appropriate command. For example:

- **Using Conda:**

```
conda install pytorch torchvision torchaudio pytorch-cuda=11.8 -c pytorch -c nvidia
```

- **Using Pip:**

```
pip install torch torchvision torchaudio --index-url
https://download.pytorch.org/whl/cu118
```

Step 3: Verify Installation

After installation, verify it in Python:

```
import torch
print(torch.__version__)
print(torch.cuda.is_available())
```

12. What is the basic structure of a PyTorch neural network2

The basic structure of a PyTorch neural network involves defining a class that inherits from torch.nn.Module. Here's a simple structure:

1. **Import Libraries**

```
import torch
import torch.nn as nn
import torch.nn.functional as F
```

2. Define the Model

```
class NeuralNetwork(nn.Module):
    def __init__(self):
        super(NeuralNetwork, self).__init__()
        self.fc1 = nn.Linear(in_features=10, out_features=20)  # Input layer
        self.fc2 = nn.Linear(20, 5)  # Output layer

    def forward(self, x):
        x = F.relu(self.fc1(x))  # Activation function
        x = self.fc2(x)         # Output layer
        return x
```

3. Initialize and Use

```
model = NeuralNetwork()
input_data = torch.randn(1, 10)
output = model(input_data)
print(output)
```

This structure includes:

- **Initialization (`__init__`)**: Layers and parameters.
- **Forward method**: Defines data flow through the layers.

13. What is the significance of tensors in PyTorch2
    Ans –
    Tensors are the core data structure in PyTorch, similar to multidimensional arrays in NumPy but with additional capabilities. Their significance lies in:
    1. **Data Representation**:
        - Represent data as scalars, vectors, matrices, or higher-dimensional arrays.
        - Example: Input data, weights, and outputs in neural networks.
    2. **GPU Acceleration**:
        - Efficiently perform computations on GPUs using CUDA.
        - Example:

```
tensor = torch.randn(3, 3, device='cuda')
```

Autograd Support:

- Enable automatic differentiation for training models.
- Example:

```
x = torch.randn(3, requires_grad=True)
y = x ** 2
y.backward(torch.ones_like(x))  # Compute gradients
print(x.grad)
```

Interoperability:

- Convert easily between PyTorch tensors and NumPy arrays.

```
numpy_array = tensor.numpy()
tensor_from_numpy = torch.tensor(numpy_array)
```

1. **Flexibility**:
    - Suitable for dynamic computations, batch operations, and broadcasting.

Tensors are foundational to PyTorch's computational framework, enabling its flexibility and power for deep learning tasks.

14. What is the difference between torch.Tensor and torch.cuda.Tensor in PyTorch2
    Ans –
    The key difference between torch.Tensor and torch.cuda.Tensor lies in **device placement** and **capabilities**:
    1. Device Placement:
    - **torch.Tensor**:
      Created on the **CPU** by default.
      Example:

```
cpu_tensor = torch.Tensor([1, 2, 3])
print(cpu_tensor.device)  # Output: cpu
```

- **torch.cuda.Tensor**:
  Specifically created on the **GPU** for accelerated computations.
  Example:

```
gpu_tensor = torch.cuda.FloatTensor([1, 2, 3])
print(gpu_tensor.device)  # Output: cuda:0
```

2. Performance:

- **torch.Tensor**: Slower for large computations as it uses the CPU.
- **torch.cuda.Tensor**: Leverages GPU, providing faster operations for large-scale computations.

3. Conversion:

You can move tensors between devices:

```
cpu_tensor = torch.Tensor([1, 2, 3])
gpu_tensor = cpu_tensor.to('cuda')  # Moves to GPU
```

4. Usage:

- Use **torch.Tensor** for small-scale tasks or non-GPU environments.
- Use **torch.cuda.Tensor** for GPU-enabled training or inference to maximize performance.

Since PyTorch 1.x, you rarely need torch.cuda.Tensor explicitly; use .to('cuda') to specify device placement.

15. What is the purpose of the torch.optim module in PyTorch2.

The torch.optim module in PyTorch provides **optimization algorithms** for training machine learning models. Its main purposes are:

1. **Update Model Parameters**:
   Adjust model weights based on gradients computed during backpropagation to minimize the loss function.
2. **Provide Optimization Algorithms**:
   Implements popular optimization methods like:
   - **SGD (Stochastic Gradient Descent)**
   - **Adam**
   - **RMSprop**
   - **Adagrad**, etc.
3. **Ease of Use**:
   Simplifies optimization by handling parameter updates automatically.

