Deep Learning with Keras and TensorFlow

PyTorch

Learning Objectives

By the end of this lesson, you will be able to:

- Identify the applications of PyTorch libraries in enhancing computer vision and natural language processing tasks

- Optimize neural network models for image and sequence processing by selecting effective activation function

- Implement pooling and normalization techniques to improve the accuracy and efficiency of deep learning models

- Employ loss functions and optimizer modules to refine the training process and enhance model performance effectively

Business Scenario

XYZ Inc. is developing a real-time fraud detection system, utilizing the powerful PyTorch deep learning framework. The company plans to harness PyTorch's tensor class for effective data manipulation, coupled with the Autograd, Optim, and NN modules for robust model training and development.

To optimize performance on vast datasets, XYZ Inc. will implement data parallelism techniques across multiple GPUs. They recognize the importance of high-quality data. That is precisely why they select appropriate model architectures and apply specialized modules, like dropout and batch normalization, to manage overfitting and ensure model stability. These strategies are crucial for tackling the class imbalance typically associated with fraud detection tasks.

By integrating these advanced techniques, XYZ Inc. aims to significantly improve its fraud detection capabilities, thereby providing a more secure and reliable service to its customers.

Introduction to PyTorch

PyTorch

PyTorch, an open-source deep learning framework, is renowned for its capabilities in computer vision, natural language processing, and reinforcement learning.
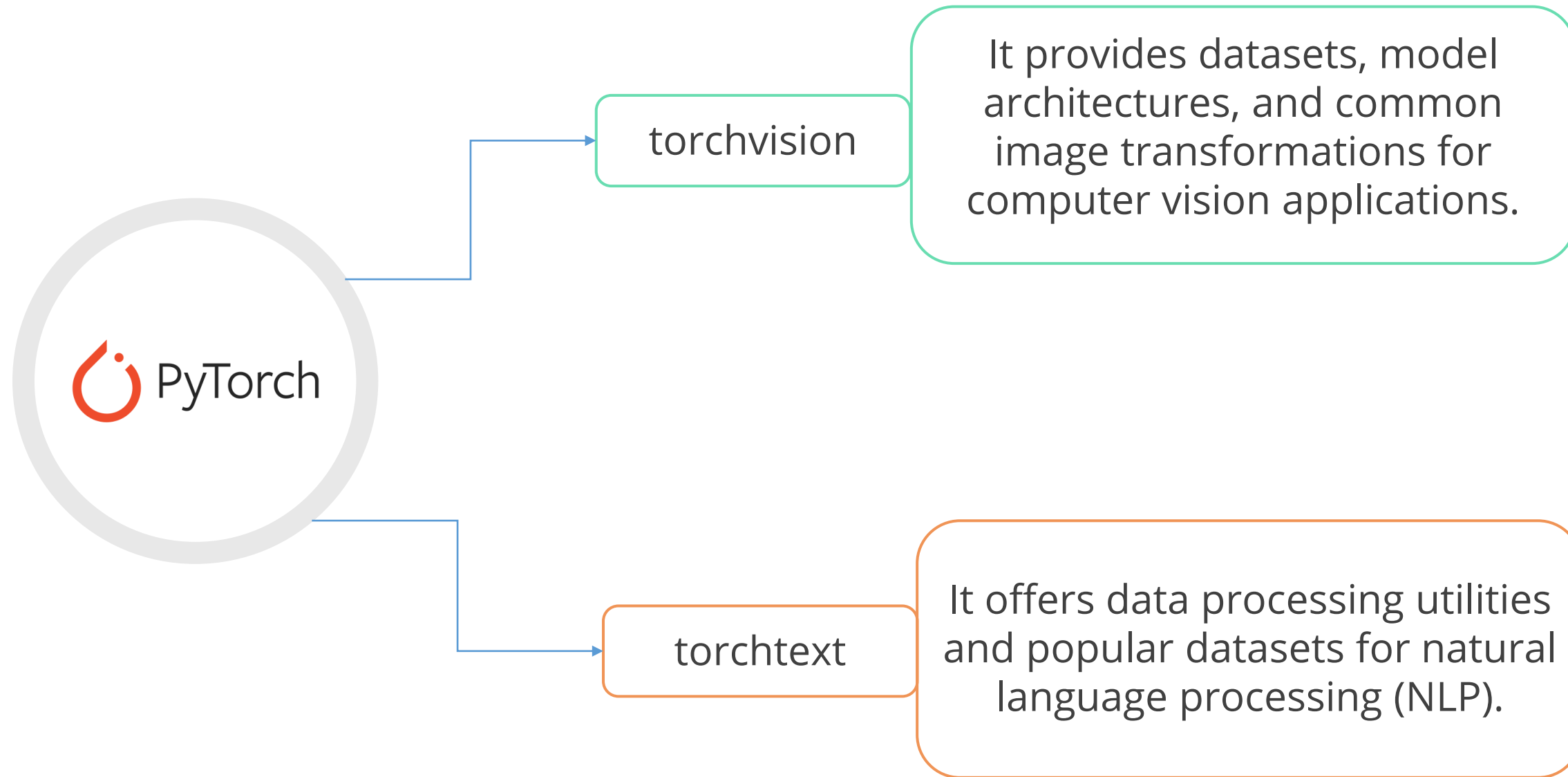
Characteristics of PyTorch

