CV_Tasks

1.CV_Classification_Task:

Main_Steps:

1. Import Necessary Libraries
2. Data Preprocessing & Loading
3. Define the CNN Model
4. Define Loss Function & Optimizer
5. Training Loop
6. Evaluation
7. Result & visualization

7.Result & visualization

1. Import Necessary Libraries

```
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
from torch.utils.data import DataLoader
import torchvision.transforms as transforms
from torchvision.datasets import ImageFolder
from tqdm import tqdm
import numpy as np
import matplotlib.pyplot as plt
```

2.Data Preprocessing & Loading

```
transform = transforms.Compose([
    transforms.Resize((32, 32)),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])
```

- **Resizes the input image to 32x32 pixels**
- **Ensures all images have the same dimensions for batch processing**
- **Converts the image from PIL format to a PyTorch tensor**
- **Normalizes pixel values to the range [0,1] (from the original 0-255)**
- **(0.5, 0.5, 0.5) → Mean for each channel. (0.5, 0.5, 0.5) → Standard deviation for each channel.**
- **This scales the pixel values from [0,1] to [-1,1], making training more stable**

2.Data Preprocessing & Loading

```
train_dataset = ImageFolder(root='E:/archive/train', transform=transform)
test_dataset = ImageFolder(root='E:/archive/test', transform=transform)

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

- **Load Training and Testing Datasets**
- **Applies the transform (resizing, tensor conversion, and normalization) to each image**
- **Wraps the dataset to efficiently load batches of images during training/testing.**
- **shuffle=True : Randomizes the order of training samples to improve generalization.**
- **shuffle=False : Keeps the test data order fixed for consistent evaluation.**

3.Define the CNN Model

```python
class CNN(nn.Module):
    def __init__(self, num_classes=100):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 32, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.pool = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(64 * 8 * 8, 512)
        self.fc2 = nn.Linear(512, 100)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 64 * 8 * 8)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

3.Define the CNN Model

1. Define the CNN Architecture

- Inherits from nn.Module: Required for defining PyTorch models.
- num_classes=100: The model is designed to classify 100 different classes

2. Define CNN Layers:

Conv2d(3, 32, kernel_size=3, padding=1)

- 3 input channels (for RGB images).
- 32 output channels (filters) → extracts 32 feature maps.
- Kernel size = 3×3, with padding = 1 (to maintain the same size).

Conv2d(32, 64, kernel_size=3, padding=1)

- Takes 32 input feature maps (from conv1).
- Outputs 64 feature maps

Max Pooling (2×2)

- Reduces the size of feature maps by half.
- Helps reduce computational complexity and extract dominant features.

Fully Connected Layers

- First fc1: Maps flattened features (64 × 8 × 8) to 512 neurons.
- Second fc2: Maps 512 neurons to 100 output classe

3.Define the CNN Model

3. Forward Propagation

- F.relu(): Applies ReLU activation to introduce non-linearity.
- Pooling (self.pool): Reduces feature size after each convolution.
- Flattening (x.view(-1, 64 * 8 * 8)): Converts feature maps into a 1D vector.
- Fully Connected Layers (fc1 → fc2): Classifies the image into one of 100 classes.

Layer	Output Shape
Conv1 (3→32)	(32, 32, 32)
Pool1	(32, 16, 16)
Conv2 (32→64)	(64, 16, 16)
Pool2	(64, 8, 8)
Flatten	(64 × 8 × 8) = 4096
FC1 (4096 → 512)	512
FC2 (512 → 100)	100

4.Define Loss Function & Optimizer

```
model = CNN(num_classes=len(train_dataset.classes)).to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

1. Initialize the Model
2. Define the Loss Function:
   CrossEntropyLoss()
      It combines Softmax activation + Negative Log Likelihood (NLL) loss
3. Define the Optimizer:
      Adam Optimizer (optim.Adam)
      Learning Rate (lr=0.001)

5.Training Loop

```python
def train(model, train_loader, criterion, optimizer, epochs=5):
    model.train()
    for epoch in range(epochs):
        running_loss = 0.0
        for images, labels in tqdm(train_loader):
            images, labels = images.to(device), labels.to(device)
            optimizer.zero_grad()
            outputs = model(images)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
            running_loss += loss.item()
        avg_loss = running_loss / len(train_loader)
        train_losses.append(avg_loss)
        print(f"Epoch {epoch+1}, Loss: {avg_loss:.4f}")
```

6.Evaluation