Example:

```
import torch
import torch.nn as nn
import torch.optim as optim

model = nn.Linear(2, 1)  # Simple model
optimizer = optim.Adam(model.parameters(), lr=0.01)  # Adam optimizer

# Training loop
for input, target in data_loader:
```

```
    optimizer.zero_grad()  # Clear previous gradients
    output = model(input)
    loss = nn.MSELoss()(output, target)
    loss.backward()        # Compute gradients
    optimizer.step()       # Update parameters
```

16. What are some common activation functions used in neural networks2

Ans –

Common Activation Functions:

1. **ReLU**: max⁡(0,x)\max(0, x)max(0,x), avoids vanishing gradients, for hidden layers.

```
python
Copy code
torch.nn.ReLU()
```

2. **Sigmoid**: 11+e−x\frac{1}{1 + e^{-x}}1+e−x1, outputs 0-1, for binary classification.

```
python
Copy code
torch.nn.Sigmoid()
```

3. **Tanh**: tanh⁡(x)\tanh(x)tanh(x), outputs -1 to 1, centered at zero.

```
python
Copy code
torch.nn.Tanh()
```

4. **Softmax**: Converts to probabilities, for multi-class classification.

```
python
Copy code
torch.nn.Softmax(dim=1)
```

5. **Leaky ReLU**: Allows small gradients for x<0x < 0x<0, avoids dying neurons.

```
python
Copy code
torch.nn.LeakyReLU()
```

17. What is the difference between torch.nn.Module and torch.nn.Sequential in PyTorch2

key Differences:

Aspect	`torch.nn.Module`	`torch.nn.Sequential`
Purpose	General base class for creating any model.	Simplifies linear stacking of layers.
Flexibility	Highly customizable with any architecture.	Limited to sequential layers.
Custom Logic	Allows complex forward methods.	No custom logic; layers run in sequence.
Use Case	For complex, non-linear models.	For simple feedforward architectures.

Examples:

- **Using torch.nn.Module**:

```
class MyModel(nn.Module):
    def __init__(self):
        super(MyModel, self).__init__()
        self.fc1 = nn.Linear(10, 20)
        self.fc2 = nn.Linear(20, 1)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        return self.fc2(x)
```

- **Using torch.nn.Sequential**:

```
model = nn.Sequential(
    nn.Linear(10, 20),
    nn.ReLU(),
    nn.Linear(20, 1)
)
```

Summary: Use nn.Module for complex models with custom logic and nn.Sequential for simpler, stacked architectures.

18. How can you monitor training progress in TensorFlow 2.02

Ans –

In TensorFlow 2.0+ (including 2.02), you can monitor training progress using the following methods:

1. TensorBoard:

- **Purpose**: Visualize training metrics like loss and accuracy.
- **Usage**:
- `from tensorflow.keras.callbacks import TensorBoard`
- `tensorboard_callback = TensorBoard(log_dir='./logs')`
- `model.fit(x_train, y_train, epochs=10, callbacks=[tensorboard_callback])`
- **Launch TensorBoard**:
- `tensorboard --logdir=./logs`

2. Verbose in `fit()`:

- **Purpose**: Show training progress with a progress bar.
- **Usage**:
- `model.fit(x_train, y_train, epochs=10, verbose=1)  # 1: progress bar, 2: per epoch details`

3. Custom Callbacks:

- **Purpose**: Add custom logic to monitor metrics.
- **Usage**:
- `class CustomCallback(tf.keras.callbacks.Callback):`
- `    def on_epoch_end(self, epoch, logs=None):`
- `        print(f"Epoch {epoch + 1}: Loss = {logs['loss']}")`
- `model.fit(x_train, y_train, epochs=10, callbacks=[CustomCallback()])`

4. Plot Metrics:

- **Purpose**: Plot metrics like loss and accuracy after training.
- **Usage**:
- `history = model.fit(x_train, y_train, epochs=10)`
- `plt.plot(history.history['loss'], label='Loss')`
- `plt.plot(history.history['accuracy'], label='Accuracy')`
- `plt.legend()`
- `plt.show()`

These methods allow you to efficiently monitor and visualize the training progress of your model.