Has dynamic neural network development with a define-by-run paradigm

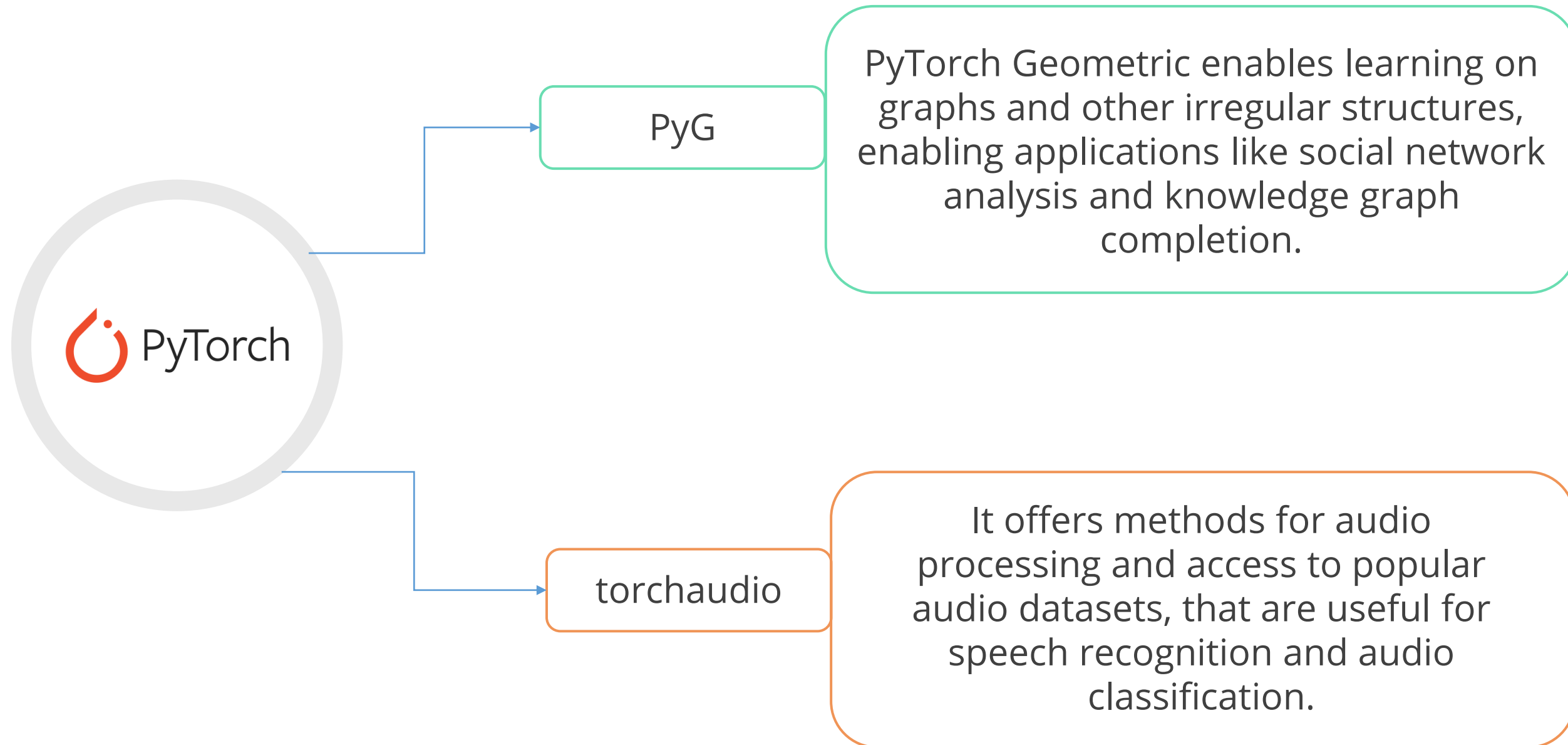Has strong GPU Support for optimized performance