```
def test(model, test_loader):
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for images, labels in test_loader:
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
    accuracy = 100 * correct / total
    test_accuracies.append(accuracy)
    print(f'Accuracy of the model on the test images: {accuracy:.2f}%')
```

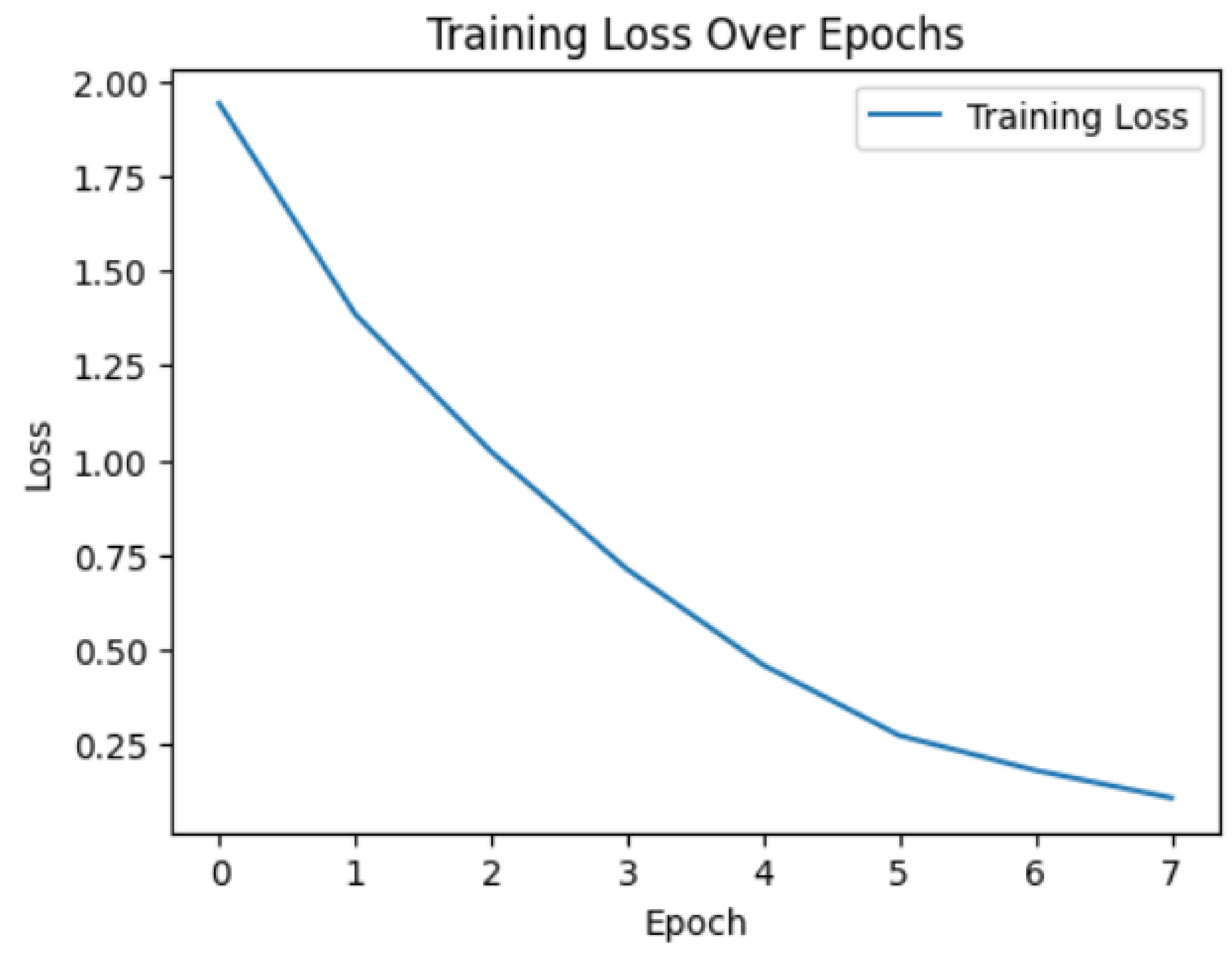
7.Result & visualization