19. How does the Keras API fit into TensorFlow 2.02

In TensorFlow 2.0+ (including 2.02), **Keras** is the default high-level API for building and training models.

Key Points:

1. **Unified API**: Keras is integrated into TensorFlow, simplifying model building, training, and evaluation.
2. **Model Building**: Use the **Sequential** or **Functional API** to define models.
3. model = tf.keras.Sequential([tf.keras.layers.Dense(64, activation='relu', input_shape=(32,))])
4. **Optimizers, Loss, and Metrics**: Easily integrate TensorFlow features with Keras (tf.keras.optimizers, tf.keras.losses).
5. **Training**: Use fit() for training and evaluation.
6. model.compile(optimizer='adam', loss='categorical_crossentropy')
7. model.fit(x_train, y_train, epochs=10)

Keras in TensorFlow 2.0+ provides a simpler interface while leveraging TensorFlow's powerful backend.

20. What is an example of a deep learning project that can be implemented using TensorFlow 2.02

Ans –

An example of a deep learning project that can be implemented using TensorFlow 2.0+ is an **Image Classification** project using a **Convolutional Neural Network (CNN)**.

Project Overview:

- **Objective**: Classify images into categories (e.g., identifying cats vs. dogs).
- **Dataset**: You can use the **Cats vs Dogs** dataset from Kaggle or any other labeled image dataset.

Steps:

1. **Import Libraries**:

```
import tensorflow as tf
from tensorflow.keras import layers, models
```

2. **Load and Preprocess Data**:

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator

# Preprocess images
train_datagen = ImageDataGenerator(rescale=1./255, validation_split=0.2)
test_datagen = ImageDataGenerator(rescale=1./255)

train_data = train_datagen.flow_from_directory('data/train', target_size=(150, 150),
batch_size=32, class_mode='binary', subset='training')
```

```
val_data = train_datagen.flow_from_directory('data/train', target_size=(150, 150),
batch_size=32, class_mode='binary', subset='validation')
```

3. **Build the Model (CNN)**:

```
model = models.Sequential([
    layers.Conv2D(32, (3, 3), activation='relu', input_shape=(150, 150, 3)),
    layers.MaxPooling2D(2, 2),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPooling2D(2, 2),
    layers.Flatten(),
    layers.Dense(128, activation='relu'),
    layers.Dense(1, activation='sigmoid')  # Binary classification
])
```

4. **Compile the Model**:

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

5. **Train the Model**:

```
model.fit(train_data, epochs=10, validation_data=val_data)
```

6. **Evaluate the Model**:

```
test_data = test_datagen.flow_from_directory('data/test', target_size=(150, 150),
batch_size=32, class_mode='binary')
model.evaluate(test_data)
```

Outcome:

- A trained CNN model that can classify images of cats and dogs based on the training data.

This is a straightforward deep learning project that demonstrates image classification using TensorFlow 2.0+ and CNNs.

21. What is the main advantage of using pre-trained models in TensorFlow and PyTorch?

Ans –

The main advantage of using **pre-trained models** in TensorFlow and PyTorch is **faster training and improved performance**. Here's why:

1. Reduced Training Time:

- Pre-trained models are already trained on large datasets (e.g., ImageNet). This means you don't have to train a model from scratch, saving time and computational resources.

2. Better Accuracy:

- Since pre-trained models have learned features from large and diverse datasets, they often perform better on similar tasks, even with smaller datasets.

3. Transfer Learning:

- You can fine-tune a pre-trained model on your specific task. This allows the model to adapt to your data with fewer epochs and less data.

4. Efficient Use of Resources:

- Using pre-trained models allows you to leverage the knowledge embedded in models trained by experts on large datasets, making it more efficient compared to training from scratch.

Example:

In TensorFlow:

```
from tensorflow.keras.applications import VGG16

# Load a pre-trained VGG16 model
model = VGG16(weights='imagenet', include_top=False)
```

In PyTorch:

```
import torch
from torchvision import models

# Load a pre-trained ResNet model
model = models.resnet50(pretrained=True)
```

Summary:

Pre-trained models provide a strong starting point, reduce the need for large datasets, save time, and often achieve better results.

Practical

1. How do you install and verify that TensorFlow 2.0 was installed successfully2

Ans –

To install TensorFlow 2.0+:

1. **Install TensorFlow**: Run the following command in your terminal or command prompt:

```
pip install tensorflow
[4] ✓ 17.0s                                                     Python
Requirement already satisfied: tensorflow in d:\anaconda\lib\site-packages (2.18.0)Note: you may need to restart the kernel to use updated packages.