Has seamless Python integration to enhance usability

PyTorch Ecosystems

PyTorch

torchvision

It provides datasets, model architectures, and common image transformations for computer vision applications.

torchtext

It offers data processing utilities and popular datasets for natural language processing (NLP).

PyTorch Ecosystems

PyG — PyTorch Geometric enables learning on graphs and other irregular structures, enabling applications like social network analysis and knowledge graph completion.

torchaudio — It offers methods for audio processing and access to popular audio datasets, that are useful for speech recognition and audio classification.

Installation of PyTorch

Installation of PyTorch

Use the following command to install PyTorch:

Syntax:-

```
pip install torch torchvision torchaudio
```

Installation of PyTorch

- Install PyTorch with CUDA support:

Syntax:-

```
pip install torch torchvision torchaudio --extra-index-url
https://download.pytorch.org/whl/cu117
```

- Install PyTorch with CUDA support using **conda**:

Syntax:-

```
conda install pytorch torchvision torchaudio cudatoolkit=11.3
-c pytorch
```

PyTorch Tensors

Tensors in PyTorch resemble NumPy's ndarrays. A tensor represents a multidimensional array composed of elements of a single data type.

The following code snippet shows how to create a tensor in PyTorch:

Syntax:-

```
import torch
# Create a tensor with random values
x = torch.rand(2, 3)
```

Modules in PyTorch

Each type of module in PyTorch serves a specific role in building and training neural network models. These roles range from structuring the network architecture to optimizing its parameters during the training process.

Basic Layer Modules

- These are the fundamental building blocks for creating neural networks.

- **Linear Layers (nn.Linear):** Fully connected linear layers

- **Convolutional Layers (nn.Conv1d, nn.Conv2d, nn.Conv3d):** Layers that apply a convolution operation over incoming data, typically used in image processing

- **Recurrent Layers (nn.LSTM, nn.GRU, nn.RNN):** Layers designed for sequential data processing

Activation Functions

- **ReLU (nn.ReLU**): A common activation function that outputs the input directly if it is positive; otherwise, it outputs zero

- **Sigmoid (nn.Sigmoid), Tanh (nn.Tanh),** and others like **LeakyReLU and ELU**

Modules in PyTorch

Pooling Layers

- Reduce the spatial dimensions of the input, which decreases the computational complexity and controls overfitting