```
train_losses = []
test_accuracies = []
epochs = 8
for epoch in range(epochs):
    print(f"Epoch {epoch+1}/{epochs}")
    train
(model, train_loader, criterion, optim
izer,
epochs=1)
    test(model, test_loader)
```

Epoch	Training Lo	Test Accuracy
1	1.9409	57.80%
2	1.3854	66.60%
3	1.0225	67.80%
4	0.7121	70.00%
5	0.4597	70.20%
6	0.2739	71.20%
7	0.1821	73.80%
8	0.1095	72.40%

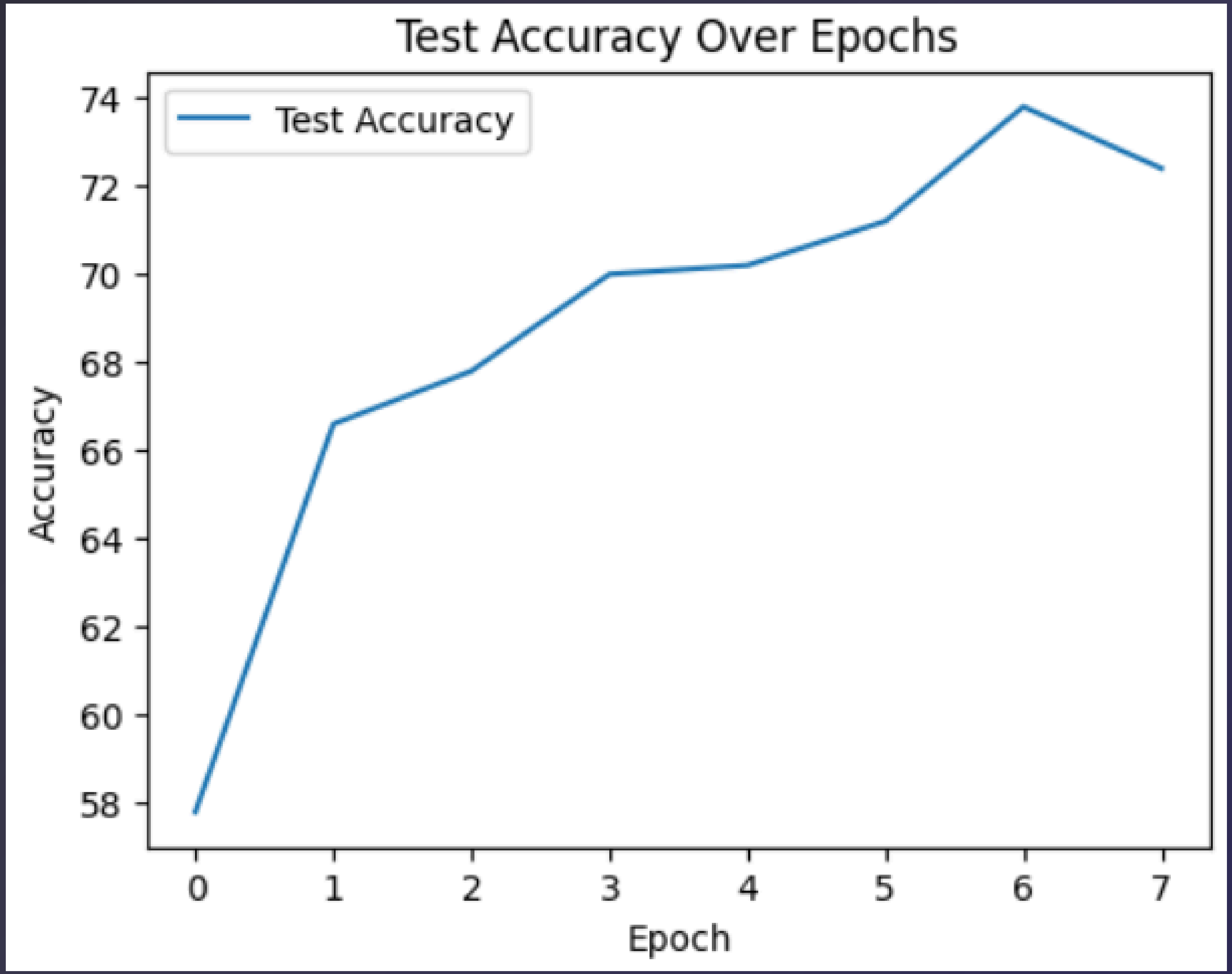
7.Result & visualization

```
import matplotlib.pyplot as plt
plt.figure(figsize=(12, 4))
plt.subplot(1, 2, 1)
plt.plot(train_losses, label=
'Training Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training Loss Over Epochs')
plt.legend()
```



7.Result & visualization

```
plt.figure(figsize=(12, 4))
plt.subplot(1, 2, 2)
plt.plot(test_accuracies, label=
'Test Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.title('Test Accuracy Over Epochs')
plt.legend()
plt.show()
```



2.Object_Detection_Task:

Main_Steps:

1. Setup Environment
2. Object Detection
3. Optimization
4. Visualizationr

1. Setup Environment

```
from ultralytics import YOLO
model = YOLO("yolov8n.pt")
import ultralytics
from IPython.display import display, Image
import cv2
import sys
import numpy as np
import matplotlib.pyplot as pl
```

2. Object Detection

```
results = model("C:/Users/MGM/Downloads/IMG-20241118-WA0117.jpg", show=True)
```

3.Optimization

```
bbox = results[0].boxes.xyxy[0]
class_id = results[0].boxes.cls[0]
score = results[0].boxes.conf[0]
class_name = model.names[int(class_id)]
```

```
def show_box(box, ax, class_name, score):
    x0, y0 = box[0], box[1]  #
    w, h = box[2] - box[0], box[3] - box[1]
    ax.add_patch(plt.Rectangle((x0, y0), w, h, edgecolor='green', facecolor=(0, 0, 0, 0), lw=2))
    ax.text(x0, y0 - 10, f'{class_name} ({score:.2f})', color='green', fontsize=12, fontweight='bold')
```

4.Visualizationr

```
plt.figure(figsize=(10, 10))
plt.imshow(results[0].orig_img)
show_box(bbox, plt.gca
(), class_name, score)
plt.axis('off')
plt.show()
```

Thank You

-