Requirement already satisfied: tensorflow-intel==2.18.0 in d:\anaconda\lib\site-packages (from tensorflow) (2.18.0)
Requirement already satisfied: absl-py>=1.0.0 in d:\anaconda\lib\site-packages (from tensorflow-intel==2.18.0->tensorflow) (2.1.0)
Requirement already satisfied: astunparse>=1.6.0 in d:\anaconda\lib\site-packages (from tensorflow-intel==2.18.0->tensorflow) (1.6.3)
Requirement already satisfied: flatbuffers>=24.3.25 in d:\anaconda\lib\site-packages (from tensorflow-intel==2.18.0->tensorflow) (24.3.25)
```

2. **Verify Installation**: After installation, you can verify that TensorFlow 2.0+ is installed successfully by running this Python code:

```
import tensorflow as tf
print(tf.__version__)  # Should print the installed TensorFlow version
[5] ✓ 26.3s
... 2.18.0
```

2. How can you define a simple function in TensorFlow 2.0 to perform addition2

Ans –

In TensorFlow 2.0, you can define a simple function to perform addition using the tf.function decorator or just TensorFlow operations.

Example: Simple addition function

```python
import tensorflow as tf

# Define a simple addition function
def add(x, y):
    return x + y

# Create TensorFlow constants
x = tf.constant(5)
y = tf.constant(3)

# Call the function
result = add(x, y)

print(result)  # Output: tf.Tensor(8, shape=(), dtype=int32)
```

[6] ✓ 0.0s

```
tf.Tensor(8, shape=(), dtype=int32)
```

Explanation:

- tf.constant is used to create TensorFlow tensors for the inputs.
- The function add(x, y) simply adds the two tensors.
- The result is a TensorFlow tensor, shown as tf.Tensor(8, shape=(), dtype=int32).

If you want to optimize the function (e.g., for faster execution), you can use @tf.function:

```python
@tf.function
def add(x, y):
    return x + y
```

[7] ✓ 0.0s

3. How can you create a simple neural network in TensorFlow 2.0 with one hidden layer2
To create a simple neural network in TensorFlow 2.0 with one hidden layer, you can use the tf.keras.Sequential API. Here's a basic example for a neural network with one hidden layer:

Example: Simple Neural Network

```python
import tensorflow as tf
from tensorflow.keras import layers, models

# Define the model
model = models.Sequential([
    layers.Dense(64, activation='relu', input_shape=(32,)),  # Input layer with 32 features
    layers.Dense(1, activation='sigmoid')  # Output layer for binary classification
])
# Compile the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
# Summary of the model
model.summary()
```

[8] ✓ 0.5s

```
d:\anaconda\Lib\site-packages\keras\src\layers\core\dense.py:87: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)

Model: "sequential"
```

Output –

```
… d:\anaconda\Lib\site-packages\keras\src\layers\core\dense.py:87: UserWarning: Do not pass an `input_
    super().__init__(activity_regularizer=activity_regularizer, **kwargs)

… Model: "sequential"

…
```

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 64)	2,112
dense_1 (Dense)	(None, 1)	65

```
…  Total params: 2,177 (8.50 KB)

…  Trainable params: 2,177 (8.50 KB)