- **MaxPooling (nn.MaxPool1d, nn.MaxPool2d**): Applies a max filter to subregions of the initial input

- **Average Pooling (nn.AvgPool1d, nn.AvgPool2d**): Computes the average of elements in a region of the input

Normalization Layers

- Are used to stabilize the learning process by normalizing the input

- **Batch Normalization (nn.BatchNorm1d, nn.BatchNorm2d):** Normalizes the input to have zero mean and unit variance across the batch

- **Layer Normalization (nn.LayerNorm):** Normalizes input across the features instead of the batch

Modules in PyTorch

Dropout Layers

- Dropout is a regularization technique to prevent overfitting by randomly dropping units during the training process.

- **Dropout (nn.Dropout):** It randomly zeroes some of the elements of the input tensor with a given probability**.**

Utility and Container Modules

- These are used to organize or combine other modules, facilitating model construction and management.

- **Sequential (nn.Sequential):** It is a simple sequential container to concatenate multiple modules.

- **ModuleList (nn.ModuleList):** It holds submodules in a list.

- **ModuleDict (nn.ModuleDict):** It holds submodules in a dictionary.

Modules in PyTorch

Loss Function Modules

- PyTorch provides modules for computing loss, which measure how well the model's predictions match the target data.

- **MSELoss (nn.MSELoss):** It measures the mean squared error between the target and output.

- **CrossEntropyLoss (nn.CrossEntropyLoss):** It computes cross-entropy loss between the target and output logits.

Optimizer Modules

- **SGD (optim.SGD), Adam (optim.Adam),** and **Adagrad (optim.Adagrad):** These are methods used to update the weights of the network during training based on the gradients.

Example: Building a Deep Learning Model with the Fashion-MNIST Dataset

Building a DL Model with the Fashion-MNIST Dataset

The Fashion-MNIST dataset, featuring 10 classes of fashion articles and 28x28 pixel images, is utilized in deep learning applications.

The steps to build a DL model with the Fashion-MNIST dataset:

01 Import libraries → Gather the required libraries

02 Load and explore the dataset → Initiate the dataset, preprocess it, and understand its structure and characteristics

Building a DL Model with the Fashion-MNIST Dataset

The steps to build a DL model with the Fashion-MNIST dataset:

03 Define the model architecture → Establish the structure of the neural network, including layers, neurons, and activation functions

04 Compile and train the model → Set up the model's learning process and train it on the dataset

06 Evaluate the model → Assess the performance of the trained model on a separate test dataset

Import the Library

Import essential libraries

Example:

```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
from torchvision import datasets, transforms
```

Load and Explore the Dataset

Load the Fashion-MNIST dataset, preprocess it, and explore its
structure

Example:

```python
# Data preprocessing: normalization
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])

# Loading the dataset
train_dataset = datasets.FashionMNIST(root='./data', train=True, download=True,
transform=transform)
test_dataset = datasets.FashionMNIST(root='./data', train=False, download=True,
transform=transform)
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=32,
shuffle=True)
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=32,
shuffle=False)
# Printing the shape of the datasets
print(f'Training data: {len(train_dataset)} samples')
print(f'Testing data: {len(test_dataset)} samples')
```

Output:

```
Training data: 60000 samples
Testing data: 10000 samples
```

Define the Model Architecture

Define simple CNN with one convolutional layer, one max pooling layer, and two fully connected layers

Example:

```python
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, 3, 1)  # 1 input channel, 32
output channels, 3x3 kernel
        self.pool = nn.MaxPool2d(2, 2)  # 2x2 pooling
        self.fc1 = nn.Linear(13*13*32, 100)  # Flattened dimensions
after pooling
        self.fc2 = nn.Linear(100, 10)  # 10 classes for FashionMNIST

    def forward(self, x):
        x = self.conv1(x)
        x = nn.ReLU()(x)
        x = self.pool(x)
        x = x.view(-1, 13*13*32)  # Flatten
        x = self.fc1(x)
        x = nn.ReLU()(x)
        x = self.fc2(x)
        return nn.Softmax(dim=1)(x)

model = CNN()
```

Compile and Train the Model

The model is compiled and trained for 10 epochs using the *Adam* optimizer and cross-entropy loss function.