…  Non-trainable params: 0 (0.00 B)
```

Explanation:

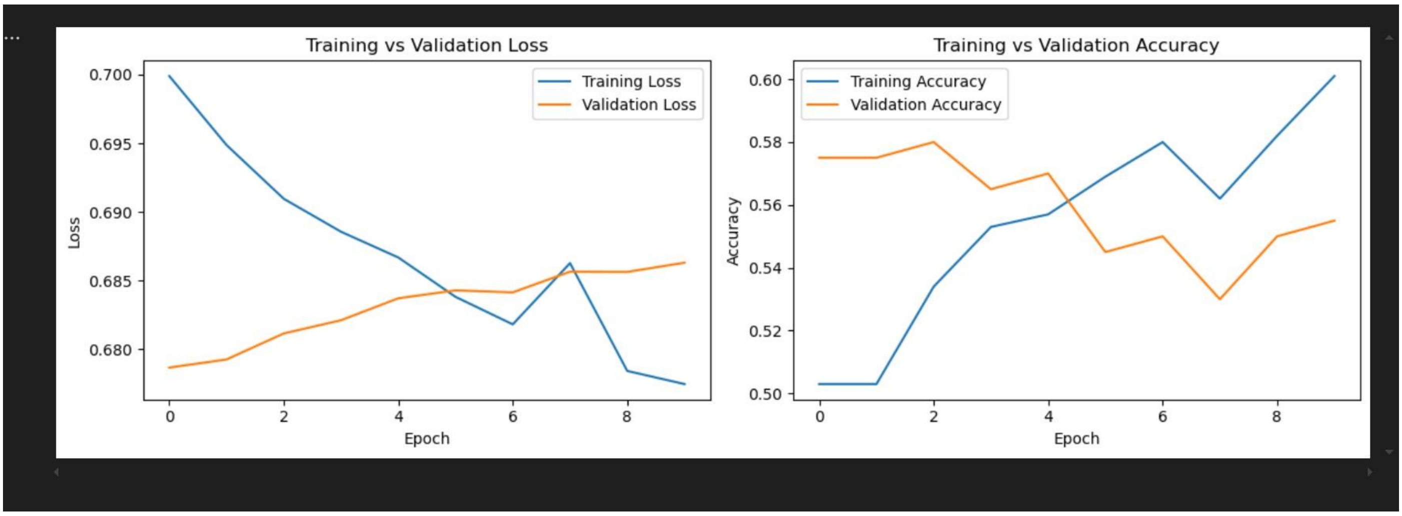
- **input_shape=(32,)**: Specifies the input layer with 32 features.
- **Dense(64, activation='relu')**: A hidden layer with 64 units and ReLU activation.
- **Dense(1, activation='sigmoid')**: Output layer for binary classification (use sigmoid activation for output between 0 and 1).
- **model.compile()**: Specifies the optimizer (adam), loss function (binary_crossentropy for binary classification), and evaluation metric (accuracy).

4. How can you visualize the training progress using TensorFlow and Matplotlib2
   To visualize the training progress in TensorFlow using **Matplotlib**, you can track the training and validation loss/accuracy during training and then plot the results. Here's how you can do it:
   Steps to visualize the training progress:
   1. **Train the Model and Save History**: During training, TensorFlow's fit() method returns a history object that stores training and validation metrics for each epoch.
   2. **Plot Metrics Using Matplotlib**: After training, you can plot the loss and accuracy for both training and validation data.

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

# Generate dummy data (replace with actual data)
x_train = np.random.rand(1000, 32)  # 1000 samples, 32 features
y_train = np.random.randint(0, 2, 1000)  # 1000 binary labels (0 or 1)
x_val = np.random.rand(200, 32)  # 200 samples, 32 features for validation
y_val = np.random.randint(0, 2, 200)  # 200 binary labels for validation
```

```python
# Define the model
model = tf.keras.Sequential([
    tf.keras.layers.Dense(64, activation='relu', input_shape=(32,)),
    tf.keras.layers.Dense(1, activation='sigmoid')  # Binary output
])

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

# Train the model and get history
history = model.fit(x_train, y_train, epochs=10, validation_data=(x_val, y_val))

# Plot training & validation loss
plt.figure(figsize=(12, 4))

# Loss plot
plt.subplot(1, 2, 1)
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Training vs Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()

# Accuracy plot
plt.subplot(1, 2, 2)
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Training vs Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.tight_layout()
plt.show()
```

output –



Explanation:

1. **Training**: The model.fit() function trains the model and stores metrics in history.history.
2. **Loss Plot**: We plot both training and validation loss against epochs.
3. **Accuracy Plot**: Similarly, we plot both training and validation accuracy.
4. **Matplotlib**: The plots are displayed using matplotlib.pyplot.

Output:

This will display two plots:

- One for **Loss** (training and validation).
- One for **Accuracy** (training and validation).

This helps you visualize how well the model is learning over time and whether it is overfitting or underfitting.

Explanation:

1. **Training**: The model.fit() function trains the model and stores metrics in history.history.
2. **Loss Plot**: We plot both training and validation loss against epochs.
3. **Accuracy Plot**: Similarly, we plot both training and validation accuracy.
4. **Matplotlib**: The plots are displayed using matplotlib.pyplot.

Output:

This will display two plots:

- One for **Loss** (training and validation).
- One for **Accuracy** (training and validation).

This helps you visualize how well the model is learning over time and whether it is overfitting or underfitting.

5. How do you install PyTorch and verify the PyTorch installation2

To install PyTorch and verify the installation, follow these steps:

Step 1: Install PyTorch

Use the following command to install the latest version of PyTorch via `pip`:

```
pip install torch torchvision torchaudio
```

[27] ✓ 41m 2.7s

```
Collecting torch
  Downloading torch-2.5.1-cp312-cp312-win_amd64.whl.metadata (28 kB)
Collecting torchvision
  Downloading torchvision-0.20.1-cp312-cp312-win_amd64.whl.metadata (6.2 kB)
Collecting torchaudio
  Downloading torchaudio-2.5.1-cp312-cp312-win_amd64.whl.metadata (6.5 kB)
Requirement already satisfied: filelock in d:\anaconda\lib\site-packages (from torch) (3.13.1)
Requirement already satisfied: typing-extensions>=4.8.0 in d:\anaconda\lib\site-packages (from torch) (4.11.
Requirement already satisfied: networkx in d:\anaconda\lib\site-packages (from torch) (3.2.1)
Requirement already satisfied: jinja2 in d:\anaconda\lib\site-packages (from torch) (3.1.4)
Requirement already satisfied: fsspec in d:\anaconda\lib\site-packages (from torch) (2024.3.1)
Requirement already satisfied: setuptools in d:\anaconda\lib\site-packages (from torch) (69.5.1)
Collecting sympy==1.13.1 (from torch)
  Downloading sympy-1.13.1-py3-none-any.whl.metadata (12 kB)
```

Step 2: Verify the Installation

To verify that PyTorch is installed successfully, run the following code in Python:

```python
import torch
print(torch.__version__)  # Prints the installed PyTorch version

# Check if CUDA is available (if you installed the CUDA version)
print(torch.cuda.is_available())  # Should return True if CUDA is available
```

```
28]  ✓ 16.1s

2.5.1+cpu
False
```

6. How do you create a simple neural network in PyTorch2

Steps to Create a Neural Network:

1. Define the network structure by subclassing torch.nn.Module.
2. Specify the layers in the __init__ method.
3. Define the forward pass in the forward method.

Example Code:

```python
import torch
import torch.nn as nn
import torch.optim as optim

# Define a simple neural network
class SimpleNN(nn.Module):
    def __init__(self):
        super(SimpleNN, self).__init__()
        self.hidden = nn.Linear(32, 64)  # Input layer: 32 features, Hidden layer: 64 units
        self.output = nn.Linear(64, 1)  # Output layer: 1 unit
        self.activation = nn.ReLU()     # Activation function

    def forward(self, x):
        x = self.activation(self.hidden(x))  # Apply hidden layer and ReLU
        x = torch.sigmoid(self.output(x))    # Apply output layer and Sigmoid
        return x

# Initialize the model
model = SimpleNN()

# Define loss function and optimizer
criterion = nn.BCELoss()  # Binary Cross-Entropy Loss for binary classification
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Example data (replace with actual dataset)
x_train = torch.rand(100, 32)  # 100 samples, each with 32 features
y_train = torch.randint(0, 2, (100, 1)).float()  # 100 binary labels (0 or 1)

# Training loop
epochs = 10
for epoch in range(epochs):
    optimizer.zero_grad()           # Reset gradients
```

```
    predictions = model(x_train)     # Forward pass
    loss = criterion(predictions, y_train)  # Compute loss
    loss.backward()                  # Backward pass
    optimizer.step()                 # Update weights
    print(f"Epoch {epoch+1}/{epochs}, Loss: {loss.item():.4f}")
```

output –

```
[29]  ✓ 4.2s

Epoch 1/10, Loss: 0.6883
Epoch 2/10, Loss: 0.6859
Epoch 3/10, Loss: 0.6839
Epoch 4/10, Loss: 0.6823
Epoch 5/10, Loss: 0.6810
Epoch 6/10, Loss: 0.6799
Epoch 7/10, Loss: 0.6790
Epoch 8/10, Loss: 0.6782
Epoch 9/10, Loss: 0.6775
Epoch 10/10, Loss: 0.6768
```

7. How do you define a loss function and optimizer in PyTorch2

Defining Loss Function and Optimizer in PyTorch

Loss Function:

```
import torch.nn as nn
loss_function = nn.BCELoss()  # Binary Cross-Entropy Loss
```