Example:

```python
# Setting up the loss and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Training the model for 10 epochs
num_epochs = 10
for epoch in range(num_epochs):
    model.train()
    running_loss = 0.0
    for images, labels in train_loader:
        # Zero the parameter gradients
        optimizer.zero_grad()

        # Forward + backward + optimize
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()

    print(f"Epoch {epoch+1}/{num_epochs}, Loss: {running_loss/len(train_loader)}")
```

Output:

```
Epoch 1/10, Loss: 1.8133027095158896
Epoch 2/10, Loss: 1.620435905393929647
Epoch 3/10, Loss: 1.5715269567489625
Epoch 4/10, Loss: 1.5599591334025065
Epoch 5/10, Loss: 1.5515947381337483
Epoch 6/10, Loss: 1.545262050501656
Epoch 7/10, Loss: 1.5412239967981973
Epoch 8/10, Loss: 1.5354691167195638
Epoch 9/10, Loss: 1.5328466287612914
Epoch 10/10, Loss: 1.5303263650894166
```

Evaluate the Model

The model's performance is evaluated on the test set.

Example:

```python
# Evaluating the model
model.eval()
correct = 0
total = 0
with torch.no_grad():
    for images, labels in test_loader:
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

accuracy = 100 * correct / total
print(f"Model accuracy on test set: {accuracy}%")
```

Output:

```
Model accuracy on test set: 90.68%
```

Example: MNIST Digit Classifier

Example: MNIST Digit Classifier

Create a deep learning model with the MNIST dataset to predict the handwritten digits using TensorFlow

The steps to be followed are:

01 Import and load the dataset → Load the dataset, import the required libraries, normalize pixel values, and reshape images

02 Visualize the data → Visualize the seventh image from a batch of the MNIST training dataset

03 Define the model → Define a fully connected neural network, also known as a dense neural network or multi-layer perceptron

Example: MNIST Digit Classifier

The steps to be followed are:

04 Compile the model → Specify the optimizer and loss function

05 Fit the model → Train the model with specified epochs and batch sizes on the training data

06 Evaluate and predict → Evaluate model performance and use the trained model to make predictions on new or unseen data

Import and Load the Dataset

Import PyTorch and load the MNIST dataset of handwritten digits, 0 to 9, apply transformations (convert to tensor and normalize), and create data loaders

Example:

```python
import torch
import torchvision.transforms as transforms
from torchvision.datasets import MNIST
from torch.utils.data import DataLoader

# Define transformation: Convert image to tensor and normalize
transform = transforms.Compose([transforms.ToTensor(),
transforms.Normalize((0.5,), (0.5,))])

# Download and load the dataset
train_dataset = MNIST(root='./data', train=True, transform=transform,
download=True)
test_dataset = MNIST(root='./data', train=False, transform=transform)

train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False)
```