Optimizer:

```
import torch.optim as optim
optimizer = optim.Adam(model.parameters(), lr=0.001)  # Adam Optimizer
```

Use in Training Loop:

```
for epoch in range(10):
    optimizer.zero_grad()                          # Reset gradients
    predictions = model(x_train)                   # Forward pass
    loss = loss_function(predictions, y_train)  # Compute loss
    loss.backward()                                # Backward pass
    optimizer.step()                               # Update weights
    print(f"Epoch {epoch+1}, Loss: {loss.item():.4f}")

[32]  ✓ 0.0s

Epoch 1, Loss: 0.6761
Epoch 2, Loss: 0.6750
Epoch 3, Loss: 0.6740
Epoch 4, Loss: 0.6729
Epoch 5, Loss: 0.6720
Epoch 6, Loss: 0.6710
Epoch 7, Loss: 0.6700
Epoch 8, Loss: 0.6690
Epoch 9, Loss: 0.6681
Epoch 10, Loss: 0.6671
```

This combines the loss function and optimizer for model training.

8. How do you implement a custom loss function in PyTorch2
   Ans –
   mplementing a Custom Loss Function in PyTorch
   You can create a custom loss function by subclassing torch.nn.Module or by writing a simple Python function
   Method 1: Using a Python Function
   Define a custom loss as a function that operates on tensors:

```
import torch

def custom_loss(output, target):
    return torch.mean((output - target) ** 2)  # Example: Mean Squared Error
```

Use it in the training loop:

```
for epoch in range(10):
    optimizer.zero_grad()
    predictions = model(x_train)
    loss = custom_loss(predictions, y_train)  # Use custom loss
    loss.backward()
    optimizer.step()
    print(f"Epoch {epoch+1}, Loss: {loss.item():.4f}")
```

```
4]  ✓ 0.1s
```

```
Epoch 1, Loss: 0.2366
Epoch 2, Loss: 0.2362
Epoch 3, Loss: 0.2357
Epoch 4, Loss: 0.2352
Epoch 5, Loss: 0.2348
Epoch 6, Loss: 0.2343
Epoch 7, Loss: 0.2339
Epoch 8, Loss: 0.2335
Epoch 9, Loss: 0.2330
Epoch 10, Loss: 0.2325
```

Method 2: Subclassing torch.nn.Module

Create a custom loss by subclassing torch.nn.Module for more complex logic:

```
import torch.nn as nn

class CustomLoss(nn.Module):
    def __init__(self):
        super(CustomLoss, self).__init__()

    def forward(self, output, target):
        return torch.mean((output - target) ** 2)  # Example: Mean Squared Error
```

```
loss_function = CustomLoss()
```

Use it in the same way as any other loss function:

```
for epoch in range(10):
    optimizer.zero_grad()
    predictions = model(x_train)
    loss = loss_function(predictions, y_train)
    loss.backward()
    optimizer.step()
    print(f"Epoch {epoch+1}, Loss: {loss.item():.4f}")
```

```
36]  ✓ 0.0s
```

```
Epoch 1, Loss: 0.2321
Epoch 2, Loss: 0.2316
Epoch 3, Loss: 0.2311
Epoch 4, Loss: 0.2307
Epoch 5, Loss: 0.2302
Epoch 6, Loss: 0.2297
Epoch 7, Loss: 0.2292
Epoch 8, Loss: 0.2286
Epoch 9, Loss: 0.2281
Epoch 10, Loss: 0.2276
```

This allows flexibility to define custom loss tailored to specific needs.

9. How do you save and load a TensorFlow model?

   Ans –

   Saving and Loading a TensorFlow Model

   1. Save the Model

   Use model.save() to save the entire model (architecture, weights, optimizer state).

```
# Save the model to a directory
model.save('my_model')  # Saved in TensorFlow SavedModel format
```

   To save in HDF5 format:

```
model.save('my_model.h5')  # HDF5 format
```

   2. Load the Model

   Use tf.keras.models.load_model() to load the saved model.

```
import tensorflow as tf

# Load the SavedModel
model = tf.keras.models.load_model('my_model')

# Load the HDF5 model
model = tf.keras.models.load_model('my_model.h5')
```

   The loaded model retains the architecture, weights, and optimizer configuration.