Visualize the Data

Visualize the seventh image from a batch of the MNIST training dataset and print its corresponding label
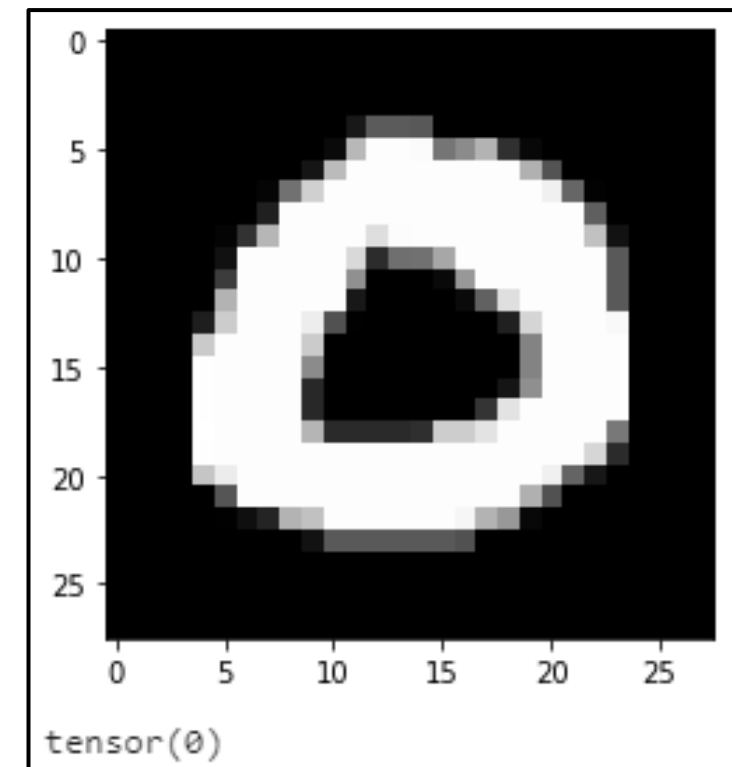
Example:

```python
import matplotlib.pyplot as plt

dataiter = iter(train_loader)
images, labels = next(dataiter)

plt.imshow(images[6].numpy().squeeze(),
cmap='gray')
plt.show()

print(labels[6])
```

Output:

tensor(0)

Define the Model

Define a neural network model with three fully connected layers

Example:

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

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(28*28, 128)
        self.fc2 = nn.Linear(128, 128)
        self.fc3 = nn.Linear(128, 10)

    def forward(self, x):
        x = x.view(-1, 28*28)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = F.softmax(self.fc3(x), dim=1)
        return x

model = Net()
```

Compile the Model

Define the loss function (cross-entropy loss) and optimizer (Adam)

Example:

```python
import torch.optim as optim

criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(),
lr=0.001)
```

Fit the Model

Train the model for 10 epochs using the training data

Example:

```python
epochs = 10
for epoch in range(epochs):
    running_loss = 0.0
    for i, (images, labels) in enumerate(train_loader, 1): # Added
enumeration to get batch number
        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()

    # Print average loss for the epoch
    print(f"Epoch [{epoch+1}/{epochs}], Loss:
{running_loss/i:.4f}")
```

Output:

```
Epoch [1/10], Loss: 1.6146
Epoch [2/10], Loss: 1.5399
Epoch [3/10], Loss: 1.5274
Epoch [4/10], Loss: 1.5198
Epoch [5/10], Loss: 1.5147
Epoch [6/10], Loss: 1.5099
Epoch [7/10], Loss: 1.5082
Epoch [8/10], Loss: 1.5073
Epoch [9/10], Loss: 1.5039
Epoch [10/10], Loss: 1.5025
```

Evaluate the Model

Evaluate the model's performance on the test set and print the accuracy

Example:

```python
correct = 0
total = 0
with torch.no_grad():
    for images, labels in test_loader:
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print(f'Test Accuracy: {100 * correct / total}%')
```

Output:

```
Test Accuracy: 95.55%
```

Predict the Model

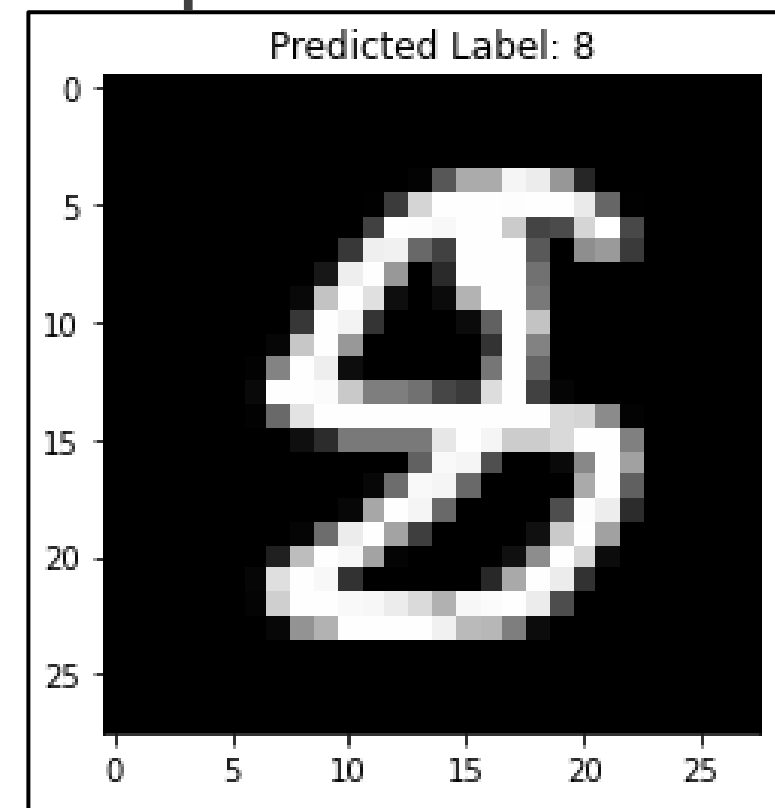The predict() function predicts the digit for the input.

Example:

```
dataiter = iter(train_loader)
images, labels = next(dataiter)

outputs = model(images[1:2])
_, predicted = torch.max(outputs.data, 1)

plt.imshow(images[1].numpy().squeeze(), cmap='gray')
plt.title(f"Predicted Label: {predicted.item()}")
plt.show()
```
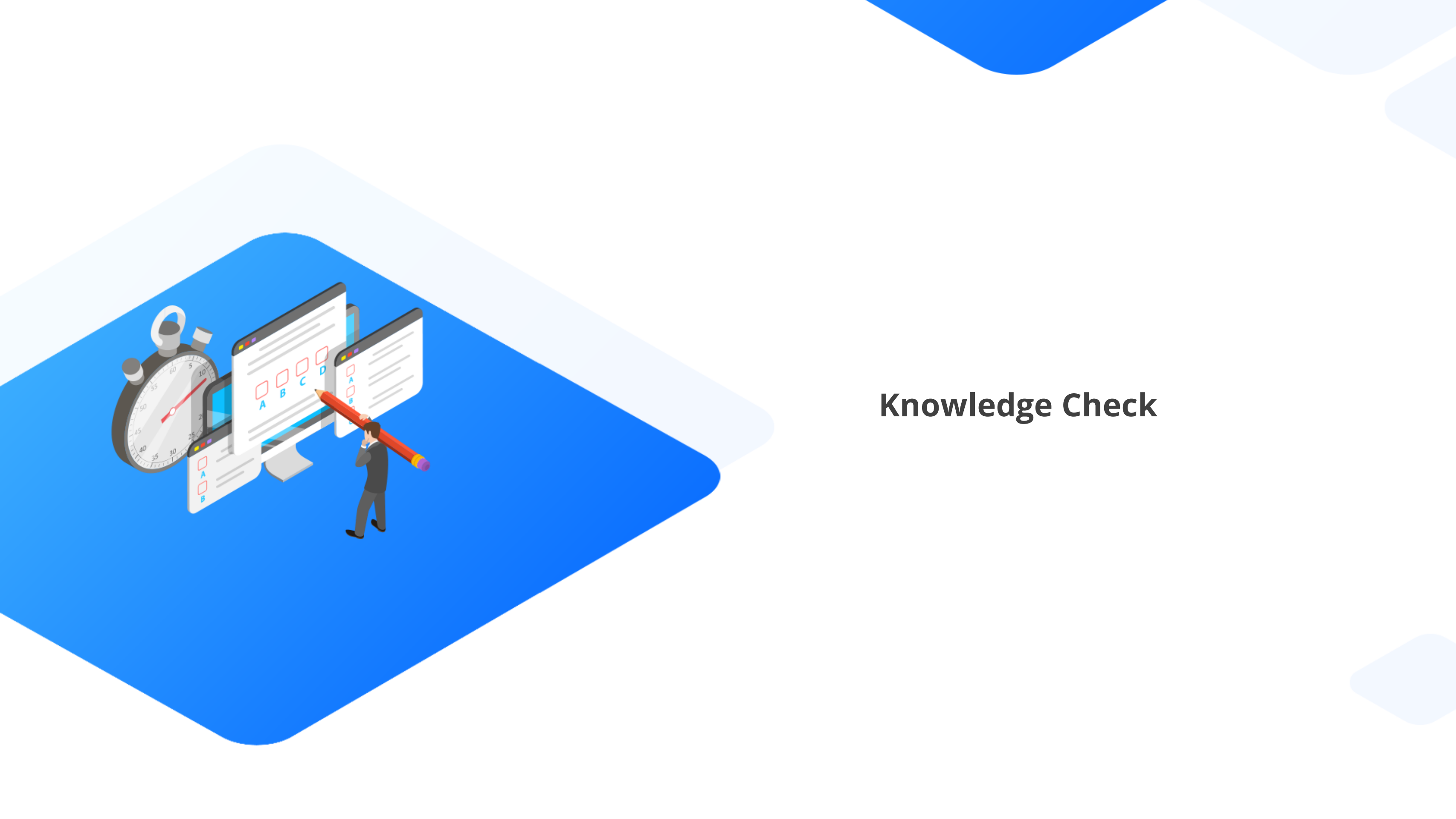
Output:

Predicted Label: 8

Key Takeaways

◉ PyTorch, an open-source deep learning framework, is renowned for its capabilities in computer vision, natural language processing, and reinforcement learning.

◉ PyTorch provides libraries tailored to various applications: torchvision for computer vision, torchtext for natural language processing, PyTorch Geometric for graph-based learning, and torchaudio for audio processing.

◉ PyTorch offers a variety of specialized modules for building and optimizing neural networks, such as basic layers, activation functions, pooling, and normalization layers.

Knowledge Check

What is the primary purpose of Convolutional Layers (nn.Conv1d, nn.Conv2d, and nn.Conv3d) in PyTorch?

A.   To output the input directly if it is positive and zero if otherwise

B.   To apply a convolution operation over incoming data, typically used in image processing

C.   To connect every neuron in one layer to every neuron in the next layer

D.   To create a feedback loop in the data, allowing for memory over time

What is the primary purpose of Convolutional Layers (nn.Conv1d, nn.Conv2d, and nn.Conv3d) in PyTorch?

A. To output the input directly if it is positive and zero if otherwise

B. To apply a convolution operation over incoming data, typically used in image processing

C. To connect every neuron in one layer to every neuron in the next layer

D. To create a feedback loop in the data, allowing for memory over time

The correct answer is **B**

Convolutional Layers in PyTorch apply filters to incoming data to detect features, primarily used in image and video processing tasks.

What is the purpose of the Optim module in PyTorch?

A.   To perform automatic differentiation

B.   To create computational graphs

C.   To distribute tasks on multiple CPUs or GPUs

D.   To implement optimization algorithms for building neural networks

What is the purpose of the Optim module in PyTorch?

A. To perform automatic differentiation

B. To create computational graphs

C. To distribute tasks on multiple CPUs or GPUs

D. To implement optimization algorithms for building neural networks

The correct answer is **D**

The Optim module in PyTorch is responsible for implementing optimization algorithms, that are used to build neural networks.

Thank You!